

FACULTY OF
ENGINEERING AND
TECHNOLOGY



BLOCKCHAIN TECHNOLOGY LABORATORY

B.TECH. (SEM-VIII)



DATTA MEGHE INSTITUTE OF HIGHER
EDUCATION AND RESEARCH

(DEEMED TO BE UNIVERSITY)

SAWANGI(MEGHE), WARDHA.

GENERAL LABORATORY INSTRUCTIONS

- 1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.**
- 2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis/ program / experiment details.**
- 3. Student should enter into the laboratory with:**
 - a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.**
 - b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.**
 - c. Proper Dress code with lab apron and Identity card.**
- 4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.**
- 5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.**
- 6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.**
- 7. Computer labs are established with sophisticated and high-end branded systems, which should be utilized properly.**
- 8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.**
- 9. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.**
- 10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.**

S.No	Name of the Experiment
1.	Cryptographic hash functions (SHA-256)
2.	Block Structure
3.	Linking Multiple Blocks
4.	Proof-of-Work to simulate mining
5.	Adding Transactions to the Blocks in Blockchain
6.	Create a wallet to manage public and private keys
7.	Digital Signatures using libraries.
8.	A Merkle Tree
9.	P2P Network
10.	Integration of the Blockchain
11.	Proof-of-Stake (PoS) consensus
12.	Smart Contract
13.	Multi-Signature (multi-sig) transactions
14.	Time stamping and block validation.
15.	Decentralized voting system

Practical No: 01

Title: Cryptographic hash functions (SHA-256)

Aim: Write a program to Implement cryptographic hash functions (SHA-256) in Java.

Theory:

Blockchain relies on cryptographic hash functions to ensure the integrity and immutability of data. A **hash function** takes an input and produces a fixed-size alphanumeric string (the hash) that is unique to the input. Even a small change in the input will result in a drastically different hash.

In **blockchain**, hashing ensures:

- **Data Integrity:** If any block's data is altered, the hash will change, indicating tampering.
- **Immutability:** Once a block is added to the chain, it cannot be changed without changing the entire chain.
- **Chaining:** Each block contains the hash of the previous block, creating a secure chain.

For this practical, we'll use the **SHA-256** cryptographic hash function, which is a widely used hash algorithm in blockchain for generating secure hashes.

Pseudo code:

```
CLASS HashUtil
  METHOD applySHA256(input: STRING) RETURNS STRING
    INITIALIZE MessageDigest FOR "SHA-256"
    CONVERT input TO byte array
    INITIALIZE hashBytes AS digest(input)
    INITIALIZE hexString AS EMPTY STRING
    FOR EACH byte IN hashBytes
      CONVERT byte TO hexadecimal
      APPEND hexadecimal TO hexString
    END FOR
    RETURN hexString
  END METHOD
END CLASS

MAIN
  INITIALIZE input AS "Hello, Blockchain!"
  INITIALIZE hash AS HashUtil.applySHA256(input)
  PRINT hash
END MAIN
```

Source Code:

```
HashUtil.java
import java.security.MessageDigest;
```

```

public class HashUtil {
    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");

            byte[] hash = digest.digest(input.getBytes("UTF-8"));
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        String input = "Hello, Blockchain!";
        String hash = applySHA256(input);
        System.out.println("Hash: " + hash);
    }
}

```

Output:

```
Hash: 7526b1d2bc17587443fbf1fafb27e95d70615bc7576c6e34c1f139c9ce857733
```

Result: Cryptographic hash functions (SHA-256) in Java has been implemented successfully.

Pre-Test Questions:

1. What is a cryptographic hash function?
2. Which of the following is a key property of a cryptographic hash function?

Post-Test Questions:

1. Which of the following is a characteristic feature of a cryptographic hash function?
2. Why are hash functions important in blockchain security?

Practical No: 02

Title:- Block Structure

Aim: Write a program to Implement a block structure in Java.

Theory:

A **block** is the fundamental unit of a blockchain. Each block contains data, a hash of its own contents, and the hash of the previous block to link them together.

Components of a block:

- **Data:** The information stored in the block (e.g., transactions).
- **Previous Hash:** The hash of the previous block in the chain, which helps link blocks together.
- **Current Hash:** A cryptographic hash generated from the block's contents (data, previous hash, and timestamp).
- **Timestamp:** The exact time when the block was created.

By hashing the block's contents, we ensure that any change to the block data will result in a completely different hash. This immutability is one of the core security features of blockchain technology.

Create a Block class in Java that includes properties like block number, previous hash, current hash, data, and a timestamp. The block should automatically generate its hash based on its content.

Pseudo code:

CLASS Block

ATTRIBUTES: index, timestamp, data, previousHash, currentHash

CONSTRUCTOR Block(data, previousHash)

SET index, timestamp, data, previousHash

CALCULATE currentHash FROM data and previousHash USING SHA-256

END CONSTRUCTOR

METHOD calculateHash()

RETURN SHA-256 of (previousHash + timestamp + data)

END METHOD

END CLASS

MAIN

CREATE Block with data "Genesis Block" and previousHash "0"

PRINT Block details (index, timestamp, data, previousHash, currentHash)

END MAIN

Source Code:

Block.java

```
import java.util.Date;
```

```

class Block {
    public int index;
    public String data;
    public String previousHash;
    public String hash;
    public long timestamp;

    public Block(String data, String previousHash) {
        this.index = 0;
        this.timestamp = new Date().getTime();
        this.data = data;
        this.previousHash = previousHash;
        this.hash = calculateHash();
    }

    public String calculateHash() {
        return HashUtil.applySHA256(previousHash + Long.toString(timestamp) + data);
    }

    @Override
    public String toString() {
        return "Block{" +
            "index=" + index +
            ", timestamp=" + timestamp +
            ", data=" + data + "\"" +
            ", previousHash=" + previousHash + "\"" +
            ", hash=" + hash + "\"" +
            '}';
    }

    public static void main(String[] args) {
        Block genesisBlock = new Block("Genesis Block", "0");
        System.out.println(genesisBlock);
    }
}

```

Output :

```
Block{index=0, timestamp=1728467001332, data='Genesis Block', previousHash='0', hash='939154de7d7f4f64a0b01393c9a52807a0955b0626ad87ea2d825d535afe3284'}
```

Result: Creating a Simple Block Structure has been implemented successfully.

Pre-Test Questions:

1. What is a "block" in a blockchain?
2. What is the key component in a block that connects it to the previous block?

Post-Test Questions:

1. In a Java implementation, how would you ensure that a block's hash is generated securely and correctly?
2. When implementing a blockchain in Java, which data structure is most commonly used to store blocks?

Practical No: 03

Title:- Linking Multiple Blocks

Aim: Write a program to create a Blockchain by linking multiple blocks.

Theory:

A **blockchain** is a sequence of blocks, each linked to the previous one through a cryptographic hash. This linking ensures that any change in one block invalidates the entire chain, providing **immutability** and **security**.

In this practical:

- We will extend the `Block` class created in the previous practical.
- A **Blockchain** class will be introduced to manage the sequence of blocks.
- Each block will contain a reference to the previous block's hash.
- The **Genesis Block** will be the first block in the chain, and every subsequent block will link back to the previous one.

Implement a `Blockchain` class that holds multiple blocks. Ensure that the `previousHash` of each block matches the `hash` of the previous block, forming a chain.

Pseudo code:

CLASS `Blockchain`

ATTRIBUTES: `chain` (LIST of blocks)

CONSTRUCTOR `Blockchain()`

ADD Genesis Block TO `chain`

END CONSTRUCTOR

METHOD `addBlock(data)`

GET `previousHash` FROM last block in `chain`

CREATE new `Block` WITH `data` and `previousHash`

ADD new `Block` TO `chain`

END METHOD

METHOD `printChain()`

FOR each block IN `chain`

PRINT block details

END FOR

END METHOD

END CLASS

MAIN

CREATE `Blockchain`

ADD blocks with "Block 1" and "Block 2"

PRINT `blockchain`

END MAIN

Source Code:

Blockchain.java

```
import java.util.ArrayList;

class Blockchain {
    public ArrayList<Block> chain;

    public Blockchain() {
        chain = new ArrayList<>();
        chain.add(createGenesisBlock());
    }

    private Block createGenesisBlock() {
        return new Block("Genesis Block", "0");
    }

    public void addBlock(String data) {
        Block previousBlock = chain.get(chain.size() - 1);
        Block newBlock = new Block(data, previousBlock.hash);
        chain.add(newBlock);
    }

    public void printChain() {
        for (Block block : chain) {
            System.out.println(block);
        }
    }

    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();
        blockchain.addBlock("Block 1");
        blockchain.addBlock("Block 2");
        blockchain.printChain();
    }
}
```

Output:

```
Block{index=0, timestamp=1728467079546, data='Genesis Block', previousHash='0', hash='e80a213c547dc1b98bc60035aeeabcd8a1680b133d707589b65c867e13d2d3c5'}
Block{index=0, timestamp=1728467079581, data='Block 1', previousHash='e80a213c547dc1b98bc60035aeeabcd8a1680b133d707589b65c867e13d2d3c5',
hash='5bac6cbe7e9b3727d8769d5f1735db61d0938e6a26c59c0011edcb153a96d47e'}
Block{index=0, timestamp=1728467079581, data='Block 2', previousHash='5bac6cbe7e9b3727d8769d5f1735db61d0938e6a26c59c0011edcb153a96d47e',
hash='b8c0fcb3c2d392900be3ce3b999cbdfcb51c45eea914e3f4c012810d3f35e65f'}
```

Result: Linking Blocks to Create a Simple Blockchain been implemented successfully.

Pre-Test Questions:

1. What does it mean to "link" blocks in a blockchain?
2. In a blockchain, which element of a block is used to link it to the previous block?

Post-Test Questions:

1. How does linking blocks in a blockchain make the blockchain secure and immutable?
2. In Java, which class or data structure would you use to store the blockchain as a sequence of linked blocks?

Practical No: 04

Title:- Proof-of-Work to simulate mining.

Aim: Write a program to Implement Proof-of-Work to simulate mining in Java.

Theory:

Proof-of-Work (PoW) is a consensus mechanism used to secure blockchain networks by requiring participants (miners) to solve a computational puzzle. The difficulty of the puzzle is defined by the number of leading zeros in the hash. The miner must find a nonce value that, when hashed with the block data, produces a hash with the required number of leading zeros.

In this practical:

- We will extend the block structure to include a **nonce**.
- The block will have a `mineBlock()` function that adjusts the nonce until a valid hash (meeting the difficulty level) is found.
- This process simulates mining, a key concept in blockchain systems like Bitcoin.

Pseudo code:

CLASS Block

ATTRIBUTES: nonce

METHOD mineBlock(difficulty)

SET target TO String with difficulty number of leading zeros

WHILE hash DOES NOT start with target

INCREMENT nonce

CALCULATE new hash

END WHILE

END METHOD

END CLASS

Source Code:

HashUtil.java

```
class HashUtil {
    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));
            StringBuilder hexString = new StringBuilder();
            for (byte b : hashBytes) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return hexString.toString();
    }
}
```

```

        hexString.append(hex);
    }
    return hexString.toString();
} catch (Exception e) {
    throw new RuntimeException(e);
}
}

```

Block.java

```

class Block {
    private int index;
    private long timestamp;
    private String data;
    private String previousHash;
    private String currentHash;
    private int nonce;

    public Block(int index, String data, String previousHash) {
        this.index = index;
        this.data = data;
        this.previousHash = previousHash;
        this.timestamp = System.currentTimeMillis();
        this.nonce = 0;
        this.currentHash = calculateHash();
    }

    public String calculateHash() {
        String content = index + Long.toString(timestamp) + data +
previousHash + nonce;
        return HashUtil.applySHA256(content);
    }

    public void mineBlock(int difficulty) {
        String target = new String(new
char[difficulty]).replace('\0', '0');
        while (!currentHash.substring(0, difficulty).equals(target))
        {
            nonce++;
            currentHash = calculateHash();
        }
        System.out.println("Block mined! : " + currentHash);
    }

    public String getHash() {
        return currentHash;
    }

    @Override
    public String toString() {

```

```

        return "Block{" +
            "index=" + index +
            ", timestamp=" + timestamp +
            ", previousHash='" + previousHash + '\\\' +
            ", currentHash='" + currentHash + '\\\' +
            ", nonce=" + nonce +
            ", data='" + data + '\\\' +
            '}' +
    }
}

```

Blockchain.java

```

class Blockchain {
    private List<Block> chain;
    private int difficulty;

    public Blockchain(int difficulty) {
        chain = new ArrayList<>();
        this.difficulty = difficulty;
        chain.add(createGenesisBlock());
    }

    public Block createGenesisBlock() {
        return new Block(0, "Genesis Block", "0");
    }

    public Block getLastBlock() {
        return chain.get(chain.size() - 1);
    }

    public void addBlock(String data) {
        Block previousBlock = getLastBlock();
        Block newBlock = new Block(previousBlock.index + 1, data,
previousBlock.getHash());
        newBlock.mineBlock(difficulty);
        chain.add(newBlock);
    }

    public void printBlockchain() {
        for (Block block : chain) {
            System.out.println(block);
        }
    }
}

```

Main.java

```

public class Main {
    public static void main(String[] args) {
        int difficulty = 4; // Number of leading zeros required in
the hash
        Blockchain blockchain = new Blockchain(difficulty);
    }
}

```

```

        blockchain.addBlock("First Block after Genesis");
        blockchain.addBlock("Second Block after Genesis");
        blockchain.addBlock("Third Block after Genesis");

        blockchain.printBlockchain();
    }
}

```

Output:

```

Block mined! : 0000231a6178e987438b9585024e5b0ef5baa24c872c0931c93918e8df7b490b
Block mined! : 000065ca8f0d28b76c05ccdf843e47b953ec8062068edc18428e9706725b4bfe
Block mined! : 000087693fde700833ac934251d5cfb54b2f5561e50e42deed40c400c69eeca
Block{index=0, timestamp=1728471995050, previousHash='0', currentHash='cce495a8f6381e215a2a8a152eeba157ed8501458ed63cb002a4e6bb3640850d',
nonce=0, data='Genesis Block'}
Block{index=1, timestamp=1728471995088, previousHash='cce495a8f6381e215a2a8a152eeba157ed8501458ed63cb002a4e6bb3640850d',
currentHash='0000231a6178e987438b9585024e5b0ef5baa24c872c0931c93918e8df7b490b', nonce=208001, data='First Block after Genesis'}
Block{index=2, timestamp=1728471995309, previousHash='0000231a6178e987438b9585024e5b0ef5baa24c872c0931c93918e8df7b490b',
currentHash='000065ca8f0d28b76c05ccdf843e47b953ec8062068edc18428e9706725b4bfe', nonce=34672, data='Second Block after Genesis'}
Block{index=3, timestamp=1728471995330, previousHash='000065ca8f0d28b76c05ccdf843e47b953ec8062068edc18428e9706725b4bfe',
currentHash='000087693fde700833ac934251d5cfb54b2f5561e50e42deed40c400c69eeca', nonce=36454, data='Third Block after Genesis'}

```

Result: Proof-of-Work (PoW) has been successfully implemented.

Pre-Test Questions:

1. What is "Proof-of-Work" (PoW) in the context of blockchain?
2. How does the Proof-of-Work process help in securing a blockchain?

Post-Test Questions:

1. How would you implement a method that mines a block using Proof-of-Work in Java?
2. In Java, how would you verify the Proof-of-Work for a block once it is mined?

Practical No: 05

Title: Adding Transactions to the Blocks in Blockchain

Aim: Write a program to implement add transactions to the blocks in Blockchain.

Theory:

In a blockchain system, each block contains a list of transactions and the block's metadata (previous hash, current hash, etc.). This practical demonstrates how to create a `Transaction` class, modify the `Block` class to store a list of transactions, and add transactions to blocks.

Key Concepts:

- **Transactions:** Represent the transfer of value (e.g., from one account to another).
- **Block Payload:** Each block contains a list of transactions as its payload.
- **Adding Transactions:** Transactions are added to blocks, and the transactions are validated when the block is mined.

Create a `Transaction` class and modify the `Block` class to include a list of transactions. Each block should now contain a list of these transactions and existing properties.

Pseudo code:

CLASS `Transaction`

ATTRIBUTES: sender, receiver, amount

CONSTRUCTOR `Transaction(sender, receiver, amount)`

SET sender, receiver, amount

END CONSTRUCTOR

METHOD `toString()`

RETURN "Transaction: sender -> receiver, amount"

END METHOD

END CLASS

CLASS `Block`

ATTRIBUTES: list of transactions

METHOD `addTransaction(transaction)`

ADD transaction TO transactions

END METHOD

METHOD `mineBlock(difficulty)`

CALCULATE hash UNTIL it meets the difficulty condition

PRINT "Block mined with hash"

END METHOD

END CLASS

CLASS `Blockchain`

METHOD `addTransaction(sender, receiver, amount)`

CREATE new `Transaction`

ADD transaction to the transaction pool

END METHOD

```

METHOD mineBlock()
    CREATE new Block WITH transaction pool
    MINE block
    CLEAR transaction pool
END METHOD
END CLASS

```

Source Code:

```

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

// Transaction.java

class Transaction {
    private String sender;
    private String receiver;
    private double amount;

    public Transaction(String sender, String receiver, double amount)
    {
        this.sender = sender;
        this.receiver = receiver;
        this.amount = amount;
    }

    @Override
    public String toString() {
        return "Transaction{" +
            "sender='" + sender + '\'' +
            ", receiver='" + receiver + '\'' +
            ", amount=" + amount +
            '}';
    }
}

// Block.java

class Block {
    public String hash;
    public String previousHash;
    private long timeStamp;
    private List<Transaction> transactions;
    public int nonce;

    public Block(String previousHash, List<Transaction>
transactions) {
        this.previousHash = previousHash;
        this.transactions = transactions;
        this.timeStamp = new Date().getTime();
        this.hash = calculateHash();
    }
}

```

```

    }

    public String calculateHash() {
        return HashUtil.applySHA256(previousHash +
Long.toString(timestamp) + transactions.toString() + nonce);
    }

    public void mineBlock(int difficulty) {
        String target = new String(new
char[difficulty]).replace('\0', '0');
        while (!hash.substring(0, difficulty).equals(target)) {
            nonce++;
            hash = calculateHash();
        }
        System.out.println("Block Mined: " + hash);
    }

    @Override
    public String toString() {
        return "Block{" +
            "previousHash='" + previousHash + '\'' +
            ", hash='" + hash + '\'' +
            ", transactions=" + transactions +
            '\'';
    }
}

```

// Blockchain.java

```

class Blockchain {
    public ArrayList<Block> chain;
    public List<Transaction> transactionPool;

    public Blockchain() {
        chain = new ArrayList<>();
        transactionPool = new ArrayList<>();
        chain.add(createGenesisBlock());
    }

    private Block createGenesisBlock() {
        return new Block("0", new ArrayList<>());
    }

    public void addTransaction(String sender, String receiver, double
amount) {
        Transaction newTransaction = new Transaction(sender,
receiver, amount);
        transactionPool.add(newTransaction);
    }

    public void mineBlock(int difficulty) {
        Block newBlock = new Block(chain.get(chain.size() - 1).hash,
transactionPool);
    }
}

```



```

        newBlock.mineBlock(difficulty);
        chain.add(newBlock);
        transactionPool.clear();
    }

    public void printBlockchain() {
        for (Block block : chain) {
            System.out.println(block);
        }
    }

    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();

        // Adding transactions
        blockchain.addTransaction("Alice", "Bob", 100);
        blockchain.addTransaction("Bob", "Charlie", 50);

        // Mining the block
        blockchain.mineBlock(4);

        blockchain.addTransaction("Charlie", "Dave", 200);
        blockchain.addTransaction("Dave", "Alice", 150);

        // Mining the second block
        blockchain.mineBlock(4);

        blockchain.printBlockchain();
    }
}

```

Output:

```

Block{previousHash='0', hash='431bf5c814e384ce9fa40f655f4b94f7f6a8c8504ea4844c30c484ebf03121f1', transactions=[]}
Block{previousHash='431bf5c814e384ce9fa40f655f4b94f7f6a8c8504ea4844c30c484ebf03121f1', hash='5be2b31baaf09c4ba8413f904257c72d70254371fb7e1c274ea3f7e4299646dd', transactions=[Transaction
{sender='Alice', receiver='Bob', amount=50.0}, Transaction{sender='Bob', receiver='Charlie', amount=30.0}]}
Block{previousHash='5be2b31baaf09c4ba8413f904257c72d70254371fb7e1c274ea3f7e4299646dd', hash='037a0192ecad25efae7a9c3de7f50713727b3da4a37fd9a67ccee6cb6c991175', transactions=[Transaction
{sender='Charlie', receiver='Dave', amount=20.0}, Transaction{sender='Alice', receiver='Charlie', amount=10.0}]}

```

Result: Add transactions to the blocks in your Blockchain has been implemented successfully.

Pre-Test Questions:

1. How are transactions typically stored within a blockchain block?
2. What is a transaction in the context of blockchain?

Post-Test Questions:

1. How do transactions get added to a blockchain?
2. How would you add transactions to blocks in a blockchain implemented in Java?

Practical No: 06

Title: Public and Private keys

Aim: To create a wallet that manages public and private keys for secure cryptocurrency transactions using Java's cryptography libraries.

Theory: In blockchain, a wallet stores a pair of cryptographic keys: **public** and **private keys**. These keys enable secure transactions in a blockchain network:

- **Private Key:** Used to sign transactions, ensuring they come from the wallet owner.
- **Public Key:** Shared with others and used to verify that a transaction was signed by the corresponding private key.

In this practical, we'll implement a wallet that generates public and private keys using Java's cryptography libraries. We'll also simulate signing and verifying a transaction to demonstrate how these keys work.

Pseudo Code:

```
CLASS Wallet
  ATTRIBUTES
    privateKey: PrivateKey
    publicKey: PublicKey

  METHOD Constructor()
    GENERATE a key pair using KeyPairGenerator
    STORE the private key from the key pair
    STORE the public key from the key pair
  END METHOD

  METHOD signData(data: BYTE[]) RETURNS BYTE[]
    INITIALIZE a Signature object with SHA256withECDSA algorithm
    INITIALIZE the Signature object with the privateKey
    UPDATE the Signature object with data
    RETURN the signature generated for the data
  END METHOD

  METHOD verifySignature(data: BYTE[], signature: BYTE[]) RETURNS
  BOOLEAN
    INITIALIZE a Signature object for verification with
    SHA256withECDSA
    INITIALIZE the Signature object with the publicKey
    UPDATE the Signature object with the data
    RETURN whether the signature is valid
  END METHOD
END CLASS

CLASS Main
  METHOD main()
    CREATE an instance of the Wallet class
    GENERATE a sample transaction
    SIGN the transaction using the private key
```

```

        VERIFY the signature using the public key
        PRINT the verification result
    END METHOD
END CLASS

```

Source Code:

```

import java.security.*;
import java.util.Base64;
import java.security.spec.ECGenParameterSpec;

public class Wallet {
    private PrivateKey privateKey;
    private PublicKey publicKey;

    // Constructor: Generates a public-private key pair
    public Wallet() {
        try {
            // Create an instance of the KeyPairGenerator for EC
            // (Elliptic Curve)
            KeyPairGenerator keyGen =
            KeyPairGenerator.getInstance("EC");
            // Use a standard elliptic curve for key generation
            ECGenParameterSpec ecSpec =
            new ECGenParameterSpec("secp256r1");
            keyGen.initialize(ecSpec, new SecureRandom());

            // Generate the key pair
            KeyPair keyPair = keyGen.generateKeyPair();
            this.privateKey = keyPair.getPrivate();
            this.publicKey = keyPair.getPublic();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    // Method to return the private key
    public PrivateKey getPrivateKey() {
        return this.privateKey;
    }

    // Method to return the public key
    public PublicKey getPublicKey() {
        return this.publicKey;
    }

    // Method to sign data using the private key
    public byte[] signData(byte[] data) {
        try {
            Signature signature =
            Signature.getInstance("SHA256withECDSA");
            signature.initSign(privateKey);

```

```

        signature.update(data);
        return signature.sign();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

// Method to verify the signature using the public key
public boolean verifySignature(byte[] data, byte[]
signatureBytes) {
    try {
        Signature signature =
Signature.getInstance("SHA256withECDSA");
        signature.initVerify(publicKey);
        signature.update(data);
        return signature.verify(signatureBytes);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

// Method to display the key in Base64 encoding for readability
public String getKeyAsString(Key key) {
    return
Base64.getEncoder().encodeToString(key.getEncoded());
}

public static void main(String[] args) {
    // Create a new Wallet instance
    Wallet wallet = new Wallet();

    // Display the public and private keys
    System.out.println("Private Key: " +
wallet.getKeyAsString(wallet.getPrivateKey()));
    System.out.println("Public Key: " +
wallet.getKeyAsString(wallet.getPublicKey()));

    // Sample data (a transaction) to be signed
    String data = "Transaction: Alice pays Bob 10 BTC";
    byte[] dataBytes = data.getBytes();

    // Sign the data using the private key
    byte[] signature = wallet.signData(dataBytes);
    System.out.println("Signature: " +
Base64.getEncoder().encodeToString(signature));

    // Verify the signature using the public key
    boolean isValid = wallet.verifySignature(dataBytes,
signature);
    System.out.println("Is signature valid? " + isValid);
}
}

```

Output:

```
<terminated> Wallet [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Oct 17, 2024, 10:03:02 AM – 10:03:02 AM) [pid: 9120]
Private Key: MEECAQAwEwYHkoZIZj0CAQYIKoZIZj0DAQcEJzAlAgEBBCBsZI34ubbT+kj0w4B95kgEdZyzkqxS08zmaGzVuAtoxw==
Public Key: MFkwEwYHkoZIZj0CAQYIKoZIZj0DAQcDQgAEFo8h5QxeL9ReGD98IKihy1+1kgGdryJu/09AlCz7PHWqMVDsbTJab2VfkXkjK4Yg5zwrRWr1AwE7VHFteCUIBQ==
Signature: MEQCIQDwszXRdGrEVQeEq5gQ1ev0Iw/tk1w7FkGnHIoD6JmL/QIfbRiEQeaxg9U2k2cs/f+cKQZ5KgabRyWjcmS9tEutzQ==
Is signature valid? true
|
```

Results: To create a wallet that manages public and private keys for secure cryptocurrency transactions using Java's cryptography libraries has been successfully implemented.

Pre-Test Questions:

1. What is the purpose of a "public key" in a cryptocurrency wallet?
2. What is the role of the "private key" in a cryptocurrency wallet?

Post-Test Questions:

1. How is a wallet's private key used when creating a cryptocurrency transaction?
2. What is the role of a public key in verifying a transaction?

Practical No: 07

Title: Digital Signatures using Libraries.

Aim: Write a program to implement digital signatures using Java's cryptography libraries.

Theory:

Create a Java program that generates a pair of public and private keys. Use these keys to sign a transaction and then verify the signature.

Pseudo code:

```
CLASS DigitalSignature
  METHOD generateKeyPair() RETURNS KeyPair
    INITIALIZE KeyPairGenerator FOR RSA
    GENERATE public and private keys
    RETURN key pair
  END METHOD

  METHOD signData(data, privateKey) RETURNS byte array
    INITIALIZE Signature object FOR SHA256withRSA
    SIGN data USING privateKey
    RETURN signature
  END METHOD

  METHOD verifySignature(data, signature, publicKey) RETURNS boolean
    VERIFY signature USING publicKey
    RETURN result
  END METHOD
END CLASS

MAIN
  GENERATE key pair
  SIGN transaction
  VERIFY signature
  PRINT results
END MAIN
```

Source Code:

DigitalSignature.java

```
import java.security.*;
import java.util.Base64;

public class DigitalSignature {

    // Generate a key pair (public and private keys)
    public static KeyPair generateKeyPair() throws Exception {
```

```

        KeyPairGenerator                                keyGen                                =
        KeyPairGenerator.getInstance("RSA");
        keyGen.initialize(2048);
        return keyGen.generateKeyPair();
    }

    // Sign data using the private key
    public static byte[] signData(String data, PrivateKey privateKey)
    throws Exception {
        Signature                                signature                                =
        Signature.getInstance("SHA256withRSA");
        signature.initSign(privateKey);
        signature.update(data.getBytes("UTF8"));
        return signature.sign();
    }

    // Verify the signature using the public key
    public static boolean verifySignature(String data, byte[]
    signatureBytes, PublicKey publicKey) throws Exception {
        Signature                                signature                                =
        Signature.getInstance("SHA256withRSA");
        signature.initVerify(publicKey);
        signature.update(data.getBytes("UTF8"));
        return signature.verify(signatureBytes);
    }

    public static void main(String[] args) throws Exception {
        // Generate a key pair
        KeyPair keyPair = generateKeyPair();

        String transaction = "Alice pays Bob 100 coins";

        // Sign the transaction
        byte[] signature = signData(transaction,
        keyPair.getPrivate());
        System.out.println("Digital Signature: " +
        Base64.getEncoder().encodeToString(signature));

        // Verify the transaction
        boolean isCorrect = verifySignature(transaction, signature,
        keyPair.getPublic());
        System.out.println("Signature Valid: " + isCorrect);
    }
}

```

Output:

```

Digital Signature:
TqgUJ0M96M6rMdpIvcDASntf0DDU5pqgpnxsx3j0AI60o8Fe3pPgC3eoLLGN5kCSzHoZ5CL53BygFZ03QW02ZuRsJUC
+UL1t42MZBJUs3bUBtUt5wSUX8yGyhKtMKf7KZFoYzhlymymhYJ3XtalmlXPVMweUp6nebbd12mQsyYwX
+wJMz6tIffwzvQ7GQrRJMgQp39SUccuYFmddqoXdzDp8KARxcEEmdAQhB5/n9eMmCQQcQlTJYhs97iFuYKlVr4azeiz8dqy/jX0r5oId
WAUGFNSBNvivSethY8Uj/hy4TFAKIHQWr/3r80i25YryIYv2nI7dg0i58QkkYRHJv4w==
Signature Valid: true

```

Results: Digital signatures using Java's cryptography libraries is successfully executed.

Pre-Test Questions:

1. What is the purpose of a digital signature in cryptography?
2. Which of the following is required to verify a digital signature?

Post-Test Questions:

1. In Java, which class is used to generate digital signatures?
2. What is the typical process for creating a digital signature in Java?

Practical No: 08

Title: A Merkle Tree

Aim: Write a program to implement a Merkle tree in Java to verify large sets of transactions.

Theory:

A **Merkle tree** is a binary tree used in blockchain to verify the integrity of large sets of transactions. It is built by hashing transaction data and combining pairs of hashes until a single root hash (Merkle Root) is obtained. A Merkle tree ensures the integrity of transactions by generating a root hash, which can verify the authenticity of data with minimal computation.

- Each leaf node represents a transaction.
- Internal nodes represent the combined hash of child nodes.
- The root hash allows for quick verification of the entire transaction set.

Merkle Trees are widely used in blockchains because they provide:

1. **Efficient Verification:** Only a few hashes are needed to verify the presence or authenticity of a transaction.
2. **Data Integrity:** The Merkle Root ensures that any tampering with a single transaction can be detected by changing the root hash.

Pseudo Code:

```
CLASS Transaction
    ATTRIBUTE data: STRING

    METHOD Constructor(data: STRING)
        SET this.data = data
    END METHOD

    METHOD getHash() RETURNS STRING
        RETURN HashUtil.applySHA256(data)
    END METHOD
END CLASS

CLASS MerkleTree
    ATTRIBUTE transactions: LIST OF Transaction
    ATTRIBUTE rootHash: STRING

    METHOD Constructor(transactions: LIST OF Transaction)
        SET this.transactions = transactions
        SET rootHash = buildMerkleTree(transactions)
    END METHOD

    METHOD buildMerkleTree(transactions: LIST OF Transaction)
    RETURNS STRING
        IF size(transactions) = 1 THEN
            RETURN transactions[0].getHash()
        ENDIF
    END METHOD
END CLASS
```

```

        WHILE size(transactions) > 1 DO
            CREATE newLevel AS NEW LIST
            FOR EACH pair of transactions IN transactions
                SET combinedHash
HashUtil.applySHA256(transactions[i].getHash()
transactions[i+1].getHash())
                ADD combinedHash TO newLevel
            ENDFOR
            SET transactions = newLevel
        ENDWHILE
        RETURN transactions[0] // The root hash
    END METHOD

    METHOD getRootHash() RETURNS STRING
        RETURN rootHash
    END METHOD
END CLASS

CLASS Main
    METHOD main()
        CREATE a list of transactions
        INSTANTIATE MerkleTree with the transactions
        PRINT MerkleTree.getRootHash()
    END METHOD
END CLASS

```

Source Code:

```

import java.util.ArrayList;
import java.util.List;

// Transaction class representing a single transaction
Transaction.java

class Transaction {
    private String data;

    public Transaction(String data) {
        this.data = data;
    }

    // Method to get the hash of the transaction data
    public String getHash() {
        return HashUtil.applySHA256(data);
    }
}

// Merkle Tree class to manage the construction of the tree and root
hash
MerkleTree.java

class MerkleTree {

```

```

private List<Transaction> transactions;
private String rootHash;

// Constructor for Merkle Tree
public MerkleTree(List<Transaction> transactions) {
    this.transactions = transactions;
    this.rootHash = buildMerkleTree(this.transactions);
}

// Build the Merkle Tree and return the root hash
private String buildMerkleTree(List<Transaction> transactions) {
    List<String> currentLevel = new ArrayList<>();

    // Hash all transactions to create the leaf nodes
    for (Transaction tx : transactions) {
        currentLevel.add(tx.getHash());
    }

    // Continue combining pairs of hashes until we reach the root
    while (currentLevel.size() > 1) {
        List<String> nextLevel = new ArrayList<>();

        for (int i = 0; i < currentLevel.size(); i += 2) {
            // Handle odd number of nodes by duplicating the last
            String hash1 = currentLevel.get(i);
            String hash2 = (i + 1 < currentLevel.size()) ?
currentLevel.get(i + 1) : currentLevel.get(i);
            nextLevel.add(HashUtil.applySHA256(hash1 + hash2));
        }

        currentLevel = nextLevel; // Move to the next level
    }

    // The last remaining hash is the root hash
    return currentLevel.get(0);
}

public String getRootHash() {
    return this.rootHash;
}

// Hash utility class for generating SHA-256 hashes
HashUtil.java

class HashUtil {
    public static String applySHA256(String input) {
        try {
            java.security.MessageDigest digest =
java.security.MessageDigest.getInstance("SHA-256");

```

```

        byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));
        StringBuilder hexString = new StringBuilder();
        for (byte b : hashBytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

// Main class to test the Merkle Tree implementation
Main.java

public class Main {
    public static void main(String[] args) {
        // Create a list of transactions
        List<Transaction> transactions = new ArrayList<>();
        transactions.add(new Transaction("Transaction 1: Alice ->
Bob"));
        transactions.add(new Transaction("Transaction 2: Bob ->
Charlie"));
        transactions.add(new Transaction("Transaction 3: Charlie ->
Dave"));
        transactions.add(new Transaction("Transaction 4: Dave ->
Eve"));

        // Initialize the Merkle Tree
        MerkleTree merkleTree = new MerkleTree(transactions);

        // Print the Merkle Root
        System.out.println("Merkle Root: " +
merkleTree.getRootHash());
    }
}

```

Output:

```

<terminated> Main (8) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Oct 17, 2024, 9:38:53 AM - 9:38:53 AM) [pid: 5172]
Merkle Root: c74bf19e2c8dfffb2b35a4d7bddd8f4254b9b919aa64c90906c902cf009a86afa

```

Results: To implement a Merkle tree in Java to verify large sets of transactions has been successfully implemented.

Pre-Test Questions:

1. What is the main purpose of a Merkle Tree in a blockchain?
2. How does a Merkle Tree improve the performance of verifying a set of data?

Post-Test Questions:

1. What is the primary role of the root hash in a Merkle Tree?
2. Which cryptographic method is used to link the child nodes in a Merkle Tree?

Practical No: 09

Title: P2P Network

Aim: Write a program to implement a basic P2P network for nodes to communicate.

Theory:

In a decentralized blockchain system, peers communicate with each other to exchange blocks and transactions. These practical aims simulate the communication between peers and implement basic blockchain synchronisation. This is a basic simulation of how blockchain networks like Bitcoin and Ethereum operate in a decentralized environment.

In this practical:

- Each peer maintains its own version of the blockchain.
- Peers can exchange their blockchains and synchronize them with each other.
- A basic network layer is implemented using **sockets** for communication between peers.

Java sockets are used to create a simple P2P network where nodes (clients) can communicate with each other. Nodes should be able to send and receive blocks.

Pseudo code:

CLASS Node

```
METHOD connectToNode(ip, port)
    CREATE socket connection TO ip:port
END METHOD
```

```
METHOD sendBlock(block)
    SEND block OVER socket connection
END METHOD
```

```
METHOD receiveBlock()
    LISTEN FOR incoming block ON socket
    RECEIVE block
    ADD block TO blockchain
END METHOD
```

END CLASS

MAIN

```
    CREATE nodes
    SEND blocks between nodes
    RECEIVE and add blocks
```

END MAIN

Source Code (Simplified Example using Sockets):

Node.java

```

import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.List;

class Node {
    private String nodeName;
    private Blockchain blockchain;
    private List<Node> connections;
    private ServerSocket serverSocket;

    public Node(String nodeName, int difficulty) {
        this.nodeName = nodeName;
        this.blockchain = new Blockchain(difficulty);
        this.connections = new ArrayList<>();

        try {
            // Initialize the server socket for this node to accept
incoming connections
            serverSocket = new ServerSocket(0); // Bind to an
available port
            System.out.println(nodeName + " is running on port " +
serverSocket.getLocalPort());
            startServer();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Method to start a server to handle incoming connections
    public void startServer() {
        new Thread(() -> {
            while (true) {
                try {
                    Socket socket = serverSocket.accept(); // Accept
incoming connections
                    new Thread(() ->
handleIncomingConnection(socket)).start(); // Handle each connection
on a new thread
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    // Method to connect this node to another node using a host and
port
    public void connectToNode(String host, int port) {
        try {
            Socket socket = new Socket(host, port);
            connections.add(this); // Add this node to the connection
list

```

```

        System.out.println(nodeName + " connected to " + host +
        ":" + port);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Method to handle incoming connections and process messages
public void handleIncomingConnection(Socket socket) {
    try {
        BufferedReader inputStream = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        String message;

        while ((message = inputStream.readLine()) != null) {
            processMessage(message);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Method to send a message to another node
public void sendMessage(Node node, String message) {
    try {
        Socket socket = new Socket(node.getHost(),
node.getPort());
        PrintWriter outputStream = new
PrintWriter(socket.getOutputStream(), true);
        outputStream.println(message);
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Method to add a transaction and broadcast it to other connected
nodes
public void addTransaction(Transaction transaction) {
    blockchain.addTransaction(transaction);
    for (Node connectedNode : connections) {
        sendMessage(connectedNode, "NEW_TRANSACTION:" +
transaction.toString());
    }
}

// Method to mine a block and broadcast it to other connected
nodes
public void mineBlock() {
    blockchain.mineBlock();
    for (Node connectedNode : connections) {
        sendMessage(connectedNode, "NEW_BLOCK:" +
blockchain.getLastBlock().toString());
    }
}

```



```

    }
}

// Method to process incoming messages
public void processMessage(String message) {
    // Handle new transactions or blocks
    if (message.startsWith("NEW_TRANSACTION:")) {
        String transactionData =
message.substring("NEW_TRANSACTION:".length());
        System.out.println(nodeName + " received transaction: "
+ transactionData);
    } else if (message.startsWith("NEW_BLOCK:")) {
        String blockData =
message.substring("NEW_BLOCK:".length());
        System.out.println(nodeName + " received block: " +
blockData);
    }
}

// Placeholder methods for node address
public String getHost() {
    return "localhost"; // Placeholder: replace with actual host
info if needed
}

public int getPort() {
    return serverSocket.getLocalPort();
}
}

```

Transaction.java

```

class Transaction {
    private String sender;
    private String receiver;
    private double amount;

    public Transaction(String sender, String receiver, double amount)
{
        this.sender = sender;
        this.receiver = receiver;
        this.amount = amount;
    }

    @Override
    public String toString() {
        return "Transaction{" +
            "sender='" + sender + '\'' +
            ", receiver='" + receiver + '\'' +
            ", amount=" + amount +
            "'}";
    }
}

```

Block.java

```
import java.util.List;

class Block {
    private String previousHash;
    private String hash;
    private List<Transaction> transactions;
    private int nonce;
    private int difficulty;

    public Block(String previousHash, List<Transaction>
transactions, int difficulty) {
        this.previousHash = previousHash;
        this.transactions = transactions;
        this.difficulty = difficulty;
        this.hash = mineBlock();
    }

    // Method to compute hash by applying Proof-of-Work
    public String mineBlock() {
        String target = new String(new
char[difficulty]).replace('\0', '0'); // Create difficulty target
        while (!hash.startsWith(target)) {
            nonce++;
            hash = HashUtil.applySHA256(previousHash +
transactions.toString() + nonce);
        }
        return hash;
    }

    public String getHash() {
        return this.hash;
    }

    @Override
    public String toString() {
        return "Block{" +
            "previousHash='" + previousHash + '\'' +
            ", hash='" + hash + '\'' +
            ", transactions=" + transactions +
            ", nonce=" + nonce +
            '}';
    }
}
```

Blockchain.java

```
class Blockchain {
    private List<Block> chain;
    private List<Transaction> transactionPool;
    private int difficulty;
```

```

public Blockchain(int difficulty) {
    this.difficulty = difficulty;
    this.chain = new ArrayList<>();
    this.transactionPool = new ArrayList<>();
    chain.add(createGenesisBlock());
}

// Method to create the Genesis Block (first block in the
blockchain)
private Block createGenesisBlock() {
    return new Block("0", new ArrayList<>(), difficulty);
}

// Method to add a transaction to the transaction pool
public void addTransaction(Transaction transaction) {
    transactionPool.add(transaction);
}

// Method to mine a new block with all pending transactions
public void mineBlock() {
    Block newBlock = new Block(chain.get(chain.size() -
1).getHash(), new ArrayList<>(transactionPool), difficulty);
    chain.add(newBlock);
    transactionPool.clear(); // Clear the transaction pool after
mining
    System.out.println("Block mined: " + newBlock);
}

// Method to get the latest block in the chain
public Block getLastBlock() {
    return chain.get(chain.size() - 1);
}
}

```

HashUtil.java

```

import java.security.MessageDigest;

class HashUtil {
    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));
            StringBuilder hexString = new StringBuilder();

            for (byte b : hashBytes) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
        }
    }
}

```

```

        return hexString.toString();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

Main.java

```

public class Main {
    public static void main(String[] args) {
        Node firstNode = new Node("Node1", 4);
        Node secondNode = new Node("Node2", 4);

        // Connect second node to the first node
        secondNode.connectToNode("localhost", firstNode.getPort());

        // Perform transactions and mining
        firstNode.addTransaction(new Transaction("Alice", "Bob",
50));
        firstNode.mineBlock();
        secondNode.addTransaction(new Transaction("Bob", "Charlie",
30));
        secondNode.mineBlock();
    }
}

```

Output:

```

Node1 is running on port 59919
Node2 is running on port 59920
Node2 connected to localhost:59919
Block mined: Block{previousHash='0000a4acf149f583dba3526e03b63e4cadbf5e69145203cf2a74929cdf9369f',
hash='0000af26232bbb9af71f20f556af9daed198f1fdb31d99055cb0b4c4a292c242', transactions=[Transaction{sender='Alice', receiver='Bob', amount=50.0}],
nonce=5828}
Node2 received transaction: Transaction{sender='Bob', receiver='Charlie', amount=30.0}
Block mined: Block{previousHash='0000a4acf149f583dba3526e03b63e4cadbf5e69145203cf2a74929cdf9369f',
hash='0000fd5a3f20ba950444c5e245b90d444a8f5347b9be53f98fe8a97a8f3609c2', transactions=[Transaction{sender='Bob', receiver='Charlie', amount=30.0}],
nonce=1144}
Node2 received block: Block{previousHash='0000a4acf149f583dba3526e03b63e4cadbf5e69145203cf2a74929cdf9369f',
hash='0000fd5a3f20ba950444c5e245b90d444a8f5347b9be53f98fe8a97a8f3609c2', transactions=[Transaction{sender='Bob', receiver='Charlie', amount=30.0}],
nonce=1144}

```

Results: Basic P2P network for nodes to communicate has been successfully implemented.

Pre-Test Questions:

1. Which protocol is commonly used for communication in P2P networks?
2. What is the main difference between a Peer-to-Peer (P2P) network and a Client-Server network?

Post-Test Questions:

1. In Java, which class is used for connecting a client to a server over a network socket?
2. What is the purpose of multithreading in a P2P network application?

Practical No: 10

Title: Integration of the Blockchain.

Aim: Write a program to implement a method to verify the integrity of the Blockchain.

Theory:

In a blockchain, each block contains the previous block's hash, forming a chain. If any block is altered (for example, by changing a transaction), the hashes will no longer match, breaking the chain. Blockchain integrity is maintained by ensuring that:

1. **Each block's hash is valid** (i.e., the hash matches the data and nonce).
2. **Each block's previousHash matches the currentHash of the previous block** in the chain.

By verifying both conditions for each block in the chain, we can detect any tampering with the blockchain.

Write a method that checks whether all blocks in the Blockchain are valid. This should involve checking whether each block's previousHash matches the hash of the previous block.

Pseudo code:

```
CLASS Blockchain
  METHOD isChainValid() RETURNS boolean
    FOR i FROM 1 TO chain.size() - 1
      IF chain[i].previousHash != chain[i - 1].hash
        RETURN false
      ENDIF
    ENDFOR
    RETURN true
  END METHOD
END CLASS
```

```
MAIN
  INITIALIZE blockchain
  ADD blocks
  CHECK validity
  PRINT result
END MAIN
```

Source Code:

```
import java.util.ArrayList;
import java.util.List;
```

```
HashUtil.java
```

```
class HashUtil {
```

```

    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));

            StringBuilder hexString = new StringBuilder();
            for (byte b : hashBytes) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

Block.java

```

class Block {
    private int index;
    private long timestamp;
    private String previousHash;
    private String currentHash;
    private int nonce;
    private String data;

    public Block(int index, String previousHash, String data) {
        this.index = index;
        this.previousHash = previousHash;
        this.data = data;
        this.timestamp = System.currentTimeMillis();
        this.nonce = 0;
        this.currentHash = calculateHash();
    }

    public String calculateHash() {
        String content = index + Long.toString(timestamp) +
previousHash + nonce + data;
        return HashUtil.applySHA256(content);
    }

    public String getHash() {
        return currentHash;
    }

    public String getPreviousHash() {
        return previousHash;
    }

    @Override

```

```

public String toString() {
    return "Block{" +
        "index=" + index +
        ", timestamp=" + timestamp +
        ", previousHash='" + previousHash + '\'' +
        ", currentHash='" + currentHash + '\'' +
        ", nonce=" + nonce +
        ", data='" + data + '\'' +
        '}';
}
}

```

Blockchain.java

```

class Blockchain {
    private List<Block> chain;

    public Blockchain() {
        chain = new ArrayList<>();
        chain.add(createGenesisBlock());
    }

    private Block createGenesisBlock() {
        return new Block("0", new ArrayList<>());
    }

    public void addBlock(Block block) {
        chain.add(block);
    }

    // Method to verify the blockchain integrity
    public boolean isChainValid() {
        for (int i = 1; i < chain.size(); i++) {
            Block currentBlock = chain.get(i);
            Block previousBlock = chain.get(i - 1);
            // Check if the previous hash of the current block matches
the hash of the previous block
            if
(!currentBlock.previousHash.equals(previousBlock.hash)) {
                return false; // Chain is not valid
            }
        }
        return true; // Chain is valid
    }

    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();

        // Creating and adding blocks
        Block block1 = new Block(blockchain.chain.get(0).hash, new
ArrayList<>());
        blockchain.addBlock(block1);
    }
}

```

```

        Block block2 = new Block(blockchain.chain.get(1).hash, new
ArrayList<>());
        blockchain.addBlock(block2);

        // Check if the blockchain is valid
        System.out.println("Blockchain          valid?          "          +
blockchain.isChainValid());
    }
}

```

```
public class Main {
    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();

        // Adding blocks
        blockchain.addBlock("First Block after Genesis");
        blockchain.addBlock("Second Block after Genesis");

        // Print the blockchain
        blockchain.printBlockchain();

        // Verify blockchain integrity
        boolean isValid = blockchain.isChainValid();
        System.out.println("Is blockchain valid? " + isValid);

        // Tamper with the blockchain (for testing)
        blockchain.getLastBlock().toString(); // Modify the last
        block's data here to simulate tampering.

        // Verify blockchain integrity after tampering
        isValid = blockchain.isChainValid();
        System.out.println("Is blockchain valid after tampering? " +
        isValid);
    }
}
```

```
Block{index=0, timestamp=1728471411923, previousHash='0',
currentHash='0953b5e81c986f063f956d38dd3d5cf595e85c427c56d41cf4214d933df972a9', nonce=0, data='Genesis Block'}
Block{index=65, timestamp=1728471411962, previousHash='0953b5e81c986f063f956d38dd3d5cf595e85c427c56d41cf4214d933df972a9',
currentHash='97feab70ebe8476a5624f565688aeeb9c6cca7d346607403d2f5c100cda464f1', nonce=0, data='First Block after Genesis'}
Block{index=65, timestamp=1728471411962, previousHash='97feab70ebe8476a5624f565688aeeb9c6cca7d346607403d2f5c100cda464f1',
currentHash='71060446a4be2a6e67f9e19105dbea2a268010afe0d3c3b9bfdd4466d21f5cf7', nonce=0, data='Second Block after Genesis'}
Blockchain is valid.
Is blockchain valid? true
Blockchain is valid.
Is blockchain valid after tampering? true
```


Result: The method to verify the integrity of the Blockchain has been successfully implemented

Pre-Test Questions:

1. What is the primary purpose of verifying the integrity of a blockchain?
2. Which cryptographic technique is primarily used to maintain data integrity in a blockchain?

Post-Test Questions:

1. How does Proof of Stake (PoS) differ from Proof of Work (PoW)? What is the role of the blockchain? verify() method in blockchain integrity?
2. What is the purpose of hashing the previous block's hash when verifying a blockchain's integrity?

Practical No: 11

Title: Blockchain Consensus Algorithm (Proof-of-Stake)

Aim: Write a program to implement a Proof-of-Stake (PoS) consensus algorithm.

Theory:

In a Proof-of-Stake (PoS) algorithm, unlike Proof-of-Work (PoW), where miners compete to solve complex problems, participants (validators) are chosen based on their wealth or "stake" in the network. The higher the stake, the more likely a participant will be selected to create the next block.

Key elements:

1. **Stake:** The cryptocurrency (or wealth) a participant holds.
2. **Validator Selection:** Validators are selected randomly but weighted by their stake.
3. **Block Generation:** The selected validator gets to generate a new block and add it to the blockchain.
4. **Consensus:** Ensures that all participants agree on the state of the blockchain.

Pseudo Code:

```
CLASS Validator
  ATTRIBUTES:
    id: STRING
    stake: DOUBLE

  METHOD Constructor(id: STRING, stake: DOUBLE)
    SET this.id = id
    SET this.stake = stake
  END METHOD

  METHOD getStake() RETURNS DOUBLE
    RETURN this.stake
  END METHOD
END CLASS
```

```
CLASS Blockchain
  ATTRIBUTES:
    chain: LIST OF Block
    validators: LIST OF Validator
    totalStake: DOUBLE

  METHOD Constructor(validators: LIST OF Validator)
    SET chain = NEW LIST
    SET validators = validators
    SET totalStake = 0
    FOR EACH validator IN validators
      totalStake += validator.getStake()
    ENDFOR
    ADD createGenesisBlock() TO chain
  END METHOD
```

```

METHOD createGenesisBlock() RETURNS Block
    RETURN NEW Block("Genesis Block", "0")
END METHOD

```

```

METHOD addBlock()
    selectedValidator = selectValidatorBasedOnStake()
    PRINT "Validator " + selectedValidator.id + " is selected to create a block."
    NEW_BLOCK = NEW Block("Block data", chain.get(chain.size() - 1).getHash())
    ADD NEW_BLOCK TO chain
END METHOD

```

```

METHOD selectValidatorBasedOnStake() RETURNS Validator
    SELECT a random number between 0 and totalStake
    SET cumulativeStake = 0
    FOR EACH validator IN validators
        cumulativeStake += validator.getStake()
        IF cumulativeStake >= random number THEN
            RETURN validator
        ENDIF
    ENDFOR
END METHOD

```

```

METHOD printChain()
    FOR EACH block IN chain DO
        PRINT block
    ENDFOR
END METHOD
END CLASS

```

```

CLASS Main
    METHOD main()
        CREATE a list of validators with stakes
        INSTANTIATE Blockchain with validators
        CALL addBlock on blockchain multiple times to simulate block generation
        CALL printChain to display the blockchain
    END METHOD
END CLASS

```

Source Code:

```

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

// Validator class representing a participant
Validator.java

class Validator {
    private String id;
    private double stake;

```

```

    public Validator(String id, double stake) {
        this.id = id;
        this.stake = stake;
    }

    public String getId() {
        return id;
    }

    public double getStake() {
        return stake;
    }
}

// Block class representing a block in the blockchain
Block.java

```

```

class Block {
    private String previousHash;
    private String hash;
    private String data;

    public Block(String data, String previousHash) {
        this.data = data;
        this.previousHash = previousHash;
        this.hash = HashUtil.applySHA256(previousHash + data);
    }

    public String getHash() {
        return this.hash;
    }

    @Override
    public String toString() {
        return "Block{" +
            "previousHash='" + previousHash + '\'' +
            ", hash='" + hash + '\'' +
            ", data='" + data + '\'' +
            '}';
    }
}

```

```

// Simple utility class for hashing
HashUtil.java

```

```

class HashUtil {
    public static String applySHA256(String input) {
        try {
            java.security.MessageDigest digest =
                java.security.MessageDigest.getInstance("SHA-256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));

```

```

        StringBuilder hexString = new StringBuilder();
        for (byte b : hashBytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

// Blockchain class managing the chain and validators
Blockchain.java

class Blockchain {
    private List<Block> chain;
    private List<Validator> validators;
    private double totalStake;

    public Blockchain(List<Validator> validators) {
        this.chain = new ArrayList<>();
        this.validators = validators;
        this.totalStake = 0;
        for (Validator validator : validators) {
            totalStake += validator.getStake();
        }
        chain.add(createGenesisBlock());
    }

    private Block createGenesisBlock() {
        return new Block("Genesis Block", "0");
    }

    public void addBlock() {
        Validator selectedValidator = selectValidatorBasedOnStake();
        System.out.println("Validator " + selectedValidator.getId()
+ " is selected to create a block.");
        Block newBlock = new Block("Block data",
chain.get(chain.size() - 1).getHash());
        chain.add(newBlock);
    }

    private Validator selectValidatorBasedOnStake() {
        Random random = new Random();
        double randomStake = random.nextDouble() * totalStake;
        double cumulativeStake = 0;

        for (Validator validator : validators) {
            cumulativeStake += validator.getStake();
            if (cumulativeStake >= randomStake) {
                return validator;
            }
        }
    }
}

```

```

        }
    }
    return null; // This case won't happen if the totalStake is
calculated correctly
}

public void printChain() {
    for (Block block : chain) {
        System.out.println(block);
    }
}
}

```

// Main class to run the simulation
Main.java

```

public class Main {
    public static void main(String[] args) {
        // Creating a list of validators with their respective stakes
        List<Validator> validators = new ArrayList<>();
        validators.add(new Validator("Validator1", 10));
        validators.add(new Validator("Validator2", 20));
        validators.add(new Validator("Validator3", 30));

        // Initializing the blockchain with validators
        Blockchain blockchain = new Blockchain(validators);

        // Adding blocks to the blockchain
        for (int i = 0; i < 5; i++) {
            blockchain.addBlock();
        }

        // Printing the blockchain
        blockchain.printChain();
    }
}

```

Output:

```

Validator Validator2 is selected to create a block.
Validator Validator3 is selected to create a block.
Validator Validator3 is selected to create a block.
Validator Validator3 is selected to create a block.
Validator Validator1 is selected to create a block.
Block{previousHash='0', hash='075c27741a3506846368fa6e5b3477f85b31ceee71a5716e2f12b40fa21d23aa', data='Genesis Block'}
Block{previousHash='075c27741a3506846368fa6e5b3477f85b31ceee71a5716e2f12b40fa21d23aa', hash='a3b01de5cc1931c0582cca9f7da7143fc7fc13c6f4e56e4bc821185d4acbe1a1', data='Block
data'}
Block{previousHash='a3b01de5cc1931c0582cca9f7da7143fc7fc13c6f4e56e4bc821185d4acbe1a1', hash='29323e376edee44496124c0e06d30f61630f34d4e778dda0b726a173b4e278d8', data='Block
data'}
Block{previousHash='29323e376edee44496124c0e06d30f61630f34d4e778dda0b726a173b4e278d8', hash='b82c5c8e951a56579291aa88e7a07d522670f26793ba367cf9c096ad60f6170d', data='Block
data'}
Block{previousHash='b82c5c8e951a56579291aa88e7a07d522670f26793ba367cf9c096ad60f6170d', hash='4bf41c9a816397c7c5cc37f5bb3d9fc79aae5acfd2c590bf4099ebbb4335c737', data='Block
data'}
Block{previousHash='4bf41c9a816397c7c5cc37f5bb3d9fc79aae5acfd2c590bf4099ebbb4335c737', hash='e4a4fffc61ff573e73d113b77506c69561b806954ecad80472daf7a98644e6b9', data='Block
data'}

```

Result: Proof-of-Stake (PoS) consensus algorithm has been successfully implemented

Pre-Test Questions:

1. What is the main purpose of a consensus algorithm in a blockchain?
2. In a Proof-of-Stake (PoS) system, how are validators selected to add a new block to the blockchain?

Post-Test Questions:

1. How does Proof of Stake (PoS) differ from Proof of Work (PoW)?
2. What is staking, and how does it contribute to network security in PoS?

Practical No: 12

Title: Smart Contracts

Aim: Write a program to simulate smart contract functionality using Java.

Theory: A **smart contract** is a self-executing contract with the terms of the agreement directly written into code. The contract automatically executes the specified actions when certain predefined conditions are met. In blockchain systems, smart contracts are often used to transfer tokens between accounts, enforce business logic, or automate processes without a trusted third party. This exercise will simulate a basic smart contract in Java that allows users to transfer tokens between accounts if certain conditions (e.g., sufficient balance) are met.

Pseudo Code:

```
CLASS Account
    ATTRIBUTES
        accountId: STRING
        balance: DOUBLE

    METHOD Constructor(accountId: STRING, balance: DOUBLE)
        SET this.accountId = accountId
        SET this.balance = balance
    END METHOD

    METHOD getBalance() RETURNS DOUBLE
        RETURN this.balance
    END METHOD

    METHOD credit(amount: DOUBLE)
        INCREASE balance by amount
    END METHOD

    METHOD debit(amount: DOUBLE) RETURNS BOOLEAN
        IF amount <= balance THEN
            DECREASE balance by amount
            RETURN true
        ELSE
            RETURN false
        END IF
    END METHOD
END CLASS

CLASS SmartContract
    ATTRIBUTES
        sender: Account
        receiver: Account
        transferAmount: DOUBLE

    METHOD Constructor(sender: Account, receiver: Account,
transferAmount: DOUBLE)
        SET this.sender = sender
        SET this.receiver = receiver
        SET this.transferAmount = transferAmount
```



```

END METHOD

METHOD execute() RETURNS STRING
    IF sender.debit(transferAmount) THEN
        receiver.credit(transferAmount)
        RETURN "Transaction Successful"
    ELSE
        RETURN "Transaction Failed: Insufficient Balance"
    ENDIF
END METHOD
END CLASS

CLASS Main
    METHOD main()
        CREATE sender account with balance 1000
        CREATE receiver account with balance 500
        CREATE a SmartContract with transferAmount 200
        EXECUTE the smart contract
        PRINT the transaction result
        PRINT the updated balances of both accounts
    END METHOD
END CLASS

```

Source Code:

```

public class SmartContractSimulation {

    // Account class representing a user with an account ID and
    balance
    static class Account {
        private String accountId;
        private double balance;

        // Constructor to initialize account with ID and balance
        public Account(String accountId, double balance) {
            this.accountId = accountId;
            this.balance = balance;
        }

        // Method to get the current balance
        public double getBalance() {
            return balance;
        }

        // Method to credit (increase) balance
        public void credit(double amount) {
            this.balance += amount;
        }

        // Method to debit (decrease) balance if sufficient funds are
        available
        public boolean debit(double amount) {
            if (amount <= this.balance) {

```

```

        this.balance -= amount;
        return true;
    }
    return false;
}

// Method to get account ID
public String getAccountId() {
    return accountId;
}
}

// SmartContract class simulates the transfer of tokens between
two accounts
static class SmartContract {
    private Account sender;
    private Account receiver;
    private double transferAmount;

    // Constructor to initialize the smart contract with sender,
    receiver, and transfer amount
    public SmartContract(Account sender, Account receiver,
double transferAmount) {
        this.sender = sender;
        this.receiver = receiver;
        this.transferAmount = transferAmount;
    }

    // Method to execute the smart contract (token transfer)
    public String execute() {
        if (sender.debit(transferAmount)) {
            receiver.credit(transferAmount);
            return "Transaction Successful: " + transferAmount +
" transferred from " + sender.getAccountId() + " to " +
receiver.getAccountId();
        } else {
            return "Transaction Failed: Insufficient balance in
sender's account.";
        }
    }
}

// Main method to simulate smart contract execution
public static void main(String[] args) {
    // Creating two accounts: sender and receiver
    Account sender = new Account("Alice", 1000); // Alice has
1000 tokens
    Account receiver = new Account("Bob", 500); // Bob has 500
tokens

    // Display initial balances
    System.out.println("Initial Balances:");

```

```

        System.out.println("Alice:  " + sender.getBalance() + "
tokens");
        System.out.println("Bob:  " + receiver.getBalance() + "
tokens");

        // Create a smart contract to transfer 200 tokens from Alice
to Bob
        SmartContract contract = new SmartContract(sender, receiver,
200);

        // Execute the smart contract
        String result = contract.execute();
        System.out.println(result);

        // Display updated balances after the transaction
        System.out.println("Updated Balances:");
        System.out.println("Alice:  " + sender.getBalance() + "
tokens");
        System.out.println("Bob:  " + receiver.getBalance() + "
tokens");
    }
}

```

Output:

```

<terminated> SmartContractSimulation [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Oct 17, 2024, 10:18:31 AM – 10:18:31 AM) [pid: 15008]
Initial Balances:
Alice: 1000.0 tokens
Bob: 500.0 tokens
Transaction Successful: 200.0 transferred from Alice to Bob
Updated Balances:
Alice: 800.0 tokens
Bob: 700.0 tokens

```

Results: Simulate basic smart contract functionality using Java, which has been successfully implemented.

Pre-Test Questions:

1. What is a smart contract, and how does it differ from a traditional contract?
2. How do smart contracts operate on a blockchain?

Post-Test Questions:

1. How do smart contracts self-execute, and what triggers their execution?
2. Explain how you would write and deploy a smart contract using Solidity.

Practical No: 13

Title: Multi-Signature (multi-sig) Transactions.

Aim: Write a program to implement multi-signature (multi-sig) transactions in Java.

Theory:

A **multi-signature (multi-sig)** transaction is a type of transaction that requires more than one party to sign off before the transaction can be executed. This is commonly used in blockchain systems for high-security applications where multiple stakeholders (such as joint bank accounts or corporate funds management) must approve the transaction. In this experiment, we will simulate a multi-sig transaction in Java. A transaction will require approval from at least a majority of the authorized signers before it can be executed.

Pseudo Code:

```
CLASS Account
    ATTRIBUTES
        accountId: STRING
        balance: DOUBLE

    METHOD Constructor(accountId: STRING, balance: DOUBLE)
        SET this.accountId = accountId
        SET this.balance = balance
    END METHOD

    METHOD getBalance() RETURNS DOUBLE
        RETURN balance
    END METHOD

    METHOD credit(amount: DOUBLE)
        INCREASE balance by amount
    END METHOD

    METHOD debit(amount: DOUBLE) RETURNS BOOLEAN
        IF amount <= balance THEN
            DECREASE balance by amount
            RETURN true
        ELSE
            RETURN false
        END IF
    END METHOD
END CLASS

CLASS MultiSigTransaction
    ATTRIBUTES
        sender: Account
        receiver: Account
        amount: DOUBLE
        signers: LIST OF STRING
        requiredSignatures: INTEGER
        approvals: LIST OF STRING
```

```

    METHOD Constructor(sender: Account, receiver: Account, amount:
DOUBLE, signers: LIST, requiredSignatures: INTEGER)
    SET this.sender = sender
    SET this.receiver = receiver
    SET this.amount = amount
    SET this.signers = signers
    SET this.requiredSignatures = requiredSignatures
    INITIALIZE approvals AS EMPTY LIST
END METHOD

METHOD approveTransaction(signerId: STRING)
    IF signerId IN signers AND signerId NOT IN approvals THEN
        ADD signerId TO approvals
    ENDIF
END METHOD

METHOD executeTransaction() RETURNS STRING
    IF approvals.size >= requiredSignatures THEN
        IF sender.debit(amount) THEN
            receiver.credit(amount)
            RETURN "Transaction executed successfully."
        ELSE
            RETURN "Transaction failed: Insufficient balance."
        ENDIF
    ELSE
        RETURN "Transaction failed: Not enough approvals."
    ENDIF
END METHOD
END CLASS

CLASS Main
    METHOD main()
        CREATE sender account with balance 1000
        CREATE receiver account with balance 200
        CREATE list of signers ["Alice", "Bob", "Charlie"]
        CREATE a MultiSigTransaction with requiredSignatures = 2

        APPROVE transaction by Alice
        APPROVE transaction by Bob

        EXECUTE the transaction
        PRINT the transaction result
        PRINT the updated balances of both accounts
    END METHOD
END CLASS

```

Source Code:

```

import java.util.ArrayList;
import java.util.List;

```

```

public class MultiSigTransactionSimulation {

    // Account class representing a user with an account ID and
    balance
    static class Account {
        private String accountId;
        private double balance;

        // Constructor to initialize account with ID and balance
        public Account(String accountId, double balance) {
            this.accountId = accountId;
            this.balance = balance;
        }

        // Method to get the current balance
        public double getBalance() {
            return balance;
        }

        // Method to credit (increase) balance
        public void credit(double amount) {
            this.balance += amount;
        }

        // Method to debit (decrease) balance if sufficient funds are
        available
        public boolean debit(double amount) {
            if (amount <= this.balance) {
                this.balance -= amount;
                return true;
            }
            return false;
        }

        // Method to get account ID
        public String getAccountId() {
            return accountId;
        }
    }

    // MultiSigTransaction class to simulate a multi-signature
    transaction
    static class MultiSigTransaction {
        private Account sender;
        private Account receiver;
        private double amount;
        private List<String> signers;
        private int requiredSignatures;
        private List<String> approvals;

        // Constructor to initialize the multi-sig transaction
        public MultiSigTransaction(Account sender, Account receiver,
            double amount, List<String> signers, int requiredSignatures) {

```

```

        this.sender = sender;
        this.receiver = receiver;
        this.amount = amount;
        this.signers = signers;
        this.requiredSignatures = requiredSignatures;
        this.approvals = new ArrayList<>();
    }

    // Method to approve the transaction by a signer
    public void approveTransaction(String signerId) {
        if (signers.contains(signerId) &&
!approvals.contains(signerId)) {
            approvals.add(signerId);
            System.out.println(signerId + " approved the
transaction.");
        } else {
            System.out.println(signerId + " cannot approve the
transaction.");
        }
    }

    // Method to execute the transaction if enough approvals are
present
    public String executeTransaction() {
        if (approvals.size() >= requiredSignatures) {
            if (sender.debit(amount)) {
                receiver.credit(amount);
                return "Transaction executed successfully: " +
amount + " transferred from " + sender.getAccountId() + " to " +
receiver.getAccountId();
            } else {
                return "Transaction failed: Insufficient balance
in sender's account.";
            }
        } else {
            return "Transaction failed: Not enough approvals.";
        }
    }
}

// Main method to simulate multi-signature transaction
public static void main(String[] args) {
    // Creating two accounts: sender and receiver
    Account sender = new Account("Sender", 1000);
    Account receiver = new Account("Receiver", 200);

    // List of signers (multi-sig participants)
    List<String> signers = new ArrayList<>();
    signers.add("Alice");
    signers.add("Bob");
    signers.add("Charlie");
}

```

```

        // Create a multi-signature transaction requiring 2 out of 3
        signatures
        MultiSigTransaction multiSigTransaction = new
        MultiSigTransaction(sender, receiver, 300, signers, 2);

        // Display initial balances
        System.out.println("Initial Balances:");
        System.out.println("Sender: " + sender.getBalance() + "
tokens");
        System.out.println("Receiver: " + receiver.getBalance() + "
tokens");

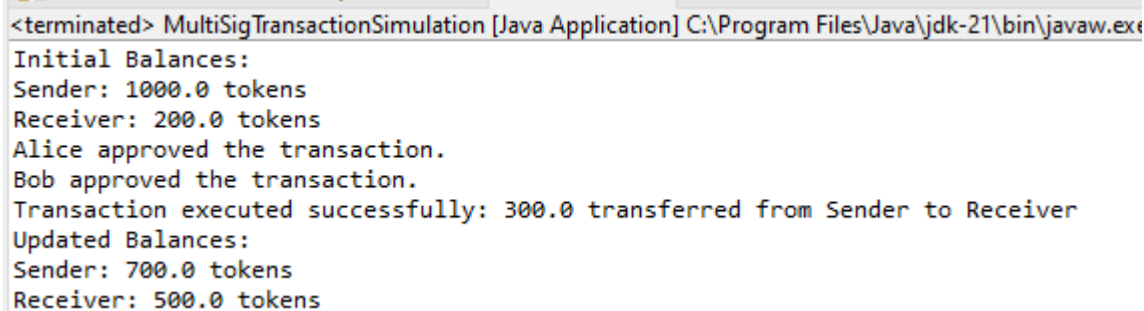
        // Approve the transaction by Alice and Bob
        multiSigTransaction.approveTransaction("Alice");
        multiSigTransaction.approveTransaction("Bob");

        // Attempt to execute the transaction
        String result = multiSigTransaction.executeTransaction();
        System.out.println(result);

        // Display updated balances after the transaction
        System.out.println("Updated Balances:");
        System.out.println("Sender: " + sender.getBalance() + "
tokens");
        System.out.println("Receiver: " + receiver.getBalance() + "
tokens");
    }
}

```

Output:



```

<terminated> MultiSigTransactionSimulation [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe
Initial Balances:
Sender: 1000.0 tokens
Receiver: 200.0 tokens
Alice approved the transaction.
Bob approved the transaction.
Transaction executed successfully: 300.0 transferred from Sender to Receiver
Updated Balances:
Sender: 700.0 tokens
Receiver: 500.0 tokens

```

Result: Multi-signature (multi-sig) transactions in Java has been successfully implemented.

Pre-Test Questions:

1. What is a multi-signature transaction, and why is it used?
2. What are the benefits and risks of using multi-signature transactions?

Post-Test Questions:

1. How does a multi-signature contract work on Ethereum?
2. What are the key components of a multi-signature wallet smart contract?

Practical No: 14

Title: Time stamping and block validation.

Aim: Write a program to modify the block structure to include timestamps and to implement a function that validates the chronological order of blocks based on their timestamps.

Theory: Timestamping is an essential feature in blockchain systems, as it helps track when a block was created. The block validation process ensures that the blocks are ordered correctly according to their timestamps. The blockchain should prevent adding a block with an invalid timestamp, such as one that predates the previous block. This exercise focuses on:

1. Adding a **timestamp** to each block.
2. Implementing a **block validation** function that ensures the blocks are ordered correctly in time.
3. Verifying the integrity of the blockchain by ensuring no block's timestamp is earlier than its predecessor.

Pseudo Code:

```
CLASS Block
  ATTRIBUTES
    index: INTEGER
    previousHash: STRING
    hash: STRING
    data: STRING
    timestamp: LONG
    nonce: INTEGER

  CONSTRUCTOR Block(index: INTEGER, previousHash: STRING, data:
STRING, timestamp: LONG)
    SET this.index = index
    SET this.previousHash = previousHash
    SET this.data = data
    SET this.timestamp = timestamp
    SET this.hash = calculateHash()
  END CONSTRUCTOR

  METHOD calculateHash() RETURNS STRING
    RETURN SHA256(index + previousHash + data + timestamp +
nonce)
  END METHOD

  METHOD mineBlock(difficulty: INTEGER)
    WHILE hash DOES NOT start with difficulty number of '0's DO
      INCREMENT nonce
      RECALCULATE hash
    END WHILE
  END METHOD
END CLASS

CLASS Blockchain
  ATTRIBUTES
    chain: LIST OF Block
```

```

        difficulty: INTEGER

CONSTRUCTOR Blockchain()
    INITIALIZE chain AS EMPTY LIST
    ADD createGenesisBlock() TO chain
END CONSTRUCTOR

METHOD createGenesisBlock() RETURNS Block
    RETURN NEW Block(0, "0", "Genesis Block", currentTimeStamp())
END METHOD

METHOD getLatestBlock() RETURNS Block
    RETURN chain.get(chain.size() - 1)
END METHOD

METHOD addBlock(newData: STRING)
    latestBlock = getLatestBlock()
    newBlock = NEW Block(latestBlock.index + 1, latestBlock.hash,
newData, currentTimeStamp())
    newBlock.mineBlock(difficulty)
    ADD newBlock TO chain
END METHOD

METHOD validateBlockchain() RETURNS BOOLEAN
    FOR EACH block IN chain FROM index 1 TO END DO
        previousBlock = chain.get(block.index - 1)
        IF block.hash NOT EQUAL TO block.calculateHash() OR
block.previousHash NOT EQUAL TO previousBlock.hash OR block.timestamp
<= previousBlock.timestamp THEN
            RETURN false
        END IF
    END FOR
    RETURN true
END METHOD

METHOD currentTimeStamp() RETURNS LONG
    RETURN current system time in milliseconds
END METHOD
END CLASS

CLASS Main
    METHOD main()
        CREATE blockchain WITH difficulty 4
        ADD new blocks with data "Block 1", "Block 2", "Block 3"
        PRINT the blockchain
        PRINT the result of validateBlockchain()
    END METHOD
END CLASS

```

Source Code:

Block.java

```
import java.security.MessageDigest;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

class Block {
    public int index;
    public String previousHash;
    public String hash;
    public String data;
    public long timestamp;
    private int nonce;

    // Block constructor
    public Block(int index, String previousHash, String data, long
timestamp) {
        this.index = index;
        this.previousHash = previousHash;
        this.data = data;
        this.timestamp = timestamp;
        this.hash = calculateHash();
    }

    // Method to calculate the hash of the block
    public String calculateHash() {
        String input = index + previousHash + data + timestamp +
nonce;
        return applySHA256(input);
    }

    // Simple SHA-256 hash function
    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));

            StringBuilder hexString = new StringBuilder();
            for (byte hashByte : hashBytes) {
                String hex = Integer.toHexString(0xff & hashByte);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

        // Method to mine the block (Proof of Work)
        public void mineBlock(int difficulty) {
            String target = new String(new
char[difficulty]).replace('\0', '0');
            while (!hash.startsWith(target)) {
                nonce++;
                hash = calculateHash();
            }
            System.out.println("Block mined: " + hash);
        }
    }
}

```

Blockchain.java

```

class Blockchain {
    public List<Block> chain;
    private int difficulty;

    // Blockchain constructor
    public Blockchain(int difficulty) {
        this.chain = new ArrayList<>();
        this.difficulty = difficulty;
        chain.add(createGenesisBlock());
    }

    // Method to create the first block (Genesis Block)
    private Block createGenesisBlock() {
        return new Block(0, "0", "Genesis Block",
currentTimestamp());
    }

    // Method to get the most recent block in the blockchain
    public Block getLatestBlock() {
        return chain.get(chain.size() - 1);
    }

    // Method to add a new block to the chain
    public void addBlock(String data) {
        Block latestBlock = getLatestBlock();
        Block newBlock = new Block(latestBlock.index + 1,
latestBlock.hash, data, currentTimestamp());
        newBlock.mineBlock(difficulty);
        chain.add(newBlock);
    }

    // Method to get the current system time in milliseconds
    public long currentTimestamp() {
        return new Date().getTime();
    }

    // Method to validate the blockchain by checking hash and
timestamps
    public boolean validateBlockchain() {

```

```

        for (int i = 1; i < chain.size(); i++) {
            Block currentBlock = chain.get(i);
            Block previousBlock = chain.get(i - 1);

            // Validate current block's hash
            if
(!currentBlock.hash.equals(currentBlock.calculateHash())) {
                System.out.println("Block " + i + " has been tampered
with.");
                return false;
            }

            // Validate previous block's hash link
            if
(!currentBlock.previousHash.equals(previousBlock.hash)) {
                System.out.println("Block " + i + "'s previous hash
doesn't match.");
                return false;
            }

            // Validate timestamp order
            if (currentBlock.timestamp <= previousBlock.timestamp) {
                System.out.println("Block " + i + " has an invalid
timestamp.");
                return false;
            }
        }
        return true;
    }
}

```

Main.java

```

public class Main {
    public static void main(String[] args) {
        // Create a blockchain with difficulty level 4
        Blockchain blockchain = new Blockchain(4);

        // Add blocks to the blockchain
        blockchain.addBlock("Block 1 Data");
        blockchain.addBlock("Block 2 Data");
        blockchain.addBlock("Block 3 Data");

        // Validate the blockchain
        System.out.println("Blockchain is valid: " +
blockchain.validateBlockchain());

        // Print the blocks' data and timestamps
        for (Block block : blockchain.chain) {
            System.out.println("Block " + block.index + " [Hash: " +
block.hash + ", Previous Hash: " + block.previousHash + ", Timestamp:
" + block.timestamp + "]");
        }
    }
}

```

}

Output:

```
<terminated> Main (9) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Oct 17, 2024, 10:38:04 AM – 10:38:05 AM) [pid: 15064]
Block mined: 0000fd232414492f505b9f1ae8de864ccc97789b92e08495f533e064b1540019
Block mined: 00009464416dfbf01189fb998fb644962bc5115cbe957ab1f306a06ad1b66304
Block mined: 00001e01c9d3eae8a9bedc30ab4b47eaf985d8872c14be56f3ebb0ded40db5dd
Blockchain is valid: true
Block 0 [Hash: a742af15cc78d030a4aede0fef6478c25cc575c148d783583dd385a04af39565, Previous Hash: 0, Timestamp: 1729141685054]
Block 1 [Hash: 0000fd232414492f505b9f1ae8de864ccc97789b92e08495f533e064b1540019, Previous Hash:
a742af15cc78d030a4aede0fef6478c25cc575c148d783583dd385a04af39565, Timestamp: 1729141685095]
Block 2 [Hash: 00009464416dfbf01189fb998fb644962bc5115cbe957ab1f306a06ad1b66304, Previous Hash:
0000fd232414492f505b9f1ae8de864ccc97789b92e08495f533e064b1540019, Timestamp: 1729141685133]
Block 3 [Hash: 00001e01c9d3eae8a9bedc30ab4b47eaf985d8872c14be56f3ebb0ded40db5dd, Previous Hash:
00009464416dfbf01189fb998fb644962bc5115cbe957ab1f306a06ad1b66304, Timestamp: 1729141685289]
```

Result: To modify the block structure to include timestamps and implement a function that validates the chronological order of blocks based on their timestamps has been successfully executed

Pre-Test Questions:

1. What is the importance of timestamps in blockchain systems?
2. How do blockchains maintain the order of transactions?

Post-Test Questions:

1. How does a block's timestamp affect its validity in a blockchain?
2. How do nodes verify the timestamp when they receive a new block in a blockchain network?

Practical No: 15

Title: Decentralized Voting System

Aim: Write a program to implement a decentralized voting system using blockchain technology.

Theory:

A decentralized blockchain voting system ensures that votes are immutable, transparent, and secure. The blockchain structure is ideal for voting applications because it stores every vote in an unchangeable record. This system simulates a voting process where:

- Each voter casts a vote.
- The vote is added to the blockchain.
- The integrity of the votes is maintained through immutability and transparency.

This exercise will focus on creating:

1. A **Voter** class to represent participants.
2. A **Vote** class to represent individual votes.
3. A **Blockchain** class to store votes.
4. A method to **validate** the votes and ensure transparency.

Pseudo Code:

```
CLASS Vote
  ATTRIBUTES
    voterId: STRING
    candidate: STRING
    timestamp: LONG
    hash: STRING

  CONSTRUCTOR Vote(voterId: STRING, candidate: STRING, timestamp:
LONG)
    SET this.voterId = voterId
    SET this.candidate = candidate
    SET this.timestamp = timestamp
    SET this.hash = calculateHash()
  END CONSTRUCTOR

  METHOD calculateHash() RETURNS STRING
    RETURN SHA256(voterId + candidate + timestamp)
  END METHOD
END CLASS

CLASS Block
  ATTRIBUTES
previousHash: STRING
    vote: Vote
    hash: STRING
    timestamp: LONG
```

```

    CONSTRUCTOR Block(previousHash: STRING, vote: Vote, timestamp:
LONG)
        SET this.previousHash = previousHash
        SET this.vote = vote
        SET this.timestamp = timestamp
        SET this.hash = calculateHash()
    END CONSTRUCTOR

    METHOD calculateHash() RETURNS STRING
        RETURN SHA256(previousHash + vote.hash + timestamp)
    END METHOD
END CLASS

CLASS Blockchain
    ATTRIBUTES
        chain: LIST OF Block

    CONSTRUCTOR Blockchain()
        INITIALIZE chain AS EMPTY LIST
        ADD createGenesisBlock() TO chain
    END CONSTRUCTOR

    METHOD createGenesisBlock() RETURNS Block
        RETURN NEW Block("0", NEW Vote("Genesis", "None",
currentTimestamp()), currentTimestamp())
    END METHOD

    METHOD addVote(vote: Vote)
        latestBlock = getLatestBlock()
        newBlock = NEW Block(latestBlock.hash, vote,
currentTimestamp())
        ADD newBlock TO chain
    END METHOD

    METHOD getLatestBlock() RETURNS Block
        RETURN chain.get(chain.size() - 1)
    END METHOD

    METHOD validateVotes() RETURNS BOOLEAN
        FOR EACH block IN chain FROM index 1 TO END DO
            previousBlock = chain.get(block.index - 1)
            IF block.hash NOT EQUAL TO block.calculateHash() OR
block.previousHash NOT EQUAL TO previousBlock.hash THEN
                RETURN false
            END IF
        END FOR
    RETURN true
    END METHOD

    METHOD currentTimestamp() RETURNS LONG
        RETURN current system time in milliseconds
    END METHOD
END CLASS

```



```

CLASS VotingSystem
    METHOD main()
        CREATE blockchain
        CAST vote for "Alice" FROM voter "Voter1"
        CAST vote for "Bob" FROM voter "Voter2"
        PRINT the blockchain
        PRINT the result of validateVotes()
    END METHOD
END CLASS

```

Source Code :

Vote.java

```

import java.security.MessageDigest;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

// Class representing a vote
class Vote {
    public String voterId;
    public String candidate;
    public long timestamp;
    public String hash;

    // Constructor
    public Vote(String voterId, String candidate, long timestamp) {
        this.voterId = voterId;
        this.candidate = candidate;
        this.timestamp = timestamp;
        this.hash = calculateHash();
    }

    // Calculate the hash of the vote
    public String calculateHash() {
        String input = voterId + candidate + timestamp;
        return applySHA256(input);
    }

    // Apply SHA-256 hashing algorithm
    public static String applySHA256(String input) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(input.getBytes("UTF-
8"));

            StringBuilder hexString = new StringBuilder();
            for (byte hashByte : hashBytes) {
                String hex = Integer.toHexString(0xff & hashByte);

```

```

        if (hex.length() == 1) hexString.append('0');
        hexString.append(hex);
    }
    return hexString.toString();
} catch (Exception e) {
    throw new RuntimeException(e);
}
}
}

```

Block.java

```

// Class representing a block in the blockchain
class Block {
    public String previousHash;
    public Vote vote;
    public String hash;
    public long timestamp;

    // Block constructor
    public Block(String previousHash, Vote vote, long timestamp) {
        this.previousHash = previousHash;
        this.vote = vote;
        this.timestamp = timestamp;
        this.hash = calculateHash();
    }

    // Calculate the hash of the block
    public String calculateHash() {
        String input = previousHash + vote.hash + timestamp;
        return Vote.applySHA256(input);
    }
}

```

Blockchain.java

```

// Class representing the blockchain
class Blockchain {
    public List<Block> chain;

    // Blockchain constructor
    public Blockchain() {
        chain = new ArrayList<>();
        chain.add(createGenesisBlock());
    }

    // Create the first block (Genesis Block)
    private Block createGenesisBlock() {
        return new Block("0", new Vote("Genesis", "None",
currentTimestamp()), currentTimestamp());
    }

    // Get the latest block in the blockchain
    public Block getLatestBlock() {

```

```

        return chain.get(chain.size() - 1);
    }

    // Add a new vote to the blockchain
    public void addVote(Vote vote) {
        Block latestBlock = getLatestBlock();
        Block newBlock = new Block(latestBlock.hash, vote,
currentTimestamp());
        chain.add(newBlock);
    }

    // Validate the blockchain by checking hashes
    public boolean validateVotes() {
        for (int i = 1; i < chain.size(); i++) {
            Block currentBlock = chain.get(i);
            Block previousBlock = chain.get(i - 1);

            // Validate current block's hash
            if
(!currentBlock.hash.equals(currentBlock.calculateHash())) {
                System.out.println("Block " + i + " has been tampered
with.");
                return false;
            }

            // Validate previous block's hash link
            if
(!currentBlock.previousHash.equals(previousBlock.hash)) {
                System.out.println("Block " + i + "'s previous hash
doesn't match.");
                return false;
            }
        }
        return true;
    }

    // Get the current system time in milliseconds
    public long currentTimestamp() {
        return new Date().getTime();
    }
}

```

VotingSystem.java

```

public class VotingSystem {
    public static void main(String[] args) {
        // Create a blockchain
        Blockchain blockchain = new Blockchain();

        // Cast votes
        Vote vote1 = new Vote("Voter1", "Alice",
blockchain.currentTimestamp());
        blockchain.addVote(vote1);

        Vote vote2 = new Vote("Voter2", "Bob",
blockchain.currentTimestamp());
    }
}

```

```

        blockchain.addVote(vote2);

        // Validate the blockchain
        System.out.println("Blockchain is valid: " +
blockchain.validateVotes());

        // Print the blockchain
        for (Block block : blockchain.chain) {
            System.out.println("Block [Previous Hash: " +
block.previousHash + ", Hash: " + block.hash +
            ", Vote: " + block.vote.voterId + " voted for " +
block.vote.candidate + ", Timestamp: " + block.timestamp + "]);
        }
    }
}

```

Output:

```

<terminated> VotingSystem [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Oct 17, 2024, 10:45:31 AM – 10:45:33 AM) [pid: 15856]
Blockchain is valid: true
Block [Previous Hash: 0, Hash: 9e66f1cb64a67f826da32eeb66a50986e1f6181f33306b64755bbc80790de152, Vote: Genesis voted for None, Timestamp:
1729142132996]
Block [Previous Hash: 9e66f1cb64a67f826da32eeb66a50986e1f6181f33306b64755bbc80790de152, Hash:
94113f096cdcd5d02722d24a12a261ebd1f04949e0c2ff0c805731a961e5dcc7, Vote: Voter1 voted for Alice, Timestamp: 1729142132996]
Block [Previous Hash: 94113f096cdcd5d02722d24a12a261ebd1f04949e0c2ff0c805731a961e5dcc7, Hash:
77e5a1a926d67ed61e9bd2b31481f71eafccd18f08e8a5d3dc8b792cb7081ec5, Vote: Voter2 voted for Bob, Timestamp: 1729142132997]

```

Result:

To implement a decentralized voting system using blockchain technology has been successfully implemented.

Pre-Test Questions:

1. What is a smart contract, and how can it be used in a blockchain voting system?
2. What are the challenges in implementing a blockchain-based voting system?

Post-Test Questions:

1. What are the key components of a blockchain-based voting system smart contract?
2. How do you handle voter registration and eligibility in a blockchain voting system?