External Practical

Slip No. 1

Demonstrate the creation of a singly linked list for a task scheduler. Insert tasks at the end and display the list.

```
// Define a Node class to represent each task in the linked list
class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
    }
}

// Define a LinkedList class for the task scheduler
class TaskScheduler {
    constructor() {
        this.head = null;
    }

    // Function to insert a task at the end of the linked list
    insertTask(task) {
        const newNode = new Node(task);

        if (!this.head) {
            this.head = newNode;
        } else {
            let current = this.head;
            while (current.next) {
                current = current.next;
            }
            current.next = newNode;
        }

        // Display the updated linked list
        this.displayTasks();
    }

    // Function to display the tasks in the linked list
    displayTasks() {
        let current = this.head;
        const tasks = [];

        while (current) {
            tasks.push(current.data);
            current = current.next;
        }

        console.log("Task List: " + tasks.join(" -> "));
    }
}

// Create an instance of the TaskScheduler
const taskScheduler = new TaskScheduler();

// Insert tasks into the scheduler
taskScheduler.insertTask("Task 1");
taskScheduler.insertTask("Task 2");
taskScheduler.insertTask("Task 3");

// Output:
// Task List: Task 1
// Task List: Task 1 -> Task 2
// Task List: Task 1 -> Task 2 -> Task 3
```

Slip No. 2

Create the doubly linked list and print the data in reverse order.

```
// Define a Node class to represent each task in the doubly linked list
class Node {
    constructor(data) {
        this.data = data;
        this.prev = null;
```

External Practical

```
// Define a DoublyLinkedList class
class DoublyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
    }

    // Function to insert a node at the end of the doubly linked list
    insertNode(data) {
        const newNode = new Node(data);

        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            newNode.prev = this.tail;
            this.tail.next = newNode;
            this.tail = newNode;
        }
    }

    // Function to display the data in reverse order
    printReverse() {
        let current = this.tail;
        const reverseData = [];

        while (current) {
            reverseData.push(current.data);
            current = current.prev;
        }

        console.log("Data in Reverse Order: " + reverseData.join(" <- "));
    }
}

// Create an instance of the DoublyLinkedList
const doublyLinkedList = new DoublyLinkedList();

// Insert nodes into the doubly linked list
doublyLinkedList.insertNode("Data 1");
doublyLinkedList.insertNode("Data 2");
doublyLinkedList.insertNode("Data 3");

// Print data in reverse order
doublyLinkedList.printReverse();
```

Slip No. 3

Implement a stack using an array to manage function calls in a recursive program. Perform push and pop operations.

```
class FunctionCallStack {
    constructor() {
        this.stack = [];
    }

    // Push a function call onto the stack
    push(call) {
        this.stack.push(call);
    }

    // Pop a function call from the stack
    pop() {
        if (!this.isEmpty()) {
            return this.stack.pop();
        } else {
            console.log("Stack is empty. Cannot pop.");
        }
    }

    // Check if the stack is empty
    isEmpty() {
        return this.stack.length === 0;
    }
```

External Practical

```
    }
}

// Example of using the FunctionCallStack to manage function calls in a recursive program

// Create an instance of the FunctionCallStack
const callStack = new FunctionCallStack();

// Function to simulate a recursive program
function recursiveFunction(n) {
    if (n <= 0) {
        console.log("Base case reached.");
    } else {
        console.log("Function call with n =", n);
        callStack.push("Function(" + n + ")");
        recursiveFunction(n - 1);
    }
}

// Perform function calls
recursiveFunction(3);

// Display the contents of the function call stack
callStack.displayStack();

// Pop one function call from the stack
callStack.pop();

// Display the updated contents of the function call stack
callStack.displayStack();
```

Slip No. 4

Use a stack to reverse a given sentence. Display the reversed sentence.

```
function reverseSentence(sentence) {
    const words = sentence.split(" ");
    const wordStack = [];

    // Push each word onto the stack
    words.forEach(word => {
        wordStack.push(word);
    });

    // Pop words from the stack to form the reversed sentence
    const reversedSentence = [];
    while (!isEmpty(wordStack)) {
        reversedSentence.push(wordStack.pop());
    }
    return reversedSentence.join(" ");
}

// Helper function to check if a stack is empty
function isEmpty(stack) {
    return stack.length === 0;
}

// Example usage
const originalSentence = "This is a sample sentence.";
const reversedSentence = reverseSentence(originalSentence);

console.log("Original Sentence: ", originalSentence);
console.log("Reversed Sentence: ", reversedSentence);
```

Slip No. 5

Check for balanced parentheses in a mathematical expression using a stack.

```
function areParenthesesBalanced(expression) {
    const stack = [];

    // Define a mapping for opening and closing parentheses
    const parenthesesMap = {
```

External Practical

```
    };

    // Iterate through each character in the expression
    for (let char of expression) {
        // If the character is an opening parenthesis, push it onto the stack
        if (parenthesesMap[char]) {
            stack.push(char);
        } else if (Object.values(parenthesesMap).includes(char)) {
            // If the character is a closing parenthesis
            // Check if the stack is not empty and the top of the stack matches the corresponding opening parenthesis
            if (stack.length === 0 || parenthesesMap[stack.pop()] !== char) {
                return false; // Unbalanced parentheses
            }
        }
    }

    // The expression is balanced if the stack is empty at the end
    return stack.length === 0;
}

// Example usage
const balancedExpression = "{(a + b) * [c - d]}";
const unbalancedExpression = "[(a + b) * (c - d)";
console.log("Balanced Expression:", areParenthesesBalanced(balancedExpression)); // true
console.log("Unbalanced Expression:", areParenthesesBalanced(unbalancedExpression)); // false
```

Slip No. 6

Demonstrate the traversal of a binary search tree in a preorder function.

```
// Define a Node class for the binary search tree
class Node {
    constructor(value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

// Define a BinarySearchTree class
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    // Function to insert a value into the BST
    insert(value) {
        const newNode = new Node(value);
        if (!this.root) {
            this.root = newNode;
        } else {
            this.insertNode(this.root, newNode);
        }
    }

    // Helper function to recursively insert a node into the BST
    insertNode(node, newNode) {
        if (newNode.value < node.value) {
            if (!node.left) {
                node.left = newNode;
            } else {
                this.insertNode(node.left, newNode);
            }
        } else {
            if (!node.right) {
                node.right = newNode;
            } else {
                this.insertNode(node.right, newNode);
            }
        }
    }

    // Function to perform pre-order traversal of the BST
```

External Practical

```
            this.preOrderTraversal(node.left, result);   // Traverse the left subtree
            this.preOrderTraversal(node.right, result);   // Traverse the right subtree
        }
        return result;
    }
}

// Create an instance of the BinarySearchTree
const bst = new BinarySearchTree();

// Insert values into the BST
bst.insert(10);
bst.insert(5);
bst.insert(15);
bst.insert(3);
bst.insert(7);
bst.insert(12);
bst.insert(20);

// Perform pre-order traversal and display the result
const preOrderResult = bst.preOrderTraversal(bst.root);
console.log("Pre-order Traversal:", preOrderResult);
```

Slip No. 7

Reverse the elements of a stack using a linked list. Explain each step of the process.

```
// Create the Original Stack using a Linked List:
// We'll first create a simple stack using a linked list. The linked list nodes will contain the data and a pointer to the next node.

class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
    }
}

class Stack {
    constructor() {
        this.top = null;
    }

    push(data) {
        const newNode = new Node(data);
        newNode.next = this.top;
        this.top = newNode;
    }

    pop() {
        if (this.top === null) {
            return null;
        }

        const data = this.top.data;
        this.top = this.top.next;
        return data;
    }

    display() {
        let current = this.top;
        while (current !== null) {
            console.log(current.data);
            current = current.next;
        }
    }
}

const originalStack = new Stack();
originalStack.push(1);
originalStack.push(2);
originalStack.push(3);
originalStack.push(4);

// Create an Auxiliary Stack:
```

External Practical

```
// Reverse the Original Stack:
// We'll pop elements from the original stack and push them onto the auxiliary stack.

while (originalStack.top !== null) {
    const element = originalStack.pop();
    auxiliaryStack.push(element);
}

// Display the Reversed Stack (Optional):
// You can display the reversed stack if needed.

auxiliaryStack.display();

// Copy Elements Back to the Original Stack:
// Now, we'll pop elements from the auxiliary stack and push them back onto the original stack.

while (auxiliaryStack.top !== null) {
    const element = auxiliaryStack.pop();
    originalStack.push(element);
}

// Display the Final Result:
// You can display the final result, which is the original stack with reversed elements.

originalStack.display();
```

Slip No. 8

Create a linear queue to simulate a printer queue for processing print jobs. Perform enqueue and dequeue operations.

```
class PrinterQueue {
    constructor() {
        this.queue = [];
    }

    // Enqueue operation to add a print job to the end of the queue
    enqueue(printJob) {
        this.queue.push(printJob);
        console.log(`Print job "${printJob}" has been added to the queue.`);
        this.displayQueue();
    }

    // Dequeue operation to process and remove the first print job from the queue
    dequeue() {
        if (this.isEmpty()) {
            console.log("The queue is empty. No print jobs to dequeue.");
            return null;
        }
        const processedJob = this.queue.shift();
        console.log(`Print job "${processedJob}" has been processed and removed from the queue.`);
        this.displayQueue();
        return processedJob;
    }

    // Function to check if the queue is empty
    isEmpty() {
        return this.queue.length === 0;
    }

    // Function to display the current state of the queue
    displayQueue() {
        if (this.isEmpty()) {
            console.log("Queue is empty.");
        } else {
            console.log("Current Queue: " + this.queue.join(" -> "));
        }
    }
}

// Create an instance of the PrinterQueue
const printerQueue = new PrinterQueue();

// Enqueue print jobs
```

External Practical

```
// Dequeue print jobs
printerQueue.dequeue();
printerQueue.dequeue();
printerQueue.dequeue();
printerQueue.dequeue(); // Attempting to dequeue from an empty queue
```

Slip No. 9

Demonstrate the traversal of a binary search tree in a postorder function.

```
// Define a Node class for the binary search tree
class Node {
    constructor(value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

// Define a BinarySearchTree class
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    // Function to insert a value into the BST
    insert(value) {
        const newNode = new Node(value);

        if (!this.root) {
            this.root = newNode;
        } else {
            this.insertNode(this.root, newNode);
        }
    }

    // Helper function to recursively insert a node into the BST
    insertNode(node, newNode) {
        if (newNode.value < node.value) {
            if (!node.left) {
                node.left = newNode;
            } else {
                this.insertNode(node.left, newNode);
            }
        } else {
            if (!node.right) {
                node.right = newNode;
            } else {
                this.insertNode(node.right, newNode);
            }
        }
    }

    // Function to perform post-order traversal of the BST
    postOrderTraversal(node, result = []) {
        if (node) {
            this.postOrderTraversal(node.left, result);    // Traverse the left subtree
            this.postOrderTraversal(node.right, result);   // Traverse the right subtree
            result.push(node.value);                       // Visit the root
        }
        return result;
    }
}

// Create an instance of the BinarySearchTree
const bst = new BinarySearchTree();

// Insert values into the BST
bst.insert(10);
bst.insert(5);
bst.insert(15);
bst.insert(3);
bst.insert(7);
bst.insert(12);
```

External Practical

```
const postOrderResult = bst.postOrderTraversal(bst.root);
console.log("Post-order Traversal:", postOrderResult);
```

Slip No. 10

Design a priority queue for processing tasks with different levels of priority.

```
class PriorityQueue {
    constructor() {
        this.items = [];
    }

    // Function to enqueue a task with a specified priority
    enqueue(task, priority) {
        const element = { task, priority };
        let added = false;

        // Iterate through the queue to find the correct position based on priority
        for (let i = 0; i < this.items.length; i++) {
            if (element.priority > this.items[i].priority) {
                this.items.splice(i, 0, element);
                added = true;
                break;
            }
        }

        // If the element has the lowest priority, add it to the end
        if (!added) {
            this.items.push(element);
        }

        console.log(`Task "${task}" with priority ${priority} has been added to the queue.`);
        this.displayQueue();
    }

    // Function to dequeue the highest priority task
    dequeue() {
        if (this.isEmpty()) {
            console.log("The queue is empty. No tasks to dequeue.");
            return null;
        }

        const task = this.items.shift().task;
        console.log(`Task "${task}" has been dequeued.`);
        this.displayQueue();
        return task;
    }

    // Function to check if the queue is empty
    isEmpty() {
        return this.items.length === 0;
    }

    // Function to display the current state of the queue
    displayQueue() {
        if (this.isEmpty()) {
            console.log("Queue is empty.");
        } else {
            console.log("Current Queue:");
            this.items.forEach(item => console.log(`  Task: ${item.task}, Priority: ${item.priority}`));
        }
    }
}

// Create an instance of the PriorityQueue
const priorityQueue = new PriorityQueue();

// Enqueue tasks with different priorities
priorityQueue.enqueue("Task1", 3);
priorityQueue.enqueue("Task2", 1);
priorityQueue.enqueue("Task3", 2);

// Dequeue tasks based on priority
priorityQueue.dequeue();
priorityQueue.dequeue();
```

External Practical

Slip No. 11

Implement a binary search tree to store employee records. Perform insertion, deletion, and search operations.

```
class Employee {
    constructor(id, name, position) {
        this.id = id;
        this.name = name;
        this.position = position;
    }
}

class Node {
    constructor(employee) {
        this.employee = employee;
        this.left = null;
        this.right = null;
    }
}

class EmployeeBST {
constructor() {
    this.root = null;
}

// Function to insert an employee record into the BST
insert(employee) {
    const newNode = new Node(employee);

    if (!this.root) {
        this.root = newNode;
    } else {
        this.insertNode(this.root, newNode);
    }

    console.log(`Employee ${employee.name} with ID ${employee.id} has been added to the BST.`);
    this.displayBST();
}

// Helper function to recursively insert a node into the BST
insertNode(node, newNode) {
    if (newNode.employee.id < node.employee.id) {
        if (!node.left) {
            node.left = newNode;
        } else {
            this.insertNode(node.left, newNode);
        }
    } else {
        if (!node.right) {
            node.right = newNode;
        } else {
            this.insertNode(node.right, newNode);
        }
    }
}

// Function to search for an employee by ID
search(id) {
    return this.searchNode(this.root, id);
}

// Helper function to recursively search for a node in the BST
searchNode(node, id) {
    if (!node || node.employee.id === id) {
        return node ? node.employee : null;
    }

    if (id < node.employee.id) {
        return this.searchNode(node.left, id);
    } else {
        return this.searchNode(node.right, id);
    }
}

// Function to delete an employee record by ID
```

External Practical

```
        this.displayBST();
    }

    // Helper function to recursively delete a node from the BST
    deleteNode(node, id) {
        if (!node) {
            return null;
        }

        if (id < node.employee.id) {
            node.left = this.deleteNode(node.left, id);
        } else if (id > node.employee.id) {
            node.right = this.deleteNode(node.right, id);
        } else {
            // Node with only one child or no child
            if (!node.left) {
                return node.right;
            } else if (!node.right) {
                return node.left;
            }

            // Node with two children, get the in-order successor (smallest in the right subtree)
            node.employee = this.findMin(node.right);
            // Delete the in-order successor
            node.right = this.deleteNode(node.right, node.employee.id);
        }

        return node;
    }

    // Helper function to find the smallest node (in-order successor) in the BST
    findMin(node) {
        while (node.left) {
            node = node.left;
        }
        return node.employee;
    }

    // Function to display the current state of the BST (in-order traversal)
    displayBST() {
        console.log("Current BST (In-order Traversal):");
        this.inOrderTraversal(this.root);
        console.log("-----------------------");
    }

    // Helper function for in-order traversal (used for display)
    inOrderTraversal(node) {
        if (node) {
            this.inOrderTraversal(node.left);
            console.log(`ID: ${node.employee.id}, Name: ${node.employee.name}, Position:
${node.employee.position}`);
            this.inOrderTraversal(node.right);
        }
    }
}

// Create an instance of the EmployeeBST
const employeeBST = new EmployeeBST();

// Insert employee records
employeeBST.insert(new Employee(101, "John Doe", "Manager"));
employeeBST.insert(new Employee(204, "Alice Johnson", "Engineer"));
employeeBST.insert(new Employee(153, "Bob Smith", "Analyst"));
employeeBST.insert(new Employee(309, "Eva Anderson", "Designer"));

// Search for an employee
const searchResult = employeeBST.search(204);
console.log("Search Result:", searchResult);

// Delete an employee record
employeeBST.delete(204);
```

Slip No. 12

Demonstrate the traversal of a binary search tree in an in-order manner.

External Practical

```
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

// Define a BinarySearchTree class
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    // Function to insert a value into the BST
    insert(value) {
        const newNode = new Node(value);

        if (!this.root) {
            this.root = newNode;
        } else {
            this.insertNode(this.root, newNode);
        }
    }

    // Helper function to recursively insert a node into the BST
    insertNode(node, newNode) {
        if (newNode.value < node.value) {
            if (!node.left) {
                node.left = newNode;
            } else {
                this.insertNode(node.left, newNode);
            }
        } else {
            if (!node.right) {
                node.right = newNode;
            } else {
                this.insertNode(node.right, newNode);
            }
        }
    }

    // Function to perform in-order traversal of the BST
    inOrderTraversal(node, result = []) {
        if (node) {
            this.inOrderTraversal(node.left, result);    // Traverse the left subtree
            result.push(node.value);                     // Visit the root
            this.inOrderTraversal(node.right, result);   // Traverse the right subtree
        }
        return result;
    }
}

// Create an instance of the BinarySearchTree
const bst = new BinarySearchTree();

// Insert values into the BST
bst.insert(10);
bst.insert(5);
bst.insert(15);
bst.insert(3);
bst.insert(7);
bst.insert(12);
bst.insert(20);

// Perform in-order traversal and display the result
const inOrderResult = bst.inOrderTraversal(bst.root);
console.log("In-order Traversal:", inOrderResult);
```

Slip No. 13

Implement an undirected graph to represent a network of cities. Perform depth-first traversal starting from a specific city.

```
class Graph {
    constructor() {
```

External Practical

```
// Function to add a city to the graph
addCity(city) {
    this.vertices.set(city, []);
}

// Function to add an undirected edge between two cities
addEdge(city1, city2) {
    this.vertices.get(city1).push(city2);
    this.vertices.get(city2).push(city1);
}

// Function to perform depth-first traversal starting from a specific city
depthFirstTraversal(startCity) {
    const visited = new Set();
    this._depthFirstTraversal(startCity, visited);
}

// Helper function for recursive depth-first traversal
_depthFirstTraversal(city, visited) {
    if (!visited.has(city)) {
        console.log("Visited City:", city);
        visited.add(city);
        const neighbors = this.vertices.get(city);
        for (const neighbor of neighbors) {
            this._depthFirstTraversal(neighbor, visited);
        }
    }
}
}

// Create an instance of the Graph
const cityNetwork = new Graph();

// Add cities to the graph
cityNetwork.addCity("CityA");
cityNetwork.addCity("CityB");
cityNetwork.addCity("CityC");
cityNetwork.addCity("CityD");
cityNetwork.addCity("CityE");

// Add connections between cities
cityNetwork.addEdge("CityA", "CityB");
cityNetwork.addEdge("CityA", "CityC");
cityNetwork.addEdge("CityB", "CityD");
cityNetwork.addEdge("CityC", "CityD");
cityNetwork.addEdge("CityD", "CityE");

// Perform depth-first traversal starting from a specific city
cityNetwork.depthFirstTraversal("CityA");
```

Slip No. 14

Perform breadth-first traversal on the same graph starting from a different city.

```
class Graph {
    constructor() {
        this.vertices = new Map();
    }

    // Function to add a city to the graph
    addCity(city) {
        this.vertices.set(city, []);
    }

    // Function to add an undirected edge between two cities
    addEdge(city1, city2) {
        this.vertices.get(city1).push(city2);
        this.vertices.get(city2).push(city1);
    }

    // Function to perform breadth-first traversal starting from a specific city
    breadthFirstTraversal(startCity) {
        const visited = new Set();
        const queue = [];
```

External Practical

```
        while (queue.length > 0) {
            const currentCity = queue.shift();
            console.log("Visited City:", currentCity);

            const neighbors = this.vertices.get(currentCity);
            for (const neighbor of neighbors) {
                if (!visited.has(neighbor)) {
                    visited.add(neighbor);
                    queue.push(neighbor);
                }
            }
        }
    }
}

// Create an instance of the Graph
const cityNetwork = new Graph();

// Add cities to the graph
cityNetwork.addCity("CityA");
cityNetwork.addCity("CityB");
cityNetwork.addCity("CityC");
cityNetwork.addCity("CityD");
cityNetwork.addCity("CityE");

// Add connections between cities
cityNetwork.addEdge("CityA", "CityB");
cityNetwork.addEdge("CityA", "CityC");
cityNetwork.addEdge("CityB", "CityD");
cityNetwork.addEdge("CityC", "CityD");
cityNetwork.addEdge("CityD", "CityE");

// Perform breadth-first traversal starting from a different city
cityNetwork.breadthFirstTraversal("CityB");
```

Slip No. 15

Design a hash table to store contact information based on names. Handle collisions appropriately.

```
class Contact {
    constructor(name, phoneNumber, email) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.email = email;
    }
}

class HashTable {
    constructor(size = 10) {
        this.size = size;
        this.table = new Array(size).fill(null).map(() => []);
    }

    // Function to generate a hash code for a given key (name in this case)
    hash(key) {
        let hash = 0;
        for (const char of key) {
            hash += char.charCodeAt(0);
        }
        return hash % this.size;
    }

    // Function to insert a contact into the hash table
    insert(contact) {
        const key = contact.name;
        const index = this.hash(key);

        const bucket = this.table[index];
        const existingContact = bucket.find(c => c.name === key);

        if (existingContact) {
            // If a contact with the same name already exists, update it
            existingContact.phoneNumber = contact.phoneNumber;
            existingContact.email = contact.email;
        } else {
```

External Practical

```
}

// Function to retrieve contact information based on a name
get(name) {
    const index = this.hash(name);
    const bucket = this.table[index];

    const contact = bucket.find(c => c.name === name);
    return contact || null;
}

// Function to remove a contact based on a name
remove(name) {
    const index = this.hash(name);
    const bucket = this.table[index];

    const contactIndex = bucket.findIndex(c => c.name === name);

    if (contactIndex !== -1) {
        // Remove the contact from the linked list
        bucket.splice(contactIndex, 1);
        console.log(`Contact ${name} has been removed.`);
    } else {
        console.log(`Contact ${name} not found.`);
    }
}

// Function to display the current state of the hash table
displayTable() {
    console.log("Current Hash Table:");
    for (let i = 0; i < this.size; i++) {
        console.log(`Bucket ${i}:`, this.table[i]);
    }
    console.log("-----------------------");
}
}

// Create an instance of the HashTable
const contactsTable = new HashTable();

// Insert contacts
contactsTable.insert(new Contact("John Doe", "123-456-7890", "john@example.com"));
contactsTable.insert(new Contact("Alice Smith", "987-654-3210", "alice@example.com"));
contactsTable.insert(new Contact("Bob Johnson", "555-555-5555", "bob@example.com"));

// Display the current state of the hash table
contactsTable.displayTable();

// Get contact information
const contactJohn = contactsTable.get("John Doe");
console.log("Contact Information for John Doe:", contactJohn);

// Remove a contact
contactsTable.remove("Alice Smith");

// Display the updated state of the hash table
contactsTable.displayTable();
```

Slip No. 16

Use the brute force technique to find the largest element in an array.

```
function findLargestElement(arr) {
    if (arr.length === 0) {
        return null; // Handle the case when the array is empty
    }

    let largest = arr[0]; // Assume the first element is the largest

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] > largest) {
            largest = arr[i]; // Update the largest element if a larger element is found
        }
    }

    return largest;
```

External Practical

```
const array = [5, 2, 9, 1, 7, 3, 6];
const largestElement = findLargestElement(array);

console.log("Array:", array);
console.log("Largest Element:", largestElement);
```

Slip No. 17

Apply brute force to search for a pattern in a text document.

```
function bruteForcePatternSearch(text, pattern) {
    const textLength = text.length;
    const patternLength = pattern.length;
    const indices = [];

    for (let i = 0; i <= textLength - patternLength; i++) {
        let j;

        // Check if the pattern matches the substring starting at index i
        for (j = 0; j < patternLength; j++) {
            if (text[i + j] !== pattern[j]) {
                break; // Break the loop if a mismatch is found
            }
        }

        // If the inner loop completed without a break, the pattern is found at index i
        if (j === patternLength) {
            indices.push(i);
        }
    }

    return indices;
}

// Example usage
const textDocument = "ABABCABABABAABC";
const searchPattern = "ABA";
const patternIndices = bruteForcePatternSearch(textDocument, searchPattern);

console.log("Text Document:", textDocument);
console.log("Search Pattern:", searchPattern);
console.log("Pattern Indices:", patternIndices);
```

Slip No. 18

Implement Prim’s algorithm to find the minimum spanning tree of a connected graph representing a network of cities.

```
class Graph {
    constructor() {
        this.vertices = new Map();
    }

    addVertex(vertex) {
        this.vertices.set(vertex, []);
    }

    addEdge(vertex1, vertex2, weight) {
        this.vertices.get(vertex1).push({ vertex: vertex2, weight });
        this.vertices.get(vertex2).push({ vertex: vertex1, weight });
    }

    primMST(startVertex) {
        const mst = new Set(); // Minimum Spanning Tree
        const visited = new Set();
        const priorityQueue = new PriorityQueue();

        // Add the start vertex to the minimum spanning tree
        mst.add(startVertex);

        // Add the edges of the start vertex to the priority queue
        this.vertices.get(startVertex).forEach(edge => {
            priorityQueue.enqueue({ vertex: edge.vertex, weight: edge.weight });
        });
```

External Practical

```
                const { vertex, weight } = priorityQueue.dequeue();

                if (!visited.has(vertex)) {
                    visited.add(vertex);
                    mst.add(vertex);

                    this.vertices.get(vertex).forEach(edge => {
                        if (!visited.has(edge.vertex)) {
                            priorityQueue.enqueue({ vertex: edge.vertex, weight: edge.weight });
                        }
                    });
                }
            }

            return mst;
        }
    }

    class PriorityQueue {
        constructor() {
            this.items = [];
        }

        enqueue(item) {
            this.items.push(item);
            this.items.sort((a, b) => a.weight - b.weight);
        }

        dequeue() {
            if (this.isEmpty()) {
                return null;
            }
            return this.items.shift();
        }

        isEmpty() {
            return this.items.length === 0;
        }
    }

    // Example usage:

    const cityNetwork = new Graph();

    // Add cities to the graph
    cityNetwork.addVertex("CityA");
    cityNetwork.addVertex("CityB");
    cityNetwork.addVertex("CityC");
    cityNetwork.addVertex("CityD");

    // Add connections between cities with weights
    cityNetwork.addEdge("CityA", "CityB", 4);
    cityNetwork.addEdge("CityA", "CityC", 2);
    cityNetwork.addEdge("CityB", "CityC", 5);
    cityNetwork.addEdge("CityB", "CityD", 10);
    cityNetwork.addEdge("CityC", "CityD", 3);

    // Find the minimum spanning tree using Prim's algorithm starting from "CityA"
    const minimumSpanningTree = cityNetwork.primMST("CityA");

    console.log("Minimum Spanning Tree:", Array.from(minimumSpanningTree));
```

Slip No. 19

Apply Kruskal’s algorithm to the same graph and compare the results.

```
    class Graph {
        constructor() {
            this.vertices = new Map();
            this.edges = [];
        }

        addVertex(vertex) {
            this.vertices.set(vertex, vertex);
        }
```

External Practical

```
    find(vertex) {
        if (this.vertices.get(vertex) === vertex) {
            return vertex;
        }

        return this.find(this.vertices.get(vertex));
    }

    union(vertex1, vertex2) {
        const root1 = this.find(vertex1);
        const root2 = this.find(vertex2);

        this.vertices.set(root1, root2);
    }

    kruskalMST() {
        const mst = new Set();
        this.edges.sort((a, b) => a.weight - b.weight);

        this.edges.forEach(edge => {
            const root1 = this.find(edge.vertex1);
            const root2 = this.find(edge.vertex2);
            if (root1 !== root2) {
                mst.add(edge.vertex1);
                mst.add(edge.vertex2);
                this.union(edge.vertex1, edge.vertex2);
            }
        });

        return mst;
    }
}

// Example usage:

const cityNetwork = new Graph();

// Add cities to the graph
cityNetwork.addVertex("CityA");
cityNetwork.addVertex("CityB");
cityNetwork.addVertex("CityC");
cityNetwork.addVertex("CityD");

// Add connections between cities with weights
cityNetwork.addEdge("CityA", "CityB", 4);
cityNetwork.addEdge("CityA", "CityC", 2);
cityNetwork.addEdge("CityB", "CityC", 5);
cityNetwork.addEdge("CityB", "CityD", 10);
cityNetwork.addEdge("CityC", "CityD", 3);

// Find the minimum spanning tree using Kruskal's algorithm
const minimumSpanningTreeKruskal = cityNetwork.kruskalMST();

console.log("Minimum Spanning Tree (Kruskal):", Array.from(minimumSpanningTreeKruskal));
```

Slip No. 20

Implement the Tower of Hanoi problem using the divide and conquer technique. Explain each step of the process.

```
function towerOfHanoi(n, sourceRod, auxiliaryRod, targetRod) {
    if (n === 1) {
        // Base case: Move the top disk from sourceRod to targetRod
        console.log(`Move disk 1 from ${sourceRod} to ${targetRod}`);
        return;
    }

    // Move (n-1) disks from sourceRod to auxiliaryRod using targetRod as auxiliary
    towerOfHanoi(n - 1, sourceRod, targetRod, auxiliaryRod);

    // Move the nth disk from sourceRod to targetRod
    console.log(`Move disk ${n} from ${sourceRod} to ${targetRod}`);
```

External Practical

```
// Example usage for 3 disks
towerOfHanoi(3, 'A', 'B', 'C');

// Explanation of the process:

// Base Case (n = 1): If there is only one disk to move, we directly move it from the source rod to the target rod.

// Recursive Step (n > 1): To move a stack of n disks from the source rod to the target rod, we break it down into three steps:

// Move (n-1) disks from the source rod to the auxiliary rod using the target rod as an auxiliary.
// Move the nth disk from the source rod to the target rod.
// Move (n-1) disks from the auxiliary rod to the target rod using the source rod as an auxiliary.
```