**JAVA**

**Day-4 Assignment**

**Task 1: Array Sorting and Searching**

1. **Implement a function called BruteForceSort that sorts an array using the brute force approach. Use this function to sort an array created with InitializeArray.**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class BruteForceSort {

private static boolean isSorted(int[] array) {

for (int i = 1; i < array.length; i++) {

if (array[i - 1] > array[i]) {

return false;

}

}

return true;

}

private static void generatePermutations(int[] array, int n, List<int[]> permutations) {

if (n == 1) {

permutations.add(array.clone());

} else {

for (int i = 0; i < n - 1; i++) {

generatePermutations(array, n - 1, permutations);

if (n % 2 == 0) {

swap(array, i, n - 1);

} else {

swap(array, 0, n - 1);

}

}

generatePermutations(array, n - 1, permutations);

}

}

private static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

public static void bruteForceSort(int[] array) {

List<int[]> permutations = new ArrayList<>();

generatePermutations(array, array.length, permutations);

for (int[] permutation : permutations) {

if (isSorted(permutation)) {

System.arraycopy(permutation, 0, array, 0, array.length);

return;

}

}

}

public static void main(String[] args) {

int[] array = ArrayUtils.initializeArray(5); // Initialize array with random integers

System.out.println("Original array: " + Arrays.toString(array));

bruteForceSort(array);

System.out.println("Sorted array: " + Arrays.toString(array));

}

}

import java.util.Random;

public class ArrayUtils {

public static int[] initializeArray(int size) {

Random rand = new Random();

int[] array = new int[size];

for (int i = 0; i < size; i++) {

array[i] = rand.nextInt(100); // Random integers between 0 and 99

}

return array;

}

}

**b) Write a function named PerformLinearSearch that searches for a specific element in an array and returns the index of the element if found or -1 if not found.**

Answer:

package com.wipro.fundamentals;

public class SumInArray {

public static void main(String[] args) {

// TODO Auto-generated method stub

int [] a= {1,2,1};

System.out.println(sum(a,0));

}

public static int sum(int []a, int i) {

if(a.length==0) return 0;

return a[i] + sum(a,i++);

}

}

**Task 2: Two-Sum Problem**

**a.Given an array of integers, write a program that finds if there are two numbers that add up to a specific target. You may assume that each input would have exactly one solution, and you may not use the same element twice. Optimize the solution for time complexity.**

import java.util.HashMap;

import java.util.Map;

public class TwoSum {

public static int[] findTwoSum(int[] nums, int target) {

Map<Integer, Integer> numMap = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int complement = target - nums[i];

if (numMap.containsKey(complement)) {

return new int[] { numMap.get(complement), i };

}

numMap.put(nums[i], i);

}

return new int[] {};

}

public static void main(String[] args) {

int[] nums = { 2, 7, 11, 15 };

int target = 9;

int[] result = findTwoSum(nums, target);

if (result.length == 2) {

System.out.println("Indices of the two numbers are: " + result[0] + " and " + result[1]);

} else {

System.out.println("No solution found.");

}

}

}

Task 3: Understanding Functions through Arrays

1. **Write a recursive function named SumArray that calculates and returns the sum of elements in an array, demonstarte with example.**

public class SumArray {

public static int sumArray(int[] array, int index) {

if (index < 0) {

return 0;

}

return array[index] + sumArray(array, index - 1);

}

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

int sum = sumArray(array, array.length - 1);

System.out.println("The sum of the array elements is: " + sum);

}}

**Task 4: Advanced Array Operations**

**a.Implement a method SliceArray that takes an array, a starting index, and an end index, then returns a new array containing the elements from the start to the end index.**

public class ArrayUtils {

public static int[] sliceArray(int[] array, int startIndex, int endIndex) {

if (startIndex < 0 || endIndex >= array.length || startIndex > endIndex) {

throw new IllegalArgumentException("Invalid start or end index.");

}

int length = endIndex - startIndex + 1;

int[] slicedArray = new int[length];

for (int i = 0; i < length; i++) {

slicedArray[i] = array[startIndex + i];

}

return slicedArray;

}

public static void main(String[] args) {

int[] array = {10, 20, 30, 40, 50, 60, 70};

int startIndex = 2;

int endIndex = 5;

int[] result = sliceArray(array, startIndex, endIndex);

System.out.println("Sliced array: " + java.util.Arrays.toString(result));

}

}

**b.Create a recursive function to find the nth element of a Fibonacci sequence and store the first n elements in an array.**

public class FibonacciSequence {

public static int fibonacci(int n) {

if (n <= 0) {

throw new IllegalArgumentException("The input must be a positive integer.");

}

if (n == 1) {

return 0;

} else if (n == 2) {

return 1;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

public static int[] generateFibonacciArray(int n) {

if (n <= 0) {

throw new IllegalArgumentException("The input must be a positive integer.");

}

int[] fibonacciArray = new int[n];

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

fibonacciArray[i - 1] = fibonacci(i);

}

return fibonacciArray;

}

public static void main(String[] args) {

int n = 10; // Example: Generate the first 10 Fibonacci numbers

int[] fibonacciArray = generateFibonacciArray(n);

System.out.println("The first " + n + " Fibonacci numbers are:");

for (int num : fibonacciArray) {

System.out.print(num + " ");

}

}

}

**Day 5:**

**Task 1: Implementing a Linked List**

1. **Write a class CustomLinkedList that implements a singly linked list with methods for InsertAtBeginning, InsertAtEnd, InsertAtPosition, DeleteNode, UpdateNode, and DisplayAllNodes. Test the class by performing a series of insertions, updates, and deletions.**

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

class CustomLinkedList {

Node head;

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

public void insertAtPosition(int data, int position) {

Node newNode = new Node(data);

if (position == 0) {

insertAtBeginning(data);

return;

}

Node temp = head;

for (int i = 0; i < position - 1; i++) {

if (temp != null) {

temp = temp.next;

} else {

throw new IllegalArgumentException("Position out of bounds");

}

}

newNode.next = temp.next;

temp.next = newNode;

}

public void deleteNode(int data) {

if (head == null) {

return;

}

if (head.data == data) {

head = head.next;

return;

}

Node temp = head;

while (temp.next != null && temp.next.data != data) {

temp = temp.next;

}

if (temp.next == null) {

throw new IllegalArgumentException("Node not found");

}

temp.next = temp.next.next;

} public void updateNode(int oldData, int newData) {

Node temp = head;

while (temp != null && temp.data != oldData) {

temp = temp.next;

}

if (temp == null) {

throw new IllegalArgumentException("Node not found");

}

temp.data = newData;

}

public void displayAllNodes() {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " -> ");

temp = temp.next;

}

System.out.println("null");

}

public static void main(String[] args) {

CustomLinkedList list = new CustomLinkedList();

list.insertAtEnd(1);

list.insertAtEnd(2);

list.insertAtEnd(3);

list.displayAllNodes();

list.insertAtBeginning(0);

list.displayAllNodes();

list.insertAtPosition(1, 2);

list.displayAllNodes();

list.updateNode(1, 9);

list.displayAllNodes();

list.deleteNode(9);

list.displayAllNodes();

}

}

**Task 2: Stack and Queue Operations**

**1.Create a CustomStack class with operations Push, Pop, Peek, and IsEmpty. Demonstrate its LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.**

package com.stack;

import java.util.Stack;

public class StackMethods {

public static void main(String[] args) {

Stack<Integer> stack= new Stack<Integer>();

stack.push(10);

stack.push(20);

stack.push(2);

Integer e1 = stack.pop();

System.out.println(e1);//2

System.out.println(stack.peek());//20

System.out.println(stack.isEmpty());//false

Integer e2 = stack.pop();

System.out.println(e2);//20

Integer e3=stack.pop();//10

System.out.println(stack.isEmpty());//true

}

}

**2) Develop a CustomQueue class with methods for Enqueue, Dequeue, Peek, and IsEmpty. Show how your queue can handle different data types by enqueuing strings and integers, then dequeuing and displaying them to confirm FIFO order.**

package com.Queue;

import java.util.LinkedList;

import java.util.List;

public class CustomQueue<T> {

private LinkedList<T> list;

public CustomQueue() {

list = new LinkedList();

}

public void enqueue(T data) {

list.addLast(data);

}

public boolean isEmpty() {

return list.isEmpty();

} public T dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return list.removeFirst();

}

public T peek() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return list.getFirst();

}

public static void main(String[] args) {

// Adding Integer Type

CustomQueue<Integer> q1 = new CustomQueue<>();

q1.enqueue(10);

q1.enqueue(20);

q1.enqueue(15);

System.out.println(q1.peek());//10

System.out.println(q1.dequeue());//10

System.out.println(q1.peek());//20

//Adding String type

CustomQueue<String> s1 = new CustomQueue<>();

s1.enqueue("Pushu");

s1.enqueue("Siree");

s1.enqueue("Shushi");

System.out.println(s1.dequeue());//pushu

System.out.println(s1.peek());//siree

}

}

**Task 3: Priority Queue Scenario**

**Implement a priority queue to manage emergency room admissions in a hospital. Patients with higher urgency should be served before those with lower urgency.**

package com.PriorityQ;

import java.util.PriorityQueue;

public class EmergencyRoom {

private PriorityQueue<Patient> patientQueue;

public EmergencyRoom() {

patientQueue = new PriorityQueue<>();

}

public void admitPatient(String name, int urgency) {

Patient newPatient = new Patient(name, urgency);

patientQueue.add(newPatient);

System.out.println("Admitted: " + newPatient);

}

public void servePatient() {

if (patientQueue.isEmpty()) {

System.out.println("No patients to serve.");

} else {

Patient nextPatient = patientQueue.poll();

System.out.println("Serving: " + nextPatient);

}

}

public void peekNextPatient() {

if (patientQueue.isEmpty()) {

System.out.println("No patients in the queue.");

} else {

Patient nextPatient = patientQueue.peek();

System.out.println("Next patient: " + nextPatient);

}

}

public boolean isEmpty() {

return patientQueue.isEmpty();

}

public static void main(String[] args) {

EmergencyRoom er = new EmergencyRoom();

er.admitPatient("ammu", 5);

er.admitPatient("Janu", 2);

er.admitPatient("Nikki", 10);

er.admitPatient("Jones", 1);

er.peekNextPatient();

er.servePatient();

er.peekNextPatient();

er.servePatient();

er.peekNextPatient();

er.servePatient();

er.servePatient();

er.servePatient();

}}

package com.PriorityQ;

public class Patient implements Comparable<Patient> {

private String pName;

private int urgency;

public Patient(String pName, int urgency) {

super();

this.pName = pName;

this.urgency = urgency;

}

public String getpName() {

return pName;

}

public void setpName(String pName) {

this.pName = pName;

}

public int getUrgency() {

return urgency;

}public void setUrgency(int urgency) {

this.urgency = urgency;

}

@Override

public String toString() {

return "Patient [pName=" + pName + ", urgency=" + urgency + "]";

}

@Override

public int compareTo(Patient p1) {

return Integer.compare(p1.urgency, this.urgency);

}}

**Day 6:**

**Task 1: Real-time Data Stream Sorting**

**A stock trading application requires real-time sorting of trade transactions by price. Implement a heap sort algorithm that can efficiently handle continuous incoming data, adding and sorting new trades as they come.**

class Trade {

private String symbol;

private double price;

public Trade(String symbol, double price) {

this.symbol = symbol;

this.price = price;

}

public String getSymbol() {

return symbol;

}

public double getPrice() {

return price;

}

}

public class RealTimeTradeSorter {

private PriorityQueue<Trade> tradeQueue;

public RealTimeTradeSorter() {

tradeQueue = new PriorityQueue<>(Comparator.comparingDouble(Trade::getPrice));

}

public void addTrade(Trade trade) {

tradeQueue.offer(trade);

}

public Trade getMinTrade() {

return tradeQueue.peek();

}

public Trade removeMinTrade() {

return tradeQueue.poll();

}

public boolean isEmpty() {

return tradeQueue.isEmpty();

}

public int getSize() {

return tradeQueue.size();

}

public static void main(String[] args) {

RealTimeTradeSorter sorter = new RealTimeTradeSorter();

sorter.addTrade(new Trade("AAPL", 150.0));

sorter.addTrade(new Trade("GOOGL", 2800.0));

sorter.addTrade(new Trade("TSLA", 700.0));

sorter.addTrade(new Trade("AMZN", 3300.0));

while (!sorter.isEmpty()) {

Trade minTrade = sorter.removeMinTrade();

System.out.println("Symbol: " + minTrade.getSymbol() + ", Price: " + minTrade.getPrice());

}

}

}

Task 2: Linked List Middle Element Search

You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.

class ListNode {

int value;

ListNode next;

ListNode(int value) {

this.value = value;

this.next = null;

}

}

public class LinkedListMiddle {

public static ListNode findMiddle(ListNode head) {

if (head == null) {

return null;

}

ListNode slowPointer = head;

ListNode fastPointer = head;

while (fastPointer != null && fastPointer.next != null) {

slowPointer = slowPointer.next;

fastPointer = fastPointer.next.next;

}

return slowPointer;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode middle = findMiddle(head);

if (middle != null) {

System.out.println("The middle element is: " + middle.value);

} else {

System.out.println("The list is empty.");

}

}

}

Task 3: Queue Sorting with Limited Space

You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.

public class QueueSorter {

public static void sortQueue(Queue<Integer> queue) {

Stack<Integer> stack = new Stack<>();

while (!queue.isEmpty()) {

int current = queue.remove();

while (!stack.isEmpty() && stack.peek() > current) {

queue.add(stack.pop());

}

stack.push(current);

}

while (!stack.isEmpty()) {

queue.add(stack.pop());

}

reverseQueue(queue);

}

private static void reverseQueue(Queue<Integer> queue) {

Stack<Integer> stack = new Stack<>();

while (!queue.isEmpty()) {

stack.push(queue.remove());

}

while (!stack.isEmpty()) {

queue.add(stack.pop());

}

}

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

queue.add(3);

queue.add(1);

queue.add(4);

System.out.println("Original Queue: " + queue);

sortQueue(queue);

System.out.println("Sorted Queue: " + queue);

}

}

Task 4: Stack Sorting In-Place

You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

public class StackSorter {

public static void sortStack(Stack<Integer> stack) {

Stack<Integer> tempStack = new Stack<>();

while (!stack.isEmpty()) {

int current = stack.pop();

while (!tempStack.isEmpty() && tempStack.peek() > current) {

stack.push(tempStack.pop());

}

tempStack.push(current);

}

while (!tempStack.isEmpty()) {

stack.push(tempStack.pop());

}

}

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(34);

stack.push(3);

stack.push(31);

stack.push(98);

System.out.println("Original Stack: " + stack);

sortStack(stack);

System.out.println("Sorted Stack: " + stack);

}

}

**Task 5: Removing Duplicates from a Sorted Linked List**

**A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.**

class ListNode {

int value;

ListNode next;

ListNode(int value) {

this.value = value;

this.next = null;

}

}

public class RemoveDuplicatesSortedList {

public static void removeDuplicates(ListNode head) {

ListNode current = head;

while (current != null && current.next != null) {

if (current.value == current.next.value) {

current.next = current.next.next;

} else {

current = current.next;

}

}

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.value + " ");

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(1);

head.next.next = new ListNode(2);

head.next.next.next = new ListNode(3);

head.next.next.next.next = new ListNode(3);

System.out.println("Original List:");

printList(head);

removeDuplicates(head);

System.out.println("List after removing duplicates:");

printList(head);

}

}

**Task 6: Searching for a Sequence in a Stack**

**Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.**

public class SequenceInStack {

public static boolean isSequencePresent(Stack<Integer> stack, int[] sequence) {

Stack<Integer> tempStack = new Stack<>();

int seqIndex = sequence.length - 1;

while (!stack.isEmpty()) {

int current = stack.pop();

tempStack.push(current);

if (seqIndex >= 0 && current == sequence[seqIndex]) {

seqIndex--;

} else if (seqIndex < sequence.length - 1 && current != sequence[seqIndex + 1]) {

seqIndex = sequence.length - 1;

}

}

while (!tempStack.isEmpty()) {

stack.push(tempStack.pop());

}

return seqIndex == -1;

}

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

stack.push(5);

stack.push(6);

stack.push(7);

int[] sequence = {4, 5, 6};

System.out.println("Stack: " + stack);

System.out.println("Sequence: " + java.util.Arrays.toString(sequence));

boolean result = isSequencePresent(stack, sequence);

System.out.println("Is sequence present: " + result);

System.out.println("Restored Stack: " + stack);

}

}

**Task 7: Merging Two Sorted Linked Lists**

**You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).**

class ListNode {

int value;

ListNode next;

ListNode(int value) {

this.value = value;

this.next = null;

}

}

public class MergeSortedLists {

public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (l1 != null && l2 != null) {

if (l1.value <= l2.value) {

current.next = l1;

l1 = l1.next;

} else {

current.next = l2;

l2 = l2.next;

}

current = current.next;

}

if (l1 != null) {

current.next = l1;

} else {

current.next = l2;

}

return dummy.next;

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.value + " ");

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

ListNode l1 = new ListNode(1);

l1.next = new ListNode(3);

l1.next.next = new ListNode(5);

ListNode l2 = new ListNode(2);

l2.next = new ListNode(4);

l2.next.next = new ListNode(6);

System.out.println("First List:");

printList(l1);

System.out.println("Second List:");

printList(l2);

ListNode mergedList = mergeTwoLists(l1, l2);

System.out.println("Merged List:");

printList(mergedList);

}

}

**Task 8: Circular Queue Binary Search**

**Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue**.

public class CircularQueueBinarySearch {

public static int binarySearchInRotatedArray(int[] arr, int target) {

int left = 0;

int right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

return mid;

}

if (arr[left] <= arr[mid]) {

if (arr[left] <= target && target < arr[mid]) {

right = mid - 1;

} else {

left = mid + 1;

}

}

else {

if (arr[mid] < target && target <= arr[right]) {

left = mid + 1;

} else {

right = mid - 1;

}

}

}

return -1;

}

public static void main(String[] args) {

int[] arr = {6, 7, 8, 9, 1, 2, 3, 4, 5};

int target = 3;

int result = binarySearchInRotatedArray(arr, target);

if (result != -1) {

System.out.println("Element " + target + " found at index " + result);

} else {

System.out.println("Element " + target + " not found.");

}

}

}

**Day 7 and 8:**

**Task 1: Balanced Binary Tree Check**

**Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.**

class TreeNode {

int value;

TreeNode left;

TreeNode right;

TreeNode(int value) {

this.value = value;

this.left = null;

this.right = null;

}

}

public class BalancedBinaryTree {

private static int checkHeight(TreeNode node) {

if (node == null) {

return 0;

}

int leftHeight = checkHeight(node.left);

if (leftHeight == -1) {

return -1; // Not balanced

}

int rightHeight = checkHeight(node.right);

if (rightHeight == -1) {

return -1; // Not balanced

}

if (Math.abs(leftHeight + 1 - rightHeight + 1) > 1) {

return -1; // Not balanced

}

return Math.max(leftHeight, rightHeight) + 1;

}

public static boolean isBalanced(TreeNode root) {

return checkHeight(root) != -1;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.left.left = new TreeNode(4);

root.left.right = new TreeNode(5);

root.right.right = new TreeNode(6);

root.left.left.left = new TreeNode(7);

System.out.println("Is the binary tree balanced? " + isBalanced(root));

TreeNode unbalancedRoot = new TreeNode(1);

unbalancedRoot.left = new TreeNode(2);

unbalancedRoot.left.left = new TreeNode(3);

unbalancedRoot.left.left.left = new TreeNode(4);

System.out.println("Is the binary tree balanced? " + isBalanced(unbalancedRoot));

}

}

**Task 2: Trie for Prefix Checking**

**Implement a trie data structure in C# that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.**

import java.util.HashMap;

import java.util.Map;

class TrieNode {

Map<Character, TrieNode> children;

boolean isEndOfWord;

public TrieNode() {

children = new HashMap<>();

isEndOfWord = false;

}

}

public class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

public void insert(String word) {

TrieNode current = root;

for (char ch : word.toCharArray()) {

current = current.children.computeIfAbsent(ch, c -> new TrieNode());

}

current.isEndOfWord = true;

} public boolean startsWith(String prefix) {

TrieNode current = root;

for (char ch : prefix.toCharArray()) {

TrieNode node = current.children.get(ch);

if (node == null) {

return false;

}

current = node;

}

return true;

} public static void main(String[] args) {

Trie trie = new Trie();

trie.insert("apple");

trie.insert("app");

trie.insert("application");

trie.insert("bat");

trie.insert("ball");

System.out.println(trie.startsWith("app")); // true

System.out.println(trie.startsWith("bat")); // true

System.out.println(trie.startsWith("ball")); // true

System.out.println(trie.startsWith("batman")); // false

System.out.println(trie.startsWith("cat")); // false

}

}

**Task 3: Implementing Heap Operations**

**Code a min-heap in C# with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation.**

public class MinHeap {

private int[] heap;

private int size;

private int capacity;

public MinHeap(int capacity) {

this.capacity = capacity;

this.size = 0;

this.heap = new int[capacity + 1];

}

public void insert(int value) {

if (size == capacity) {

System.out.println("Heap is full. Cannot insert.");

return;

}size++;

int index = size;

heap[index] = value;

while (index > 1 && heap[index / 2] > heap[index]) {

swap(index, index / 2);

index /= 2;

}

}

public int deleteMin() {

if (size == 0) {

System.out.println("Heap is empty. Cannot delete.");

return -1;

}int min = heap[1];

heap[1] = heap[size];

size--;

heapify(1);

return min;

}

public int getMin() {

if (size == 0) {

System.out.println("Heap is empty. No minimum element.");

return -1;

}

return heap[1];

}

private void heapify(int index) {

int left = 2 \* index;

int right = 2 \* index + 1;

int smallest = index;

if (left <= size && heap[left] < heap[index]) {

smallest = left;

}

if (right <= size && heap[right] < heap[smallest]) {

smallest = right;

}if (smallest != index) {

swap(index, smallest);

heapify(smallest);

}

}

private void swap(int i, int j) {

int temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}public static void main(String[] args) {

MinHeap minHeap = new MinHeap(10);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(8);

minHeap.insert(1);

System.out.println("Minimum element: " + minHeap.getMin());

minHeap.deleteMin();

System.out.println("Minimum element after deletion: " + minHeap.getMin());

minHeap.insert(2);

System.out.println("Minimum element after insertion: " + minHeap.getMin());

}

}

**Task 4: Graph Edge Addition Validation**

**Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.**

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

import java.util.Stack;

class Graph {

private int V;

private List<List<Integer>> adj;

public Graph(int V) {

this.V = V;

adj = new ArrayList<>(V);

for (int i = 0; i < V; i++) {

adj.add(new ArrayList<>());

}

}

public void addEdge(int u, int v) {

adj.get(u).add(v);

}

private boolean hasCycleUtil(int v, boolean[] visited, boolean[] recStack) {

if (!visited[v]) {

visited[v] = true;

recStack[v] = true;

for (int neighbor : adj.get(v)) {

if (!visited[neighbor] && hasCycleUtil(neighbor, visited, recStack)) {

return true;

} else if (recStack[neighbor]) {

return true;

}

}

}

recStack[v] = false;

return false;

}

public boolean hasCycleAfterAddingEdge(int u, int v) {

addEdge(u, v);

boolean[] visited = new boolean[V];

boolean[] recStack = new boolean[V];

for (int i = 0; i < V; i++) {

if (hasCycleUtil(i, visited, recStack)) {

adj.get(u).remove(Integer.valueOf(v));

return true;

}

}

adj.get(u).remove(Integer.valueOf(v));

return false;

}

}public class GraphEdgeValidation {

public static void main(String[] args) {

int V = 4;

Graph graph = new Graph(V);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(2, 0);

graph.addEdge(2, 3);

graph.addEdge(3, 3);

int u = 1, v = 3;

if (graph.hasCycleAfterAddingEdge(u, v)) {

System.out.println("Adding edge (" + u + ", " + v + ") creates a cycle.");

} else {

System.out.println("Adding edge (" + u + ", " + v + ") does not create a cycle.");

}

}

}

**Task 5: Breadth-First Search (BFS) Implementation**

**For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.**

import java.util.\*;

class Graph {

private int V;

private List<List<Integer>> adj;

public Graph(int V) {

this.V = V;

adj = new ArrayList<>(V);

for (int i = 0; i < V; i++) {

adj.add(new ArrayList<>());

}

}

public void addEdge(int u, int v) {

adj.get(u).add(v);

adj.get(v).add(u);

}

public void BFS(int start) {

boolean[] visited = new boolean[V];

Queue<Integer> queue = new LinkedList<>();

visited[start] = true;

queue.offer(start);

while (!queue.isEmpty()) {

int node = queue.poll();

System.out.print(node + " ");

for (int neighbor : adj.get(node)) {

if (!visited[neighbor]) {

visited[neighbor] = true;

queue.offer(neighbor);

} }}

}

}

public class BFSImplementation {

public static void main(String[] args) {

int V = 6;

Graph graph = new Graph(V);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(1, 4);

graph.addEdge(2, 4);

graph.addEdge(3, 5);

graph.addEdge(4, 5);

int startNode = 0;

System.out.println("BFS Traversal starting from node " + startNode + ":");

graph.BFS(startNode);

}

}

**Task 6: Depth-First Search (DFS) Recursive**

**Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.**

import java.util.\*;

class Graph {

private int V;

private List<List<Integer>> adj;

public Graph(int V) {

this.V = V;

adj = new ArrayList<>(V);

for (int i = 0; i < V; i++) {

adj.add(new ArrayList<>());

}

}

public void addEdge(int u, int v) {

adj.get(u).add(v);

adj.get(v).add(u);

} private void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

System.out.print(v + " ");

for (int neighbor : adj.get(v)) {

if (!visited[neighbor]) {

DFSUtil(neighbor, visited);

}

}

}

public void DFS(int start) {

boolean[] visited = new boolean[V];

DFSUtil(start, visited);

}

}public class DFSRecursive {

public static void main(String[] args) {

int V = 6;

Graph graph = new Graph(V);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(1, 4);

graph.addEdge(2, 4);

int startNode = 0;

System.out.println("DFS Traversal starting from node " + startNode + ":");

graph.DFS(startNode);

}

}

**Day 9 and 10:**

**Task 1: Dijkstra’s Shortest Path Finder**

**Code Dijkstra’s algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.**

import java.util.\*;

class Graph {

private int V;

private List<List<Node>> adj;

static class Node implements Comparable<Node> {

int vertex;

int distance;

Node(int vertex, int distance) {

this.vertex = vertex;

this.distance = distance;

} @Override

public int compareTo(Node other) {

return Integer.compare(this.distance, other.distance);

}

} public Graph(int V) {

this.V = V;

adj = new ArrayList<>(V);

for (int i = 0; i < V; i++) {

adj.add(new ArrayList<>());

}

}

public void addEdge(int u, int v, int weight) {

adj.get(u).add(new Node(v, weight));

}

public void dijkstra(int start) {

PriorityQueue<Node> pq = new PriorityQueue<>();

int[] distance = new int[V];

Arrays.fill(distance, Integer.MAX\_VALUE);

distance[start] = 0;

pq.offer(new Node(start, 0));

while (!pq.isEmpty()) {

Node node = pq.poll();

int u = node.vertex;

for (Node neighbor : adj.get(u)) {

int v = neighbor.vertex;

int weight = neighbor.distance;

if (distance[u] != Integer.MAX\_VALUE && distance[u] + weight < distance[v]) {

distance[v] = distance[u] + weight;

pq.offer(new Node(v, distance[v]));

}

}

}

System.out.println("Shortest distances from node " + start + ":");

for (int i = 0; i < V; i++) {

System.out.println("To node " + i + ": " + distance[i]);

}

}

}public class DijkstraShortestPath {

public static void main(String[] args) {

int V = 5;

int startNode = 0;

Graph graph = new Graph(V);

graph.addEdge(0, 1, 10);

graph.addEdge(0, 3, 5);

graph.addEdge(1, 2, 1);

graph.addEdge(1, 3, 2);

graph.dijkstra(startNode);

}

}

Task 2: Kruskal’s Algorithm for MST

Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

import java.util.\*;

class Edge implements Comparable<Edge> {

int src, dest, weight;

public Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

@Override

public int compareTo(Edge other) {

return Integer.compare(this.weight, other.weight);

}

}class Graph {

private int V;

private List<Edge> edges;

public Graph(int V) {

this.V = V;

edges = new ArrayList<>();

}

public void addEdge(int src, int dest, int weight) {

edges.add(new Edge(src, dest, weight));

}

public void kruskalMST() {

Collections.sort(edges);

int[] parent = new int[V];

for (int i = 0; i < V; i++) {

parent[i] = i;

}

List<Edge> mstEdges = new ArrayList<>();

for (Edge edge : edges) {

int srcParent = find(parent, edge.src);

int destParent = find(parent, edge.dest);

if (srcParent != destParent) {

mstEdges.add(edge);

union(parent, srcParent, destParent);

}

}

System.out.println("Edges of the Minimum Spanning Tree:");

for (Edge edge : mstEdges) {

System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight);

}

}

private int find(int[] parent, int node) {

if (parent[node] != node) {

parent[node] = find(parent, parent[node]); // Path compression

}

return parent[node];

}

private void union(int[] parent, int x, int y) {

int xParent = find(parent, x);

int yParent = find(parent, y);

parent[xParent] = yParent;

}

}

public class KruskalMST {

public static void main(String[] args) {

int V = 6;

Graph graph = new Graph(V);

graph.addEdge(0, 1, 4);

graph.addEdge(0, 2, 1);

graph.addEdge(1, 2, 3);

graph.addEdge(1, 3, 2);

graph.kruskalMST();

}

}

**Task 3: Union-Find for Cycle Detection**

**Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.**

import java.util.\*;

class UnionFind {

private int[] parent;

private int[] rank;

public UnionFind(int n) {

parent = new int[n];

rank = new int[n];

for (int i = 0; i < n; i++) {

parent[i] = i;

rank[i] = 0;

}

} public int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

} public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

}class Graph {

private int V;

private List<int[]> edges;

public Graph(int V) {

this.V = V;

edges = new ArrayList<>();

}

public void addEdge(int u, int v) {

edges.add(new int[]{u, v});

}

public boolean containsCycle() {

UnionFind uf = new UnionFind(V);

for (int[] edge : edges) {

int u = edge[0];

int v = edge[1];

int parentU = uf.find(u);

int parentV = uf.find(v);

if (parentU == parentV) {

return true;

} uf.union(u, v);

} return false;

}

}public class CycleDetection {

public static void main(String[] args) {

int V = 4;

Graph graph = new Graph(V);

graph.addEdge(0, 1);

graph.addEdge(1, 2);

if (graph.containsCycle()) {

System.out.println("The graph contains a cycle.");

} else {

System.out.println("The graph does not contain a cycle.");

}

}

}

**Day 11:**

**Task 1: String Operations**

**Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.**

public class StringOperations {

public static String extractMiddleSubstring(String str1, String str2, int length) {

String concatenated = str1 + str2;

StringBuilder reversed = new StringBuilder(concatenated).reverse();

int reversedLength = reversed.length();

if (reversedLength == 0 || length > reversedLength) {

return "Invalid input or substring length too large";

}

int startIndex = (reversedLength - length) / 2;

return reversed.substring(startIndex, startIndex + length);

}public static void main(String[] args) {

String str1 = "Hello";

String str2 = "World";

int length = 5;

String result = extractMiddleSubstring(str1, str2, length);

System.out.println("Middle Substring: " + result);

}

}

**Task 2: Naive Pattern Search**

**Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.**

public class NaivePatternSearch {

public static void searchPattern(String text, String pattern) {

int textLength = text.length();

int patternLength = pattern.length();

int comparisons = 0;

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

int j;

for (j = 0; j < patternLength; j++) {

comparisons++;

if (text.charAt(i + j) != pattern.charAt(j)) {

break;

}

} if (j == patternLength) {

System.out.println("Pattern found at index " + i);

}

}System.out.println("Total comparisons made: " + comparisons);

} public static void main(String[] args) {

String text = "AABAACAADAABAAABAA";

String pattern = "AABA";

System.out.println("Text: " + text);

System.out.println("Pattern: " + pattern);

System.out.println("Occurrences of pattern:");

searchPattern(text, pattern);

}

}