DataStructures and algorithms

Lab 5

Name:Pranjal bajaj

Cms:023-21-0022

Task 1

class Task {

    private int[] queue;

    private int front;

    private int rear;

    private int capacity;

    private int size;

    public Task(int capacity) {

        this.capacity = capacity;

        this.queue = new int[capacity];

        this.front = -1;

        this.rear = -1;

        this.size = 0;

    }

    public void enqueue(int value) {

        if (isFull()) {

            System.out.println("Queue is full. Cannot enqueue " + value);

            return;

        }

        if (isEmpty()) {

            front = 0;

            rear = 0;

        } else {

            rear = (rear + 1) % capacity;

        }

        queue[rear] = value;

        size++;

        System.out.println("Enqueued: " + value);

    }

    public int dequeue() {

        if (isEmpty()) {

            System.out.println("Queue is empty. Cannot dequeue.");

            return -1;

        }

        int value = queue[front];

        if (front == rear) {

            front = -1;

            rear = -1;

        } else {

                       front = (front + 1) % capacity;

        }

        size--;

        System.out.println("Dequeued: " + value);

        return value;

    }

    public int getFront() {

        if (isEmpty()) {

            System.out.println("Queue is empty.");

            return -1;

        }

        return queue[front];

    }

    public int getSize() {

        return size;

    }

    public boolean isEmpty() {

        return size == 0;

    }

public boolean isFull()

        return size == capacity;

    }

    public void printAll() {

        if (isEmpty()) {

            System.out.println("Queue is empty.");

            return;

        }

        System.out.print("Queue elements: ");

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

            System.out.print(queue[(front + i) % capacity] + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Task queue = new Task(5);

        queue.enqueue(10);

        queue.enqueue(20);

        queue.enqueue(30);

        queue.enqueue(40);

        queue.enqueue(50);

        queue.printAll();

        queue.enqueue(60);

        queue.dequeue();

        queue.dequeue();

        queue.printAll();

        queue.enqueue(60);

        queue.enqueue(70);

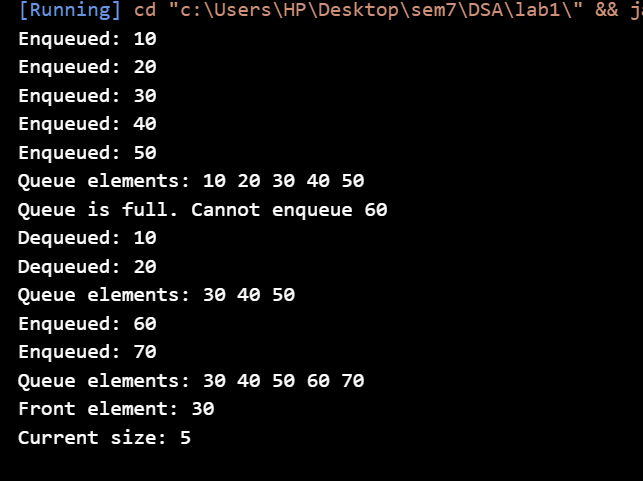
        queue.printAll();

        System.out.println("Front element: " + queue.getFront());

        System.out.println("Current size: " + queue.getSize());

    }

}



Task 2

class Node {

    int data;

    Node next;

    Node(int data) {

        this.data = data;

        this.next = null;

    }

}

class llqueue {

    private Node front;

    private Node rear;

    private int size;

    public llqueue() {

        this.front = null;

        this.rear = null;

        this.size = 0;

    }

    public void enqueue(int value) {

        Node newNode = new Node(value);

        if (isEmpty()) {

            front = newNode;

        } else {

            rear.next = newNode;

        }

        rear = newNode;

        System.out.println("Enqueued: " + value);

    }

    public int dequeue() {

        if (isEmpty()) {

            System.out.println("Queue is empty. Cannot dequeue.");

            return -1;

        }

        int value = front.data;

        front = front.next;

        if (front == null) {

            rear = null;

        }

        size--;

        System.out.println("Dequeued: " + value);

        return value;

    }

    public int getFront() {

        if (isEmpty()) {

            System.out.println("Queue is empty.");

            return -1;

        }

        return front.data;

    }

    public int getSize() {

        return size;

    }

    public boolean isEmpty() {

        return size == 0;

    }

    public void printAll() {

        if (isEmpty()) {

            System.out.println("Queue is empty.");

            return;

        }

        System.out.print("Queue elements: ");

        Node current = front;

        while (current != null) {

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

            current = current.next;

        }

        System.out.println();

    }

    public static void main(String[] args) {

        llqueue queue = new llqueue();

        queue.enqueue(10);

        queue.enqueue(20);

        queue.enqueue(30);

        queue.enqueue(40);

        queue.printAll();

        queue.dequeue();

        queue.printAll();

        queue.enqueue(50);

        queue.enqueue(60);

        queue.printAll();

        System.out.println("Front element: " + queue.getFront());

        System.out.println("Current size: " + queue.getSize());

        queue.dequeue();

        queue.dequeue();

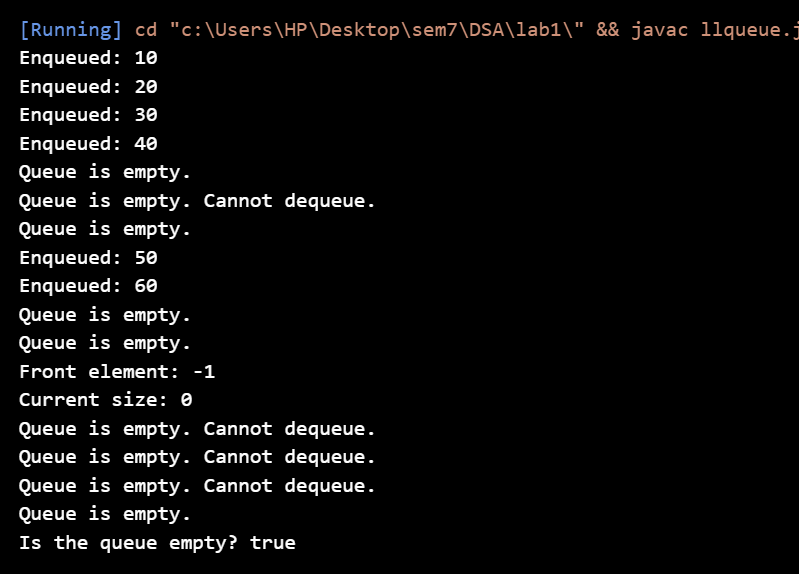
        queue.dequeue();

        queue.printAll();

        System.out.println("Is the queue empty? " + queue.isEmpty());

    }

}



Task 3

class Task {

    private int[] stack;

    private int capacity;

    private int top;

    public Task(int capacity) {

        this.capacity = capacity;

        this.stack = new int[capacity];

        this.top = -1;

    }

    public void push(int value) {

        if (isFull()) {

            System.out.println("Stack is full. Cannot push " + value);

            return;

        }

        stack[++top] = value;

        System.out.println("Pushed: " + value);

    }

    public int pop() {

        if (isEmpty()) {

            System.out.println("Stack is empty. Cannot pop.");

            return -1;

        }

        int value = stack[top--];

        System.out.println("Popped: " + value);

        return value;

    }

    public int peek() {

        if (isEmpty()) {

            System.out.println("Stack is empty.");

            return -1;

        }

        return stack[top];

    }

    public int getSize() {

        return top+1;

    }

    public boolean isEmpty() {

        return top == -1;

    }

    public boolean isFull() {

        return top == capacity - 1;

    }

    public void printAll() {

        if (isEmpty()) {

            System.out.println("Stack is empty.");

            return;

        }

        System.out.print("Stack elements: ");

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

            System.out.print(stack[i] + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Task stack = new Task(5);

        stack.push(10);

        stack.push(20);

        stack.push(30);

        stack.push(40);

        stack.push(50);

        stack.printAll();

        stack.push(60);

        stack.pop();

        stack.pop();

        stack.printAll();

        stack.push(60);

        stack.push(70);

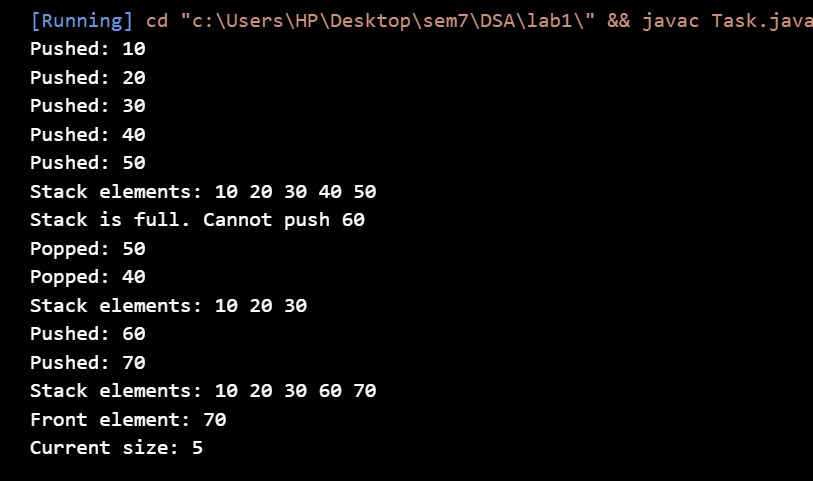
        stack.printAll();

        System.out.println("Front element: " + stack.peek());

        System.out.println("Current size: " + stack.getSize());

    }

}



Task 4

class Node {

    int data;

    Node next;

    Node(int data) {

        this.data = data;

        this.next = null;

    }

}

class llStack {

    private Node top;

    private int size;

    public llStack() {

        this.top = null;

        this.size = 0;

    }

    public void push(int value) {

        Node newNode = new Node(value);

        newNode.next = top;

        top = newNode;

        size++;

        System.out.println("pushed: " + value);

    }

    public int pop() {

        if (isEmpty()) {

            System.out.println("Stack is empty. Cannot pop.");

            return -1;

        }

        int value = top.data;

        top = top.next;

        size--;

        System.out.println("popped: " + value);

        return value;

    }

    public int peek() {

        if (isEmpty()) {

            System.out.println("Stack is empty.");

            return -1;

        }

        return top.data;

    }

    public int getSize() {

        return size;

    }

    public boolean isEmpty() {

        return top == null;

    }

    public void printAll() {

        if (isEmpty()) {

            System.out.println("Stack is empty.");

            return;

        }

        System.out.print("Stack elements: ");

        Node current = top;

        while (current != null) {

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

            current = current.next;

        }

        System.out.println();

    }

    public static void main(String[] args) {

        llStack Stack = new llStack();

        Stack.push(10);

        Stack.push(20);

        Stack.push(30);

        Stack.push(40);

        Stack.printAll();

        Stack.pop();

        Stack.printAll();

        Stack.push(50);

        Stack.push(60);

        Stack.printAll();

        System.out.println("top element: " + Stack.peek());

        System.out.println("Current size: " + Stack.getSize());

        Stack.pop();

        Stack.pop();

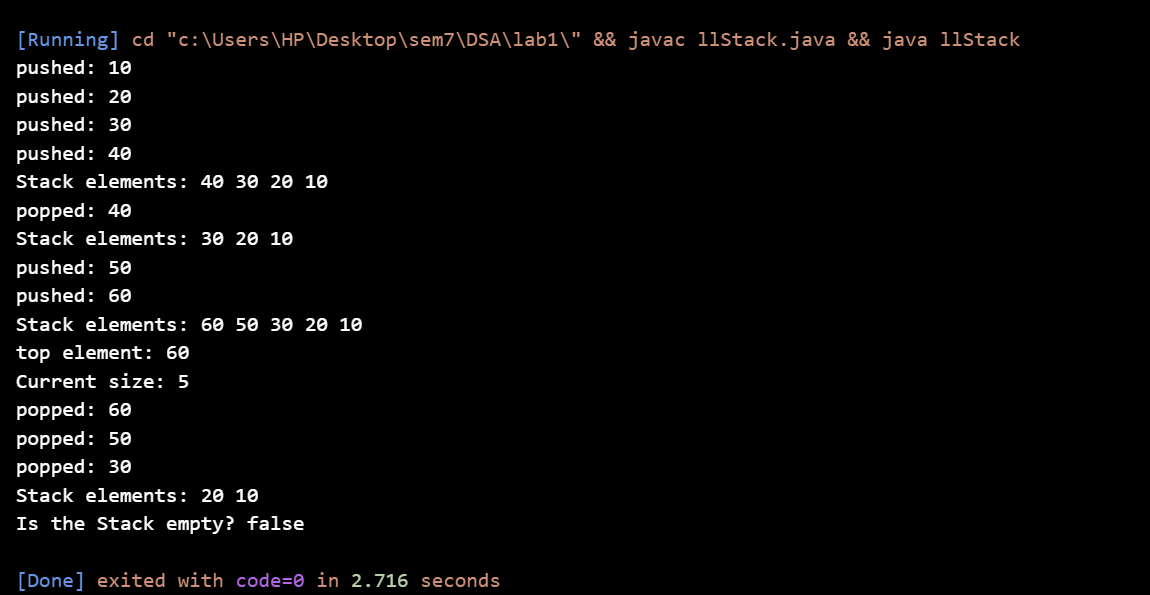
        Stack.pop();

        Stack.printAll();

        System.out.println("Is the Stack empty? " + Stack.isEmpty());

    }

}



Task 5

// Node class to represent each element in the linked list

class Node {

    char data;

    Node next;

    // Constructor to create a new node

    public Node(char data) {

        this.data = data;

        this.next = null;

    }

}

// LinkedList class to implement the stack functionality using linked list

class LinkedListStack {

    private Node top;

    // Constructor to initialize the stack

    public LinkedListStack() {

        this.top = null;

    }

    // Push a character onto the stack

    public void push(char data) {

        Node newNode = new Node(data);

        if (top == null) {

            top = newNode;

        } else {

            newNode.next = top;

            top = newNode;

        }

    }

    // Pop a character from the stack

    public char pop() {

        if (top == null) {

            System.out.println("Stack is empty");

            return '\0'; // Returning null character if stack is empty

        }

        char poppedData = top.data;

        top = top.next;

        return poppedData;

    }

    // Check if the stack is empty

    public boolean isEmpty() {

        return top == null;

    }

}

// Palindrome checker using custom linked list-based stack

public class llStack {

    // Function to check if the string is a palindrome

    public static boolean isPalindrome(String str) {

        // Remove spaces and convert to lowercase for uniformity

        String formattedStr = str.replaceAll("\\s+", "").toLowerCase();

        LinkedListStack stack = new LinkedListStack();

        // Push all characters of the string into the stack

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

            stack.push(formattedStr.charAt(i));

        }

        // Check for palindrome by popping and comparing with the original string

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

            if (formattedStr.charAt(i) != stack.pop()) {

                return false; // Not a palindrome

            }

        }

        return true; // Is a palindrome

    }

    public static void main(String[] args) {

        // Test cases

        String test1 = "madam";

        String test2 = "hello";

        String test3 = "A man a plan a canal Panama";

        // Checking the strings

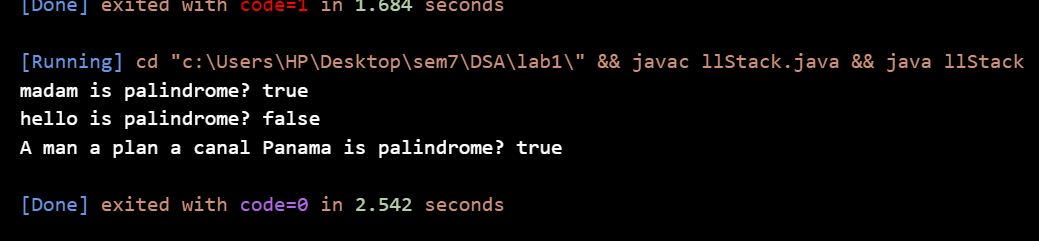
        System.out.println(test1 + " is palindrome? " + isPalindrome(test1)); // true

        System.out.println(test2 + " is palindrome? " + isPalindrome(test2)); // false

        System.out.println(test3 + " is palindrome? " + isPalindrome(test3)); // true

    }

}



Task 6

import java.util.Stack;

public class llStack {

    // Method to check if parentheses are balanced

    public static boolean isBalanced(String expression) {

        // Create a custom stack to hold opening brackets

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

        // Loop through each character in the expression

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

            // If opening bracket, push onto stack

            if (ch == '(' || ch == '{' || ch == '[') {

                stack.push(ch);

            }

            // If closing bracket, check if it matches the last opening bracket

            else if (ch == ')' || ch == '}' || ch == ']') {

                // If stack is empty or top doesn't match, return false

                if (stack.isEmpty()) {

                    return false;

                }

                char last = stack.pop();

                if (!isMatchingPair(last, ch)) {

                    return false;

                }

            }

        }

        // If stack is empty, all parentheses were matched

        return stack.isEmpty();

    }

    // Helper method to check if the pair of parentheses match

    private static boolean isMatchingPair(char open, char close) {

        return (open == '(' && close == ')') ||

               (open == '{' && close == '}') ||

               (open == '[' && close == ']');

    }

    // Main method to demonstrate the parentheses checker

    public static void main(String[] args) {

        String expression = "{[()]}";  // Test case 1: Balanced

        System.out.println("Is the expression " + expression + " balanced? " + isBalanced(expression));

        expression = "{[(])}";  // Test case 2: Not balanced

        System.out.println("Is the expression " + expression + " balanced? " + isBalanced(expression));

        expression = "(([]))";  // Test case 3: Balanced

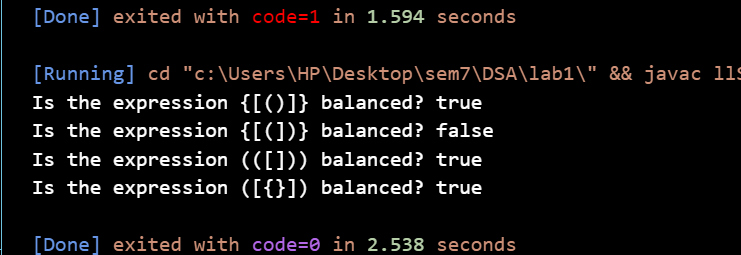
        System.out.println("Is the expression " + expression + " balanced? " + isBalanced(expression));

        expression = "([{}])";  // Test case 4: Balanced

        System.out.println("Is the expression " + expression + " balanced? " + isBalanced(expression));

    }

}



Task 7

import java.util.Stack;

public class DSAStack {

    // Method to sort the stack

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

        // Create an additional temporary stack

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

        // Process all elements in the original stack

        while (!stack.isEmpty()) {

            // Pop the top element from the original stack

            int current = stack.pop();

            // Transfer elements from tempStack to stack if they're larger than the current element

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

                stack.push(tempStack.pop());

            }

            // Push the current element onto the temporary stack

            tempStack.push(current);

        }

        // Transfer the sorted elements back to the original stack

        while (!tempStack.isEmpty()) {

            stack.push(tempStack.pop());

        }

    }

    // Main method to demonstrate stack sorting

    public static void main(String[] args) {

        // Create and populate the stack

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

        stack.push(34);

        stack.push(3);

        stack.push(31);

        stack.push(98);

        stack.push(92);

        stack.push(23);

        // Display the original stack

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

        // Sort the stack

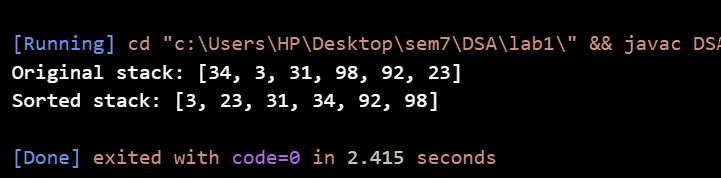
        sortStack(stack);

        // Display the sorted stack

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

    }

}



Task 8

import java.util.Stack;

public class DSAStack {

private Stack<Integer> stack1;

    private Stack<Integer> stack2;

    public DSAStack() {

        stack1 = new Stack<>();

        stack2 = new Stack<>();

    }

    // Enqueue operation: Add an element to the queue

    public void enqueue(int data) {

        // Push the element onto stack1

        stack1.push(data);

    }

    public int dequeue()  {

        if (stack1.isEmpty() && stack2.isEmpty()) {

            System.out.println("Queue is empty.");

        }

        if (stack2.isEmpty()) {

            while (!stack1.isEmpty()) {

                stack2.push(stack1.pop());

            }

        }

        // The top element of stack2 is the front of the queue

        return stack2.pop();

    }

    public int peek()  {

        if (stack1.isEmpty() && stack2.isEmpty()) {

            System.out.println("Queue is empty.");

        }

         if (stack2.isEmpty()) {

            while (!stack1.isEmpty()) {

                stack2.push(stack1.pop());

            }

        }

        // The top element of stack2 is the front of the queue

        return stack2.peek();

    }

    public boolean isEmpty() {

        return stack1.isEmpty() && stack2.isEmpty();

    }

    // size operation: Return the number of elements in the queue

    public int size() {

        return stack1.size() + stack2.size();

    }

public static void main(String[] args) {

            DSAStack queue = new DSAStack();

            // Enqueue elements

            queue.enqueue(10);

            queue.enqueue(20);

            queue.enqueue(30);

            System.out.println("Enqueued 10, 20, 30");

            // Dequeue element and print it

            System.out.println("Dequeued: " + queue.dequeue()); // Should print 10

            // Enqueue more elements

            queue.enqueue(40);

            System.out.println("Enqueued 40");

            // Peek the front element

            System.out.println("Peek: " + queue.peek()); // Should print 20

            // Dequeue the rest of the elements

            System.out.println("Dequeued: " + queue.dequeue()); // Should print 20

            System.out.println("Dequeued: " + queue.dequeue()); // Should print 30

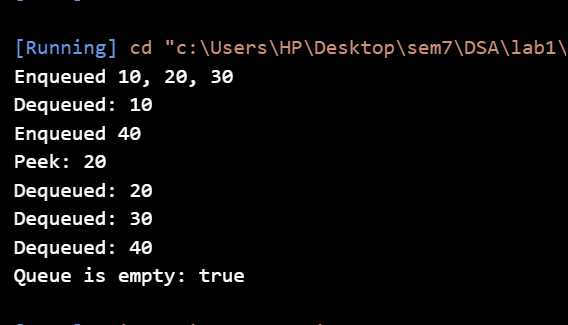
            System.out.println("Dequeued: " + queue.dequeue()); // Should print 40

            // Check if queue is empty

            System.out.println("Queue is empty: " + queue.isEmpty());

        }

}



Task9

import java.util.Stack;

class DSAStack {

    private Stack<Integer> stack;

    private Stack<Integer> minStack;

    // Constructor to initialize the stacks

    public DSAStack() {

        stack = new Stack<>();

        minStack = new Stack<>();

    }

    // Method to push an element onto the stack

    public void push(int value) {

        stack.push(value);

        // Push the minimum value onto the minStack

        if (minStack.isEmpty() || value <= minStack.peek()) {

            minStack.push(value);

        }

    }

    // Method to pop the top element from the stack

    public int pop() {

        if (stack.isEmpty()) {

            System.out.println("Stack is empty");

        }

        int value = stack.pop();

        // If the popped value is the current minimum, pop it from minStack as well

        if (value == minStack.peek()) {

            minStack.pop();

        }

        return value;

    }

    // Method to get the top element of the stack

    public int top() {

        if (stack.isEmpty()) {

            System.out.println("Stack is empty");

        }

        return stack.peek();

    }

    // Method to get the minimum element in O(1) time

    public int getMinimum() {

        if (minStack.isEmpty()) {

            System.out.println("Stack is empty");

        }

        return minStack.peek();

    }

    // Method to check if the stack is empty

    public boolean isEmpty() {

        return stack.isEmpty();

    }

    // Method to get the size of the stack

    public int size() {

        return stack.size();

    }

    // Main method to demonstrate the MinStack functionality

    public static void main(String[] args) {

        DSAStack minStack = new DSAStack();

        minStack.push(3);

        System.out.println("Minimum: " + minStack.getMinimum()); // 3

        minStack.push(5);

        System.out.println("Minimum: " + minStack.getMinimum()); // 3

        minStack.push(2);

        System.out.println("Minimum: " + minStack.getMinimum()); // 2

        minStack.push(1);

        System.out.println("Minimum: " + minStack.getMinimum()); // 1

        minStack.pop(); // pops 1

        System.out.println("Minimum after popping 1: " + minStack.getMinimum()); // 2

        minStack.pop(); // pops 2

        System.out.println("Minimum after popping 2: " + minStack.getMinimum()); // 3

    }

}

