Q1.import java.util.\*;

public class NextGreaterElement {

public static int[] findNextGreaterElements(int[] arr) {

int n = arr.length;

int[] result = new int[n];

Arrays.fill(result, -1); // Initialize result array with -1

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

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

while (!stack.isEmpty() && arr[i] > arr[stack.peek()]) {

int index = stack.pop();

result[index] = arr[i];

}

stack.push(i);

}

return result;

}

public static void main(String[] args) {

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

int[] nextGreater = findNextGreaterElements(arr);

System.out.println("Next Greater Elements:");

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

System.out.println(arr[i] + " -> " + nextGreater[i]);

}

}

}

Q2.import java.util.\*;

public class NearestSmallerNumber {

public static int[] findNearestSmallerNumbers(int[] arr) {

int n = arr.length;

int[] result = new int[n];

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

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

while (!stack.isEmpty() && stack.peek() >= arr[i]) {

stack.pop();

}

result[i] = stack.isEmpty() ? -1 : stack.peek();

stack.push(arr[i]);

}

return result;

}

public static void main(String[] args) {

int[] arr = {1, 6, 2};

int[] nearestSmaller = findNearestSmallerNumbers(arr);

System.out.println("Nearest Smaller Numbers:");

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

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

}

}

}

Q3.import java.util.LinkedList;

import java.util.Queue;

public class StackUsingQueues {

private Queue<Integer> q1;

private Queue<Integer> q2;

public StackUsingQueues() {

q1 = new LinkedList<>();

q2 = new LinkedList<>();

}

public void push(int x) {

q2.add(x);

while (!q1.isEmpty()) {

q2.add(q1.poll());

}

Queue<Integer> temp = q1;

q1 = q2;

q2 = temp;

}

public int pop() {

if (q1.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return q1.poll();

}

public static void main(String[] args) {

StackUsingQueues stack = new StackUsingQueues();

stack.push(2);

stack.push(3);

System.out.println(stack.pop()); // Output: 3

stack.push(4);

System.out.println(stack.pop()); // Output: 4

}

}

Q4.import java.util.Stack;

public class ReverseStack {

public static void reverseStack(Stack<Integer> st) {

if (st.isEmpty() || st.size() == 1) {

return;

}

int x = st.pop();

reverseStack(st);

insertAtBottom(st, x);

}

private static void insertAtBottom(Stack<Integer> st, int x) {

if (st.isEmpty()) {

st.push(x);

return;

}

int y = st.pop();

insertAtBottom(st, x);

st.push(y);

}

public static void main(String[] args) {

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

st.push(3);

st.push(2);

st.push(1);

st.push(7);

st.push(6);

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

reverseStack(st);

System.out.println("Reversed Stack: " + st);

}

}

Q5.import java.util.Stack;

public class ReverseStringUsingStack {

public static String reverseString(String str) {

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

// Push each character onto the stack

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

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

}

// Pop each character from the stack and append it to the reversed string

StringBuilder reversed = new StringBuilder();

while (!stack.isEmpty()) {

reversed.append(stack.pop());

}

return reversed.toString();

}

public static void main(String[] args) {

String str = "GeeksforGeeks";

String reversed = reverseString(str);

System.out.println("Original String: " + str);

System.out.println("Reversed String: " + reversed);

}

}

Q6.import java.util.Stack;

public class EvaluatePostfixExpression {

public static int evaluatePostfix(String postfix) {

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

// Iterate through each character in the postfix expression

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

char ch = postfix.charAt(i);

// If the character is a digit, push it onto the stack

if (Character.isDigit(ch)) {

stack.push(ch - '0');

} else {

// If the character is an operator, pop two operands from the stack

int operand2 = stack.pop();

int operand1 = stack.pop();

// Perform the operation and push the result onto the stack

int result = performOperation(ch, operand1, operand2);

stack.push(result);

}

}

// The final result will be left on the stack

return stack.pop();

}

// Helper method to perform arithmetic operations

private static int performOperation(char operator, int operand1, int operand2) {

switch (operator) {

case '+':

return operand1 + operand2;

case '-':

return operand1 - operand2;

case '\*':

return operand1 \* operand2;

case '/':

return operand1 / operand2;

default:

throw new IllegalArgumentException("Invalid operator: " + operator);

}

}

public static void main(String[] args) {

String postfix = "231\*+9-";

int result = evaluatePostfix(postfix);

System.out.println("Postfix expression: " + postfix);

System.out.println("Result: " + result);

}

}

Q7.import java.util.Stack;

class MinStack {

Stack<Integer> stack;

Stack<Integer> minStack;

/\*\* initialize your data structure here \*/

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

public void push(int val) {

stack.push(val);

// Update the minimum stack if the new element is smaller or equal to the current minimum

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

minStack.push(val);

}

}

public void pop() {

// If the element to be popped is the current minimum, remove it from the minimum stack

if (stack.peek().equals(minStack.peek())) {

minStack.pop();

}

stack.pop();

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

public class Main {

public static void main(String[] args) {

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

System.out.println(minStack.getMin()); // Output: -3

minStack.pop();

System.out.println(minStack.top()); // Output: 0

System.out.println(minStack.getMin()); // Output: -2

}

}

Q8.class Solution {

public int trap(int[] height) {

if (height == null || height.length == 0) {

return 0;

}

int left = 0;

int right = height.length - 1;

int leftMax = 0;

int rightMax = 0;

int water = 0;

while (left < right) {

if (height[left] < height[right]) {

// Update the leftMax if necessary

if (height[left] >= leftMax) {

leftMax = height[left];

} else {

// Calculate the trapped water on the left side

water += leftMax - height[left];

}

left++;

} else {

// Update the rightMax if necessary

if (height[right] >= rightMax) {

rightMax = height[right];

} else {

// Calculate the trapped water on the right side

water += rightMax - height[right];

}

right--;

}

}

return water;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

int[] height = {0,1,0,2,1,0,1,3,2,1,2,1};

int trappedWater = solution.trap(height);

System.out.println("Trapped water: " + trappedWater);

}

}