**Assignment 3**

**Solve the assignment with following thing to be added in each question.**

-Program

-Flow chart

-Explanation

-Output

-Time and Space complexity

Submission Date: 01/10/2024

**1. Implement a Stack using an array.**

* **Test Case 1**:  
  Input: Push 5, 3, 7, Pop  
  Output: Stack = [5, 3], Popped element = 7
* **Test Case 2**:  
  Input: Push 10, Push 20, Pop, Push 15  
  Output: Stack = [10, 15], Popped element = 20

package Org.example.demo;

class Stack {

private int[] stackArray; // Array to store the stack elements

private int top; // Index of the top element

private int maxSize; // Maximum size of the stack

// Constructor to initialize the stack

public Stack(int size) {

stackArray = new int[size];

maxSize = size;

top = -1; // Initially, the stack is empty

}

// Push operation

public void push(int value) {

if (top == maxSize - 1) {

System.out.println("Stack is full, cannot push.");

} else {

stackArray[++top] = value;

}

}

// Pop operation

public int pop() {

if (top == -1) {

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

return -1; // Return a sentinel value

} else {

return stackArray[top--];

}

}

// Display stack elements

public void display() {

if (top == -1) {

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

} else {

System.out.print("Stack = [");

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

System.out.print(stackArray[i]);

if (i < top) System.out.print(", ");

}

System.out.println("]");

}

}

}

public class StackDemo {

public static void main(String[] args) {

Stack stack = new Stack(5); // Create a stack with a maximum size of 5

// Test Case 1

System.out.println("Test Case 1:");

stack.push(5);

stack.push(3);

stack.push(7);

stack.display();

int poppedElement = stack.pop();

System.out.println("Popped element = " + poppedElement);

stack.display();

// Test Case 2

System.out.println("\nTest Case 2:");

stack.push(10);

stack.push(20);

stack.display();

poppedElement = stack.pop();

System.out.println("Popped element = " + poppedElement);

stack.push(15);

stack.display();

}

}

**2. Check for balanced parentheses using a stack.**

* **Test Case 1**:  
  Input: "({[()]})"  
  Output: Balanced
* **Test Case 2**:  
  Input: "([)]"  
  Output: Not Balanced

package Org.example.demo;

import java.util.Stack;

public class ParenthesesChecker {

// Function to check if the parentheses are balanced

public static boolean areParenthesesBalanced(String expression) {

// Create a stack to hold opening parentheses

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

// Loop through each character in the expression

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

// If it's an opening bracket, push to the stack

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

stack.push(ch);

}

// If it's a closing bracket, check for matching pair

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

// If stack is empty, it means there is no matching opening bracket

if (stack.isEmpty()) {

return false;

}

// Pop the top element from the stack and check if it matches

char top = stack.pop();

if (!isMatchingPair(top, ch)) {

return false;

}

}

}

// If stack is empty, all parentheses were balanced

return stack.isEmpty();

}

// Helper function to check if the characters form a matching pair

private static boolean isMatchingPair(char opening, char closing) {

return (opening == '(' && closing == ')') ||

(opening == '{' && closing == '}') ||

(opening == '[' && closing == ']');

}

// Main function to test the parentheses checker

public static void main(String[] args) {

// Test Case 1

String expression1 = "({[()]})";

if (areParenthesesBalanced(expression1)) {

System.out.println("Balanced");

} else {

System.out.println("Not Balanced");

}

// Test Case 2

String expression2 = "([)]";

if (areParenthesesBalanced(expression2)) {

System.out.println("Balanced");

} else {

System.out.println("Not Balanced");

}

}

}

**3. Reverse a string using a stack.**

* **Test Case 1**:  
  Input: "hello"  
  Output: "olleh"
* **Test Case 2**:  
  Input: "world"  
  Output: "dlrow"

package Org.example.demo;

import java.util.Stack;

public class StringReverser {

// Function to reverse a string using a stack

public static String reverseString(String input) {

// Create a stack to hold characters

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

// Push all characters of the string onto the stack

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

stack.push(ch);

}

// Create a StringBuilder to store the reversed string

StringBuilder reversed = new StringBuilder();

// Pop all characters from the stack to reverse the string

while (!stack.isEmpty()) {

reversed.append(stack.pop());

}

return reversed.toString();

}

// Main function to test the string reverser

public static void main(String[] args) {

// Test Case 1

String input1 = "hello";

String reversed1 = reverseString(input1);

System.out.println("Input: " + input1 + ", Reversed: " + reversed1);

// Test Case 2

String input2 = "world";

String reversed2 = reverseString(input2);

System.out.println("Input: " + input2 + ", Reversed: " + reversed2);

}

}

**4. Evaluate a postfix expression using a stack.**

* **Test Case 1**:  
  Input: "5 3 + 2 \*"  
  Output: 16
* **Test Case 2**:  
  Input: "4 5 \* 6 /"  
  Output: 3

package Org.example.demo;

import java.util.Stack;

public class PostfixEvaluator {

// Function to evaluate a postfix expression

public static int evaluatePostfix(String expression) {

// Create a stack to store operands

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

// Split the expression by spaces to get tokens

String[] tokens = expression.split(" ");

// Traverse through each token in the expression

for (String token : tokens) {

// If the token is a number, push it onto the stack

if (isNumeric(token)) {

stack.push(Integer.parseInt(token));

}

// If the token is an operator, pop two operands, apply the operator, and push the result

else {

int operand2 = stack.pop();

int operand1 = stack.pop();

int result = applyOperator(token, operand1, operand2);

stack.push(result);

}

}

// The final result will be the only value left in the stack

return stack.pop();

}

// Function to check if a token is a number

private static boolean isNumeric(String str) {

try {

Integer.parseInt(str);

return true;

} catch (NumberFormatException e) {

return false;

}

}

// Function to apply an operator to two operands

private static int applyOperator(String 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);

}

}

// Main method to test the postfix evaluator

public static void main(String[] args) {

// Test Case 1

String expression1 = "5 3 + 2 \*";

System.out.println("Input: " + expression1 + ", Output: " + evaluatePostfix(expression1));

// Test Case 2

String expression2 = "4 5 \* 6 /";

System.out.println("Input: " + expression2 + ", Output: " + evaluatePostfix(expression2));

}

}

**5. Convert an infix expression to postfix using a stack.**

* **Test Case 1**:  
  Input: "A + B \* C"  
  Output: "A B C \* +"
* **Test Case 2**:  
  Input: "A \* B + C / D"  
  Output: "A B \* C D / +"

package Org.example.demo;

import java.util.Stack;

public class InfixToPostfix {

// Function to check if a character is an operand (i.e., A-Z or 0-9)

private static boolean isOperand(char ch) {

return Character.isLetterOrDigit(ch);

}

// Function to get precedence of an operator

private static int getPrecedence(char operator) {

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

default:

return -1;

}

}

// Function to convert infix expression to postfix expression

public static String infixToPostfix(String expression) {

// Stack to hold operators

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

// StringBuilder to build the postfix expression

StringBuilder postfix = new StringBuilder();

// Traverse through each character in the infix expression

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

char ch = expression.charAt(i);

// If the character is an operand, add it to the output (postfix expression)

if (isOperand(ch)) {

postfix.append(ch).append(' ');

}

// If the character is '(', push it onto the stack

else if (ch == '(') {

stack.push(ch);

}

// If the character is ')', pop from the stack until '(' is found

else if (ch == ')') {

while (!stack.isEmpty() && stack.peek() != '(') {

postfix.append(stack.pop()).append(' ');

}

stack.pop(); // Remove the '(' from the stack

}

// If the character is an operator

else {

// Pop operators from the stack to the postfix output until an operator with less precedence is found

while (!stack.isEmpty() && getPrecedence(ch) <= getPrecedence(stack.peek())) {

postfix.append(stack.pop()).append(' ');

}

stack.push(ch); // Push the current operator onto the stack

**6. Implement a Queue using an array.**

* **Test Case 1**:  
  Input: Enqueue 5, Enqueue 10, Dequeue  
  Output: Queue = [10], Dequeued element = 5
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, Dequeue, Dequeue  
  Output: Queue = [3], Dequeued elements = 1, 2

package Org.example.demo;

public class ArrayQueue {

private int[] queue;

private int front;

private int rear;

private int size;

private int capacity;

// Constructor to initialize the queue with a given capacity

public ArrayQueue(int capacity) {

this.capacity = capacity;

this.queue = new int[capacity];

this.front = 0;

this.rear = -1;

this.size = 0;

}

// Function to check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Function to check if the queue is full

public boolean isFull() {

return size == capacity;

}

// Function to enqueue (add) an element to the queue

public void enqueue(int element) {

if (isFull()) {

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

return;

}

rear = (rear + 1) % capacity; // Circular increment

queue[rear] = element;

size++;

}

// Function to dequeue (remove) an element from the queue

public int dequeue() {

if (isEmpty()) {

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

return -1;

}

int dequeuedElement = queue[front];

front = (front + 1) % capacity; // Circular increment

size--;

return dequeuedElement;

}

// Function to display the current state of the queue

public void displayQueue() {

if (isEmpty()) {

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

return;

}

System.out.print("Queue = [");

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

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

if (i < size - 1) {

System.out.print(", ");

}

}

System.out.println("]");

}

// Main method to test the queue implementation

public static void main(String[] args) {

ArrayQueue queue = new ArrayQueue(5); // Initializing a queue with capacity 5

// Test Case 1

queue.enqueue(5);

queue.enqueue(10);

int dequeued1 = queue.dequeue();

System.out.println("Dequeued element = " + dequeued1);

queue.displayQueue();

// Test Case 2

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

int dequeued2 = queue.dequeue();

int dequeued3 = queue.dequeue();

System.out.println("Dequeued elements = " + dequeued2 + ", " + dequeued3);

queue.displayQueue();

}

}

**7. Implement a Circular Queue using an array.**

* **Test Case 1**:  
  Input: Enqueue 4, 5, 6, 7, Dequeue, Enqueue 8  
  Output: Queue = [8, 5, 6, 7]
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, 4, Dequeue, Dequeue, Enqueue 5  
  Output: Queue = [5, 3, 4]

package Org.example.demo;

public class CircularQueue {

private int[] queue;

private int front;

private int rear;

private int size;

private int capacity;

// Constructor to initialize the queue with a given capacity

public CircularQueue(int capacity) {

this.capacity = capacity;

this.queue = new int[capacity];

this.front = 0;

this.rear = -1;

this.size = 0;

}

// Function to check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Function to check if the queue is full

public boolean isFull() {

return size == capacity;

}

// Function to enqueue (add) an element to the queue

public void enqueue(int element) {

if (isFull()) {

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

return;

}

rear = (rear + 1) % capacity; // Circular increment for rear

queue[rear] = element;

size++;

}

// Function to dequeue (remove) an element from the queue

public int dequeue() {

if (isEmpty()) {

System.out.println("Queue”);

**8. Implement a Queue using two Stacks.**

* **Test Case 1**:  
  Input: Enqueue 3, Enqueue 7, Dequeue  
  Output: Queue = [7], Dequeued element = 3
* **Test Case 2**:  
  Input: Enqueue 10, 20, Dequeue, Dequeue  
  Output: Queue = [], Dequeued elements = 10, 20

import java.util.Stack;

public class QueueUsingTwoStacks {

private Stack<Integer> stack1;

private Stack<Integer> stack2;

// Constructor to initialize the two stacks

public QueueUsingTwoStacks() {

stack1 = new Stack<>();

stack2 = new Stack<>();

}

// Function to enqueue an element into the queue

public void enqueue(int data) {

stack1.push(data);

}

// Function to dequeue an element from the queue

public int dequeue() {

if (stack2.isEmpty()) {

// Transfer elements from stack1 to stack2 if stack2 is empty

if (stack1.isEmpty()) {

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

return -1;

**9. Implement a Min-Heap.**

* **Test Case 1**:  
  Input: Insert 10, 15, 20, 17, Extract Min  
  Output: Min-Heap = [15, 17, 20], Extracted Min = 10
* **Test Case 2**:  
  Input: Insert 30, 40, 20, 50, Extract Min  
  Output: Min-Heap = [30, 40, 50], Extracted Min = 20

import java.util.ArrayList;

import java.util.Collections;

public class MinHeap {

private ArrayList<Integer> heap;

// Constructor to initialize the min-heap

public MinHeap() {

heap = new ArrayList<>();

}

// Function to insert a new element into the heap

public void insert(int element) {

heap.add(element); // Add the new element at the end

int index = heap.size() - 1;

// Bubble up to maintain heap property

while (index > 0) {

int parentIndex = (index - 1) / 2;

if (heap.get(parentIndex) > heap.get(index)) {

Collections.swap(heap, parentIndex, index);

index = parentIndex; // Move up

} else {

break;

}

}

}

// Function to extract the minimum element (root)

public int extractMin() {

if (heap.isEmpty()) {

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

return -1;

}

int min = heap.get(0); // The root element (min element)

// Replace root with the last element and remove the last element

heap.set(0, heap.get(heap.size() - 1));

heap.remove(heap.size() - 1);

// Bubble down to maintain heap property

int index = 0;

while (index < heap.size()) {

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

int smallest = index;

// Find the smallest among the current node and its children

if (leftChild < heap.size() && heap.get(leftChild) < heap.get(smallest)) {

smallest = leftChild;

}

if (rightChild < heap.size() && heap.get(rightChild) < heap.get(smallest)) {

smallest = rightChild;

}

// If the smallest is not the current node, swap

if (smallest != index) {

Collections.swap(heap, index, smallest);

index = smallest; // Move down

} else {

break;

}

}

return min;

}

// Function to display the current state of the heap

public void displayHeap() {

System.out.println("Min-Heap = " + heap);

}

// Main method to test the Min-Heap implementation

public static void main(String[] args) {

MinHeap minHeap = new MinHeap();

// Test Case 1

minHeap.insert(10);

minHeap.insert(15);

minHeap.insert(20);

minHeap.insert(17);

int extractedMin1 = minHeap.extractMin();

System.out.println("Extracted Min = " + extractedMin1);

minHeap.displayHeap();

// Test Case 2

minHeap.insert(30);

minHeap.insert(40);

minHeap.insert(20);

minHeap.insert(50);

int extractedMin2 = minHeap.extractMin();

System.out.println("Extracted Min = " + extractedMin2);

minHeap.displayHeap();

}

}

**10. Implement a Max-Heap.**

* **Test Case 1**:  
  Input: Insert 12, 7, 15, 5, Extract Max  
  Output: Max-Heap = [12, 7, 5], Extracted Max = 15
* **Test Case 2**:  
  Input: Insert 8, 20, 10, 3, Extract Max  
  Output: Max-Heap = [10, 8, 3], Extracted Max = 20

import java.util.ArrayList;

import java.util.Collections;

public class MaxHeap {

private ArrayList<Integer> heap;

// Constructor to initialize the max-heap

public MaxHeap() {

heap = new ArrayList<>();

}

// Function to insert a new element into the heap

public void insert(int element) {

heap.add(element); // Add the element at the end

int index = heap.size() - 1;

// Bubble up to maintain heap property

while (index > 0) {

int parentIndex = (index - 1) / 2;

if (heap.get(parentIndex) < heap.get(index)) {

Collections.swap(heap, parentIndex, index);

index = parentIndex; // Move up

} else {

break;

}

}

}

// Function to extract the maximum element (root)

public int extractMax() {

if (heap.isEmpty()) {

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

return -1;

}

int max = heap.get(0); // The root element (max element)

// Replace root with the last element and remove the last element

heap.set(0, heap.get(heap.size() - 1));

heap.remove(heap.size() - 1);

// Bubble down to maintain heap property

int index = 0;

while (index < heap.size()) {

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

int largest = index;

// Find the largest among the current node and its children

if (leftChild < heap.size() && heap.get(leftChild) > heap.get(largest)) {

largest = leftChild;

}

if (rightChild < heap.size() && heap.get(rightChild) > heap.get(largest)) {

largest = rightChild;

}

// If the largest is not the current node, swap

if (largest != index) {

Collections.swap(heap, index, largest);

index = largest; // Move down

} else {

break;

}

}

**11. Sort an array using a heap (Heap Sort).**

* **Test Case 1**:  
  Input: [5, 1, 12, 3, 9]  
  Output: [1, 3, 5, 9, 12]
* **Test Case 2**:  
  Input: [20, 15, 8, 10]  
  Output: [8, 10, 15, 20]

import java.util.Arrays;

public class HeapSort {

// Function to perform heap sort on the given array

public static void heapSort(int[] array) {

int n = array.length;

// Build a max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(array, n, i);

}

// Extract elements from the heap one by one

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

// Swap the root of the heap with the last element

int temp = array[i];

array[i] = array[0];

array[0] = temp;

// Call heapify on the reduced heap

heapify(array, i, 0);

}

}

// Function to maintain the heap property

private static void heapify(int[] array, int n, int i) {

int largest = i; // Initialize largest as root

int leftChild = 2 \* i + 1; // left = 2\*i + 1

int rightChild = 2 \* i + 2; // right = 2\*i + 2

// If the left child is larger than the root

if (leftChild < n && array[leftChild] > array[largest]) {

largest = leftChild;

}

// If the right child is larger than the largest so far

if (rightChild < n && array[rightChild] > array[largest]) {

largest = rightChild;

}

// If the largest is not root

if (largest != i) {

int swap = array[i];

array[i] = array[largest];

array[largest] = swap;

// Recursively heapify the affected subtree

heapify(array, n, largest);

}

}

// Main method to test the Heap Sort implementation

public static void main(String[] args) {

// Test Case 1

int[] array1 = {5, 1, 12, 3, 9};

heapSort(array1);

System.out.println("Sorted Array: " + Arrays.toString(array1)); // Output: [1, 3, 5, 9, 12]

// Test Case 2

int[] array2 = {20, 15, 8, 10};

heapSort(array2);

System.out.println("Sorted Array: " + Arrays.toString(array2)); // Output: [8, 10, 15, 20]

}

}

**13. Implement a Priority Queue using a heap.**

* **Test Case 1**:  
  Input: Enqueue with priorities: 3 (priority 1), 10 (priority 3), 5 (priority 2), Dequeue  
  Output: Dequeued element = 10 (highest priority), Priority Queue = [5, 3]
* **Test Case 2**:  
  Input: Enqueue with priorities: 7 (priority 4), 8 (priority 2), 6 (priority 3), Dequeue  
  Output: Dequeued element = 7, Priority Queue = [6, 8]

import java.util.Arrays;

class PriorityQueue {

private class Node {

int value;

int priority;

Node(int value, int priority) {

this.value = value;

this.priority = priority;

}

}

private Node[] heap;

private int size;

private static final int CAPACITY = 10;

public PriorityQueue() {

heap = new Node[CAPACITY];

size = 0;

}

// Method to enqueue elements with priorities

public void enqueue(int value, int priority) {

if (size >= heap.length) {

resize();

}

heap[size] = new Node(value, priority);

size++;

heapifyUp();

}

// Method to dequeue the element with the highest priority

public int dequeue() {

if (size == 0) {

throw new IllegalStateException("Priority queue is empty");

}

int highestPriorityValue = heap[0].value;

heap[0] = heap[size - 1];

size--;

heapifyDown();

return highestPriorityValue;

}

// Method to resize the heap array

private void resize() {

heap = Arrays.copyOf(heap, heap.length \* 2);

}

// Method to maintain the heap property after enqueue

private void heapifyUp() {

int index = size - 1;

while (index > 0) {

int parentIndex = (index - 1) / 2;

if (heap[index].priority > heap[parentIndex].priority) {

swap(index, parentIndex);

index = parentIndex;

} else {

break;

}

}

}

// Method to maintain the heap property after dequeue

private void heapifyDown() {

int index = 0;

while (index < size) {

int leftChildIndex = 2 \* index + 1;

int rightChildIndex = 2 \* index + 2;

int largestIndex = index;

if (leftChildIndex < size && heap[leftChildIndex].priority > heap[largestIndex].priority) {

largestIndex = leftChildIndex;

}

if (rightChildIndex < size && heap[rightChildIndex].priority > heap[largestIndex].priority) {

largestIndex = rightChildIndex;

}

if (largestIndex != index) {

swap(index, largestIndex);

index = largestIndex;

} else {

break;

}

}

}

// Helper method to swap two elements in the heap

private void swap(int i, int j) {

Node temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

// Method to display the current state of the priority queue

public void display() {

System.out.print("Priority Queue = [");

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

System.out.print(heap[i].value + (i < size - 1 ? ", " : ""));

}

System.out.println("]");

}

// Main method to test the Priority Queue implementation

public static void main(String[] args) {

// Test Case 1

PriorityQueue pq1 = new PriorityQueue();

pq1.enqueue(3, 1);

pq1.enqueue(10, 3);

pq1.enqueue(5, 2);

System.out.println("Dequeued element = " + pq1.dequeue()); // Output: 10 (highest priority)

pq1.display(); // Output: [5, 3]

// Test Case 2

PriorityQueue pq2 = new PriorityQueue();

pq2.enqueue(7, 4);

pq2.enqueue(8, 2);

pq2.enqueue(6, 3);

System.out.println("Dequeued element = " + pq2.dequeue()); // Output: 7 (highest priority)

pq2.display(); // Output: [6, 8]

}

}

**14. Design an algorithm to implement a stack with a getMin() function to return the minimum element in constant time.**

* **Test Case 1**:  
  Input: Push 5, Push 3, Push 7, Get Min  
  Output: Min = 3

**Test Case 2**:  
Input: Push 10, Push 8, Push 6, Push 12, Get Min  
Output: Min = 6

import java.util.Stack;

class MinStack {

private Stack<Integer> mainStack;

private Stack<Integer> minStack;

public MinStack() {

mainStack = new Stack<>();

minStack = new Stack<>();

}

// Push a new element onto the stack

public void push(int x) {

mainStack.push(x);

// If minStack is empty or the current element is smaller than or equal to the top of minStack

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

minStack.push(x);

}

}

// Pop the top element from the stack

public void pop() {

if (mainStack.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

int poppedElement = mainStack.pop();

// If the popped element is the same as the top of minStack, pop it from minStack too

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

minStack.pop();

}

}

// Get the top element of the stack

public int top() {

if (mainStack.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return mainStack.peek();

}

// Get the minimum element in the stack

public int getMin() {

if (minStack.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

return minStack.peek();

}

// Main method to test the MinStack implementation

public static void main(String[] args) {

// Test Case 1

MinStack stack1 = new MinStack();

stack1.push(5);

stack1.push(3);

stack1.push(7);

System.out.println("Min = " + stack1.getMin()); // Output: Min = 3

// Test Case 2

MinStack stack2 = new MinStack();

stack2.push(10);

stack2.push(8);

stack2.push(6);

stack2.push(12);

System.out.println("Min = " + stack2.getMin()); // Output: Min = 6

}

}

**15. Design a Circular Queue with a fixed size, supporting enqueue, dequeue, and isFull/isEmpty operations.**

* **Test Case 1**:  
  Input: Size = 4, Enqueue 1, 2, 3, 4, isFull()  
  Output: True
* **Test Case 2**:  
  Input: Size = 3, Enqueue 5, 6, Dequeue, Enqueue 7, isEmpty()  
  Output: False

class CircularQueue {

private int[] queue;

private int front, rear, size, capacity;

public CircularQueue(int capacity) {

this.capacity = capacity;

queue = new int[capacity];

front = 0;

rear = 0;

size = 0;

}

// Enqueue an element to the queue

public void enqueue(int value) {

if (isFull()) {

throw new IllegalStateException("Queue is full");

}

queue[rear] = value;

rear = (rear + 1) % capacity; // Circular increment

size++;

}

// Dequeue an element from the queue

public int dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

int value = queue[front];

front = (front + 1) % capacity; // Circular increment

size--;

return value;

}

// Check if the queue is full

public boolean isFull() {

return size == capacity;

}

// Check if the queue is empty

public boolean isEmpty() {

return size == 0;

}

// Main method to test the CircularQueue implementation

public static void main(String[] args) {

// Test Case 1

CircularQueue queue1 = new CircularQueue(4);

queue1.enqueue(1);

queue1.enqueue(2);

queue1.enqueue(3);

queue1.enqueue(4);

System.out.println("Is Full? " + queue1.isFull()); // Output: True

// Test Case 2

CircularQueue queue2 = new CircularQueue(3);

queue2.enqueue(5);

queue2.enqueue(6);

queue2.dequeue();

queue2.enqueue(7);

System.out.println("Is Empty? " + queue2.isEmpty()); // Output: False

}

}