**C-DAC Mumbai Date 29/09/2024**

**Subject: Algorithm and Data Structure**

**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

**class Stack {**

**private int[] arr;**

**private int top;**

**private int capacity;**

**Stack(int size) {**

**arr = new int[size];**

**capacity = size;**

**top = -1;**

**}**

**public void push(int x) {**

**if (top == capacity - 1) {**

**System.out.println("Stack Overflow");**

**return;**

**}**

**arr[++top] = x;**

**}**

**public int pop() {**

**if (top == -1) {**

**System.out.println("Stack Underflow");**

**return -1;**

**}**

**return arr[top--];**

**}**

**public boolean isEmpty() {**

**return top == -1;**

**}**

**public void printStack() {**

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

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

**}**

**System.out.println();**

**}**

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

**Stack stack = new Stack(5);**

**// Test Case 1**

**stack.push(5);**

**stack.push(3);**

**stack.push(7);**

**System.out.println("Popped: " + stack.pop()); // Output: 7**

**stack.printStack(); // Output: [5 3]**

**// Test Case 2**

**stack.push(10);**

**stack.push(20);**

**System.out.println("Popped: " + stack.pop()); // Output: 20**

**stack.push(15);**

**stack.printStack(); // Output: [10 15]**

**}**

**}**

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

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

**public class BalancedParentheses {**

**private char[] stack;**

**private int top;**

**BalancedParentheses(int size) {**

**stack = new char[size];**

**top = -1;**

**}**

**public void push(char ch) {**

**stack[++top] = ch;**

**}**

**public char pop() {**

**if (top == -1) {**

**return '\0';**

**}**

**return stack[top--];**

**}**

**public boolean isEmpty() {**

**return top == -1;**

**}**

**public static boolean isBalanced(String expr) {**

**BalancedParentheses stack = new BalancedParentheses(expr.length());**

**for (char ch : expr.toCharArray()) {**

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

**stack.push(ch);**

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

**char popped = stack.pop();**

**if ((ch == ')' && popped != '(') ||**

**(ch == '}' && popped != '{') ||**

**(ch == ']' && popped != '[')) {**

**return false;**

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

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

**public class ReverseString {**

**public static String reverse(String s) {**

**char[] stack = new char[s.length()];**

**int top = -1;**

**for (char ch : s.toCharArray()) {**

**stack[++top] = ch;**

**}**

**StringBuilder reversed = new StringBuilder();**

**while (top >= 0) {**

**reversed.append(stack[top--]);**

**}**

**return reversed.toString();**

**}**

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

**// Test Case 1**

**String input1 = "hello";**

**System.out.println(reverse(input1)); // Output: olleh**

**// Test Case 2**

**String input2 = "world";**

**System.out.println(reverse(input2)); // Output: dlrow**

**}**

**}**

**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

**public class PostfixEvaluation {**

**public static int evaluatePostfix(String expression) {**

**int[] stack = new int[expression.length()];**

**int top = -1;**

**for (String token : expression.split(" ")) {**

**if (Character.isDigit(token.charAt(0))) {**

**stack[++top] = Integer.parseInt(token);**

**} else {**

**int b = stack[top--];**

**int a = stack[top--];**

**switch (token) {**

**case "+" -> stack[++top] = a + b;**

**case "-" -> stack[++top] = a - b;**

**case "\*" -> stack[++top] = a \* b;**

**case "/" -> stack[++top] = a / b;**

**}**

**}**

**}**

**return stack[top];**

**}**

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

**// Test Case 1**

**String expr1 = "5 3 + 2 \*";**

**System.out.println(evaluatePostfix(expr1)); // Output: 16**

**// Test Case 2**

**String expr2 = "4 5 \* 6 /";**

**System.out.println(evaluatePostfix(expr2)); // Output: 3**

**}**

**}**

**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 / +"

**public class InfixToPostfix {**

**private static int precedence(char ch) {**

**return switch (ch) {**

**case '+', '-' -> 1;**

**case '\*', '/' -> 2;**

**default -> -1;**

**};**

**}**

**public static String convert(String expression) {**

**char[] stack = new char[expression.length()];**

**int top = -1;**

**StringBuilder result = new StringBuilder();**

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

**if (Character.isLetterOrDigit(ch)) {**

**result.append(ch).append(' ');**

**} else if (ch == '(') {**

**stack[++top] = ch;**

**} else if (ch == ')') {**

**while (top >= 0 && stack[top] != '(') {**

**result.append(stack[top--]).append(' ');**

**}**

**top--; // Pop '('**

**} else {**

**while (top >= 0 && precedence(ch) <= precedence(stack[top])) {**

**result.append(stack[top--]).append(' ');**

**}**

**stack[++top] = ch;**

**}**

**}**

**while (top >= 0) {**

**result.append(stack[top--]).append(' ');**

**}**

**return result.toString().trim();**

**}**

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

**// Test Case 1**

**String infix1 = "A + B \* C";**

**System.out.println(convert(infix1)); // Output: A B C \* +**

**// Test Case 2**

**String infix2 = "A \* B + C / D";**

**System.out.println(convert(infix2)); // Output: A B \* C D / +**

**}**

**}**

**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

**class Queue {**

**private int[] arr;**

**private int front, rear, size, capacity;**

**Queue(int capacity) {**

**arr = new int[capacity];**

**this.capacity = capacity;**

**front = 0;**

**rear = -1;**

**size = 0;**

**}**

**public void enqueue(int x) {**

**if (size == capacity) {**

**System.out.println("Queue Overflow");**

**return;**

**}**

**rear = (rear + 1) % capacity;**

**arr[rear] = x;**

**size++;**

**}**

**public int dequeue() {**

**if (size == 0) {**

**System.out.println("Queue Underflow");**

**return -1;**

**}**

**int value = arr[front];**

**front = (front + 1) % capacity;**

**size--;**

**return value;**

**}**

**public void printQueue() {**

**int count = 0;**

**for (int i = front; count < size; i = (i + 1) % capacity) {**

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

**count++;**

**}**

**System.out.println();**

**}**

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

**Queue queue = new Queue(5);**

**// Test Case 1**

**queue.enqueue(5);**

**queue.enqueue(10);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 5**

**queue.printQueue(); // Output: [10]**

**// Test Case 2**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 1**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 2**

**queue.printQueue(); // Output: [3]**

**}**

**}**

**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]

**class CircularQueue {**

**private int[] arr;**

**private int front, rear, size, capacity;**

**CircularQueue(int capacity) {**

**arr = new int[capacity];**

**this.capacity = capacity;**

**front = 0;**

**rear = -1;**

**size = 0;**

**}**

**public void enqueue(int x) {**

**if (size == capacity) {**

**System.out.println("Queue Overflow");**

**return;**

**}**

**rear = (rear + 1) % capacity;**

**arr[rear] = x;**

**size++;**

**}**

**public int dequeue() {**

**if (size == 0) {**

**System.out.println("Queue Underflow");**

**return -1;**

**}**

**int value = arr[front];**

**front = (front + 1) % capacity;**

**size--;**

**return value;**

**}**

**public void printQueue() {**

**int count = 0;**

**for (int i = front; count < size; i = (i + 1) % capacity) {**

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

**count++;**

**}**

**System.out.println();**

**}**

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

**CircularQueue queue = new CircularQueue(5);**

**// Test Case 1**

**queue.enqueue(4);**

**queue.enqueue(5);**

**queue.enqueue(6);**

**queue.enqueue(7);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 4**

**queue.enqueue(8);**

**queue.printQueue(); // Output: [8 5 6 7]**

**// Test Case 2**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**queue.enqueue(4);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 1**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 2**

**queue.enqueue(5);**

**queue.printQueue(); // Output: [5 3 4]**

**}**

**}**

**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

**class TwoStackQueue {**

**private int[] stack1;**

**private int[] stack2;**

**private int top1, top2;**

**private int capacity;**

**TwoStackQueue(int size) {**

**stack1 = new int[size];**

**stack2 = new int[size];**

**top1 = -1;**

**top2 = -1;**

**capacity = size;**

**}**

**public void enqueue(int x) {**

**if (top1 == capacity - 1) {**

**System.out.println("Queue Overflow");**

**return;**

**}**

**stack1[++top1] = x;**

**}**

**public int dequeue() {**

**if (top2 == -1) {**

**while (top1 != -1) {**

**stack2[++top2] = stack1[top1--];**

**}**

**}**

**if (top2 == -1) {**

**System.out.println("Queue Underflow");**

**return -1;**

**}**

**return stack2[top2--];**

**}**

**public void printQueue() {**

**if (top2 != -1) {**

**for (int i = top2; i >= 0; i--) {**

**System.out.print(stack2[i] + " ");**

**}**

**}**

**for (int i = 0; i <= top1; i++) {**

**System.out.print(stack1[i] + " ");**

**}**

**System.out.println();**

**}**

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

**TwoStackQueue queue = new TwoStackQueue(5);**

**// Test Case 1**

**queue.enqueue(3);**

**queue.enqueue(7);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 3**

**queue.printQueue(); // Output: [7]**

**// Test Case 2**

**queue.enqueue(10);**

**queue.enqueue(20);**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 7**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 10**

**queue.printQueue(); // Output: []**

**}**

**}**

**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

**class MinHeap {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**MinHeap(int capacity) {**

**this.capacity = capacity;**

**heap = new int[capacity];**

**size = 0;**

**}**

**private int parent(int i) { return (i - 1) / 2; }**

**private int left(int i) { return 2 \* i + 1; }**

**private int right(int i) { return 2 \* i + 2; }**

**public void insert(int x) {**

**if (size == capacity) {**

**System.out.println("Heap Overflow");**

**return;**

**}**

**heap[size] = x;**

**int i = size;**

**size++;**

**while (i != 0 && heap[parent(i)] > heap[i]) {**

**int temp = heap[i];**

**heap[i] = heap[parent(i)];**

**heap[parent(i)] = temp;**

**i = parent(i);**

**}**

**}**

**public int extractMin() {**

**if (size == 0) return Integer.MAX\_VALUE;**

**if (size == 1) {**

**size--;**

**return heap[0];**

**}**

**int root = heap[0];**

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

**size--;**

**minHeapify(0);**

**return root;**

**}**

**private void minHeapify(int i) {**

**int l = left(i);**

**int r = right(i);**

**int smallest = i;**

**if (l < size && heap[l] < heap[smallest]) smallest = l;**

**if (r < size && heap[r] < heap[smallest]) smallest = r;**

**if (smallest != i) {**

**int temp = heap[i];**

**heap[i] = heap[smallest];**

**heap[smallest] = temp;**

**minHeapify(smallest);**

**}**

**}**

**public void printHeap() {**

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

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

**}**

**System.out.println();**

**}**

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

**MinHeap minHeap = new MinHeap(10);**

**// Test Case 1**

**minHeap.insert(10);**

**minHeap.insert(15);**

**minHeap.insert(20);**

**minHeap.insert(17);**

**System.out.println("Extracted Min: " + minHeap.extractMin()); // Output: 10**

**minHeap.printHeap(); // Output: [15 17 20]**

**// Test Case 2**

**minHeap.insert(30);**

**minHeap.insert(40);**

**minHeap.insert(20);**

**minHeap.insert(50);**

**System.out.println("Extracted Min: " + minHeap.extractMin()); // Output: 20**

**minHeap.printHeap(); // Output: [30 40 50]**

**}**

**}**

**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

**class MaxHeap {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**MaxHeap(int capacity) {**

**this.capacity = capacity;**

**heap = new int[capacity];**

**size = 0;**

**}**

**private int parent(int i) { return (i - 1) / 2; }**

**private int left(int i) { return 2 \* i + 1; }**

**private int right(int i) { return 2 \* i + 2; }**

**public void insert(int x) {**

**if (size == capacity) {**

**System.out.println("Heap Overflow");**

**return;**

**}**

**heap[size] = x;**

**int i = size;**

**size++;**

**while (i != 0 && heap[parent(i)] < heap[i]) {**

**int temp = heap[i];**

**heap[i] = heap[parent(i)];**

**heap[parent(i)] = temp;**

**i = parent(i);**

**}**

**}**

**public int extractMax() {**

**if (size == 0) return Integer.MIN\_VALUE;**

**if (size == 1) {**

**size--;**

**return heap[0];**

**}**

**int root = heap[0];**

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

**size--;**

**maxHeapify(0);**

**return root;**

**}**

**private void maxHeapify(int i) {**

**int l = left(i);**

**int r = right(i);**

**int largest = i;**

**if (l < size && heap[l] > heap[largest]) largest = l;**

**if (r < size && heap[r] > heap[largest]) largest = r;**

**if (largest != i) {**

**int temp = heap[i];**

**heap[i] = heap[largest];**

**heap[largest] = temp;**

**maxHeapify(largest);**

**}**

**}**

**public void printHeap() {**

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

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

**}**

**System.out.println();**

**}**

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

**MaxHeap maxHeap = new MaxHeap(10);**

**// Test Case 1**

**maxHeap.insert(12);**

**maxHeap.insert(7);**

**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]

**class HeapSort {**

**public void heapSort(int[] arr) {**

**int n = arr.length;**

**// Build heap (rearrange array)**

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

**heapify(arr, n, i);**

**}**

**// One by one extract an element from heap**

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

**// Move current root to end**

**int temp = arr[0];**

**arr[0] = arr[i];**

**arr[i] = temp;**

**// Call max heapify on the reduced heap**

**heapify(arr, i, 0);**

**}**

**}**

**void heapify(int[] arr, int n, int i) {**

**int largest = i; // Initialize largest as root**

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

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

**// If left child is larger than root**

**if (left < n && arr[left] > arr[largest]) {**

**largest = left;**

**}**

**// If right child is larger than largest**

**if (right < n && arr[right] > arr[largest]) {**

**largest = right;**

**}**

**// If largest is not root**

**if (largest != i) {**

**int swap = arr[i];**

**arr[i] = arr[largest];**

**arr[largest] = swap;**

**// Recursively heapify the affected sub-tree**

**heapify(arr, n, largest);**

**}**

**}**

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

**HeapSort heapSort = new HeapSort();**

**int[] arr1 = {5, 1, 12, 3, 9};**

**int[] arr2 = {20, 15, 8, 10};**

**heapSort.heapSort(arr1);**

**System.out.println("Sorted array: ");**

**for (int num : arr1) {**

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

**} // Output: [1 3 5 9 12]**

**heapSort.heapSort(arr2);**

**System.out.println("\nSorted array: ");**

**for (int num : arr2) {**

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

**} // Output: [8 10 15 20]**

**}**

**}**

**12. Find the kth largest element in a stream of numbers using a heap.**

* **Test Case 1**:  
  Input: Stream = [3, 10, 5, 20, 15], k = 3  
  Output: 10
* **Test Case 2**:  
  Input: Stream = [7, 4, 8, 2, 9], k = 2  
  Output: 8

**class KthLargestElement {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**KthLargestElement(int capacity) {**

**this.capacity = capacity;**

**heap = new int[capacity];**

**size = 0;**

**}**

**public void insert(int x) {**

**if (size < capacity) {**

**heap[size] = x;**

**int i = size++;**

**while (i != 0 && heap[(i - 1) / 2] > heap[i]) {**

**int temp = heap[i];**

**heap[i] = heap[(i - 1) / 2];**

**heap[(i - 1) / 2] = temp;**

**i = (i - 1) / 2;**

**}**

**} else if (x > heap[0]) {**

**heap[0] = x;**

**heapify(0);**

**}**

**}**

**public void heapify(int i) {**

**int smallest = i;**

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

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

**if (left < size && heap[left] < heap[smallest]) {**

**smallest = left;**

**}**

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

**smallest = right;**

**}**

**if (smallest != i) {**

**int temp = heap[i];**

**heap[i] = heap[smallest];**

**heap[smallest] = temp;**

**heapify(smallest);**

**}**

**}**

**public int getKthLargest() {**

**return heap[0];**

**}**

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

**KthLargestElement stream1 = new KthLargestElement(3);**

**// Test Case 1**

**int[] stream = {3, 10, 5, 20, 15};**

**for (int num : stream) {**

**stream1.insert(num);**

**}**

**System.out.println("3rd Largest: " + stream1.getKthLargest()); // Output: 10**

**// Test Case 2**

**KthLargestElement stream2 = new KthLargestElement(2);**

**int[] stream2Arr = {7, 4, 8, 2, 9};**

**for (int num : stream2Arr) {**

**stream2.insert(num);**

**}**

**System.out.println("2nd Largest: " + stream2.getKthLargest()); // Output: 8**

**}**

**}**

**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]

**class PriorityQueue {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**PriorityQueue(int capacity) {**

**this.capacity = capacity;**

**heap = new int[capacity];**

**size = 0;**

**}**

**public void insert(int x) {**

**if (size == capacity) {**

**System.out.println("Queue Overflow");**

**return;**

**}**

**heap[size] = x;**

**int i = size;**

**size++;**

**while (i != 0 && heap[(i - 1) / 2] < heap[i]) {**

**int temp = heap[i];**

**heap[i] = heap[(i - 1) / 2];**

**heap[(i - 1) / 2] = temp;**

**i = (i - 1) / 2;**

**}**

**}**

**public int extractMax() {**

**if (size == 0) {**

**System.out.println("Queue Underflow");**

**return Integer.MIN\_VALUE;**

**}**

**if (size == 1) {**

**size--;**

**return heap[0];**

**}**

**int root = heap[0];**

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

**size--;**

**maxHeapify(0);**

**return root;**

**}**

**private void maxHeapify(int i) {**

**int largest = i;**

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

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

**if (left < size && heap[left] > heap[largest]) largest = left;**

**if (right < size && heap[right] > heap[largest]) largest = right;**

**if (largest != i) {**

**int swap = heap[i];**

**heap[i] = heap[largest];**

**heap[largest] = swap;**

**maxHeapify(largest);**

**}**

**}**

**public void printQueue() {**

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

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

**}**

**System.out.println();**

**}**

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

**PriorityQueue pq = new PriorityQueue(10);**

**// Test Case 1**

**pq.insert(3);**

**pq.insert(10);**

**pq.insert(5);**

**System.out.println("Extracted Max: " + pq.extractMax()); // Output: 10**

**pq.printQueue(); // Output: [5 3]**

**// Test Case 2**

**pq.insert(7);**

**pq.insert(8);**

**pq.insert(6);**

**System.out.println("Extracted Max: " + pq.extractMax()); // Output: 8**

**pq.printQueue(); // Output: [7 6]**

**}**

**}**

**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

**class MinStack {**

**private int[] stack;**

**private int[] minStack;**

**private int top, minTop;**

**MinStack(int size) {**

**stack = new int[size];**

**minStack = new int[size];**

**top = -1;**

**minTop = -1;**

**}**

**public void push(int x) {**

**stack[++top] = x;**

**if (minTop == -1 || x <= minStack[minTop]) {**

**minStack[++minTop] = x;**

**}**

**}**

**public int pop() {**

**int popped = stack[top--];**

**if (popped == minStack[minTop]) {**

**minTop--;**

**}**

**return popped;**

**}**

**public int getMin() {**

**return minStack[minTop];**

**}**

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

**MinStack minStack = new MinStack(5);**

**// Test Case 1**

**minStack.push(5);**

**minStack.push(3);**

**minStack.push(7);**

**System.out.println("Min: " + minStack.getMin()); // Output: 3**

**// Test Case 2**

**minStack.push(10);**

**minStack.push(8);**

**minStack.push(6);**

**System.out.println("Min: " + minStack.getMin()); // Output: 3**

**minStack.pop();**

**System.out.println("Min after pop: " + minStack.getMin()); // Output: 3**

**}**

**}**

**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 CircularQueueFixed {**

**private int[] arr; // Array to hold the queue elements**

**private int front; // Index of the front element**

**private int rear; // Index of the last element**

**private int size; // Current number of elements**

**private int capacity; // Maximum size of the queue**

**// Constructor to initialize the queue**

**CircularQueueFixed(int capacity) {**

**arr = new int[capacity];**

**this.capacity = capacity;**

**front = 0;**

**rear = -1;**

**size = 0;**

**}**

**// Method to add an element to the queue**

**public void enqueue(int x) {**

**if (isFull()) {**

**System.out.println("Queue Overflow");**

**return;**

**}**

**rear = (rear + 1) % capacity; // Wrap around if needed**

**arr[rear] = x; // Add the new element**

**size++;**

**}**

**// Method to remove an element from the queue**

**public int dequeue() {**

**if (isEmpty()) {**

**System.out.println("Queue Underflow");**

**return -1;**

**}**

**int value = arr[front]; // Get the front element**

**front = (front + 1) % capacity; // Wrap around if needed**

**size--;**

**return value; // Return the dequeued element**

**}**

**// Method to check if the queue is full**

**public boolean isFull() {**

**return size == capacity;**

**}**

**// Method to check if the queue is empty**

**public boolean isEmpty() {**

**return size == 0;**

**}**

**// Method to print the elements of the queue**

**public void printQueue() {**

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

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

**}**

**System.out.println();**

**}**

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

**CircularQueueFixed queue = new CircularQueueFixed(4);**

**// Test Case 1**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**queue.enqueue(4);**

**System.out.println("Is Full: " + queue.isFull()); // Output: true**

**queue.printQueue(); // Output: 1 2 3 4**

**// Test Case 2**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 1**

**System.out.println("Is Empty: " + queue.isEmpty()); // Output: false**

**queue.printQueue(); // Output: 2 3 4**

**// Adding one more element to test wrap around**

**queue.enqueue(5);**

**queue.printQueue(); // Output: 2 3 4 5**

**// Check if full**

**System.out.println("Is Full: " + queue.isFull()); // Output: true**

**// Dequeue all elements**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 2**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 3**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 4**

**System.out.println("Dequeued: " + queue.dequeue()); // Output: 5**

**System.out.println("Is Empty: " + queue.isEmpty()); // Output: true**

**}**

**}**