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

**Subject: Algorithm and Data Structure**

**Assignment 3**

**Name:** Yash Bandu Dhole

**Center:** Juhu.

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

**Code:**

package problem1;

class Stack {

private int[] stack;

private int top;

private int capacity;

public Stack(int size) {

stack = new int[size];

top = -1;

capacity = size;

}

public void push(int value) {

if (top == capacity - 1) {

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

return;

}

stack[++top] = value;

}

public int pop() {

if (top == -1) {

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

return -1;

}

return stack[top--];

}

public void display() {

if (top == -1) {

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

return;

}

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

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

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

}

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

}

public int getTop() {

return top;

}

}

public class Main {

public static void main(String[] args) {

Stack stack = new Stack(10);

stack.push(5);

stack.push(3);

stack.push(7);

stack.display();

int popped1 = stack.pop();

System.***out***.println("Popped element = " + popped1);

stack.display();

stack.push(10);

stack.push(20);

int popped2 = stack.pop();

System.***out***.println("Popped element = " + popped2);

stack.push(15);

stack.display();

}

}

**O/p:**

Stack = 5 3 7

Popped element = 7

Stack = 5 3

Popped element = 20

Stack = 5 3 10 15

**Time complexity:** O(1)

**Space complexity:** O(n)

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

**Code:**

package problem2;

import java.util.Stack;

public class balancedParanthesis{

public static boolean isBalancedIterative(String expr) {

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

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

char current = expr.charAt(i);

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

stack.push(current);

}

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

if (stack.isEmpty() || !*isMatchingPair*(stack.pop(), current)) {

return false;

}

}

}

return stack.isEmpty();

}

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

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

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

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

}

public static void main(String[] args) {

String expr1 = "({[()]})";

System.***out***.println(*isBalancedIterative*(expr1) ? "Balanced" : "Not Balanced");

String expr2 = "([)]";

System.***out***.println(*isBalancedIterative*(expr2) ? "Balanced" : "Not Balanced");

}

}

**O/p:**

Balanced

Not Balanced

**Time complexity:** O(n)

**Space complexity:** O(n)

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

**Code:**

package problem3;

public class StringReverser {

public static String reverseStringIterative(String str) {

int n = str.length();

char[] stack = new char[n];

int top = -1;

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

stack[++top] = str.charAt(i);

}

StringBuilder reversed = new StringBuilder();

while (top >= 0) {

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

}

return reversed.toString();

}

public static void main(String[] args) {

String input1 = "hello";

String input2 = "world";

System.***out***.println("Iterative Output for 'hello': " + *reverseStringIterative*(input1)); // olleh

System.***out***.println("Iterative Output for 'world': " + *reverseStringIterative*(input2)); // dlrow

}

}

**O/p:**

Iterative Output for 'hello': olleh

Iterative Output for 'world': dlrow

**Time complexity:** O(n)

**Space complexity:**O(n)

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

**Code:**

package problem4;

public class postfixEvaluator {

public static int evaluatePostfix(String expression) {

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

int n = tokens.length;

int[] stack = new int[n];

int top = -1;

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

String token = tokens[i];

if (*isNumber*(token)) {

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

} else {

int operand2 = stack[top--];

int operand1 = stack[top--];

int result = *performOperation*(operand1, operand2, token);

stack[++top] = result;

}

}

return stack[top];

}

private static boolean isNumber(String token) {

try {

Integer.*parseInt*(token);

return true;

} catch (NumberFormatException e) {

return false;

}

}

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

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 input1 = "5 3 + 2 \*";

String input2 = "4 5 \* 6 /";

System.***out***.println(*evaluatePostfix*(input1)); // 16

System.***out***.println(*evaluatePostfix*(input2)); // 3

}

}

**O/p:**

16

3

**Time complexity:**  O(n)

**Space complexity:** O(n)

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

**Code:**

package problem5;

public class InfixToPostfix {

public static String infixToPostfix(String expression) {

char[] exp = expression.toCharArray();

int n = exp.length;

char[] stack = new char[n];

int top = -1;

StringBuilder result = new StringBuilder();

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

char token = exp[i];

if (token == ' ') {

continue; // Skip spaces

}

if (*isOperand*(token)) {

result.append(token).append(" ");

} else if (token == '(') {

stack[++top] = token;

} else if (token == ')') {

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

result.append(stack[top--]).append(" ");

}

top--; // Pop '('

} else {

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

result.append(stack[top--]).append(" ");

}

stack[++top] = token;

}

}

while (top >= 0) {

result.append(stack[top--]).append(" ");

}

return result.toString().trim();

}

private static boolean isOperand(char token) {

return Character.*isLetterOrDigit*(token);

}

private static int precedence(char operator) {

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '(':

return 0;

default:

return -1;

}

}

public static void main(String[] args) {

String input1 = "A + B \* C";

String input2 = "A \* B + C / D";

System.***out***.println(*infixToPostfix*(input1)); // A B C \* +

System.***out***.println(*infixToPostfix*(input2)); // A B \* C D / +

}

}

**O/p:**

A B C \* +

A B \* C D / +

**Time complexity:** O(n)

**Space complexity:** O(n)

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

**Code:**

package problem6;

public class ArrayQueue {

private int[] queue;

private int front, rear, size, capacity;

public ArrayQueue(int capacity) {

this.capacity = capacity;

this.queue = new int[capacity];

this.front = 0;

this.rear = -1;

this.size = 0;

}

public void enqueue(int item) {

if (size == capacity) {

throw new IllegalStateException("Queue is full");

}

rear = (rear + 1) % capacity;

queue[rear] = item;

size++;

}

public int dequeue() {

if (size == 0) {

throw new IllegalStateException("Queue is empty");

}

int item = queue[front];

front = (front + 1) % capacity;

size--;

return item;

}

public void printQueue() {

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

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

}

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

}

public static void main(String[] args) {

ArrayQueue queue1 = new ArrayQueue(5);

queue1.enqueue(5);

queue1.enqueue(10);

System.***out***.println("Dequeued element = " + queue1.dequeue());

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

queue1.printQueue();

ArrayQueue queue2 = new ArrayQueue(5);

queue2.enqueue(1);

queue2.enqueue(2);

queue2.enqueue(3);

System.***out***.println("Dequeued element = " + queue2.dequeue());

System.***out***.println("Dequeued element = " + queue2.dequeue());

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

queue2.printQueue();

}

}

**O/p:**

Dequeued element = 5

Queue = 10

Dequeued element = 1

Dequeued element = 2

Queue = 3

**Time complexity:**

EnQue: O(1)

DeQue:O(1)

PrintQue: O(n)

**Space complexity:** O(n)

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

**Code:**

package problem7;

public class CircularQueue {

private int[] queue;

private int front, rear, size, capacity;

public CircularQueue(int capacity) {

this.capacity = capacity;

this.queue = new int[capacity];

this.front = 0;

this.rear = -1;

this.size = 0;

}

public void enqueue(int item) {

if (size == capacity) {

throw new IllegalStateException("Queue is full");

}

rear = (rear + 1) % capacity;

queue[rear] = item;

size++;

}

public int dequeue() {

if (size == 0) {

throw new IllegalStateException("Queue is empty");

}

int item = queue[front];

front = (front + 1) % capacity;

size--;

return item;

}

public void printQueue() {

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

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

}

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

}

public static void main(String[] args) {

CircularQueue queue1 = new CircularQueue(4);

queue1.enqueue(4);

queue1.enqueue(5);

queue1.enqueue(6);

queue1.enqueue(7);

queue1.dequeue();

queue1.enqueue(8);

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

queue1.printQueue();

CircularQueue queue2 = new CircularQueue(4);

queue2.enqueue(1);

queue2.enqueue(2);

queue2.enqueue(3);

queue2.enqueue(4);

queue2.dequeue();

queue2.dequeue();

queue2.enqueue(5);

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

queue2.printQueue();

}

}

**O/p:**

Queue = 5 6 7 8

Queue = 3 4 5

**Time complexity:**

EnQue: O(1)

DeQue:O(1)

PrintQue: O(n)

**Space complexity:**

O(n)

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

**Code:**

package problem8;

public class QueueWith2Stacks {

private int[] stack1, stack2;

private int top1, top2, size1, size2, capacity;

public QueueWith2Stacks(int capacity) {

this.capacity = capacity;

this.stack1 = new int[capacity];

this.stack2 = new int[capacity];

this.top1 = -1;

this.top2 = -1;

this.size1 = 0;

this.size2 = 0;

}

public void enqueue(int item) {

if (size1 == capacity) {

throw new IllegalStateException("Queue is full");

}

stack1[++top1] = item;

size1++;

}

public int dequeue() {

if (size2 == 0) {

while (size1 > 0) {

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

size1--;

size2++;

}

}

if (size2 == 0) {

throw new IllegalStateException("Queue is empty");

}

size2--;

return stack2[top2--];

}

public void printQueue() {

if (size2 > 0) {

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

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

}

}

if (size1 > 0) {

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

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

}

}

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

}

public static void main(String[] args) {

QueueWith2Stacks queue1 = new QueueWith2Stacks(5);

queue1.enqueue(3);

queue1.enqueue(7);

System.***out***.println("Dequeued element = " + queue1.dequeue());

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

queue1.printQueue();

QueueWith2Stacks queue2 = new QueueWith2Stacks(5);

queue2.enqueue(10);

queue2.enqueue(20);

System.***out***.println("Dequeued element = " + queue2.dequeue());

System.***out***.println("Dequeued element = " + queue2.dequeue());

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

queue2.printQueue();

}

}

**O/p:**

Dequeued element = 3

Queue = 7

Dequeued element = 10

Dequeued element = 20

Queue =

**Time complexity:**

EnQue:O(1)

DeQue:O(n)

PrintQue:O(n)

**Space complexity:** O(n)

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

**Code:**

package problem9;

public class MInHeap {

private int[] heap;

private int size;

private int capacity;

public MInHeap(int capacity) {

this.capacity = capacity;

this.heap = new int[capacity];

this.size = 0;

}

public void insert(int value) {

if (size == capacity) {

throw new IllegalStateException("Heap is full");

}

heap[size] = value;

int current = size;

size++;

while (current > 0 && heap[current] < heap[parent(current)]) {

swap(current, parent(current));

current = parent(current);

}

}

public int extractMin() {

if (size == 0) {

throw new IllegalStateException("Heap is empty");

}

int min = heap[0];

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

size--;

heapify(0);

return min;

}

private void heapify(int index) {

int smallest = index;

int left = leftChild(index);

int right = rightChild(index);

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

smallest = left;

}

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

smallest = right;

}

if (smallest != index) {

swap(index, smallest);

heapify(smallest);

}

}

private int parent(int index) {

return (index - 1) / 2;

}

private int leftChild(int index) {

return (2 \* index) + 1;

}

private int rightChild(int index) {

return (2 \* index) + 2;

}

private void swap(int index1, int index2) {

int temp = heap[index1];

heap[index1] = heap[index2];

heap[index2] = temp;

}

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 heap1 = new MInHeap(10);

heap1.insert(10);

heap1.insert(15);

heap1.insert(20);

heap1.insert(17);

System.***out***.println("Extracted Min = " + heap1.extractMin());

System.***out***.print("Min-Heap = ");

heap1.printHeap();

MInHeap heap2 = new MInHeap(10);

heap2.insert(30);

heap2.insert(40);

heap2.insert(20);

heap2.insert(50);

System.***out***.println("Extracted Min = " + heap2.extractMin());

System.***out***.print("Min-Heap = ");

heap2.printHeap();

}

}

**O/p:**

Extracted Min = 10

Min-Heap = 15 17 20

Extracted Min = 20

Min-Heap = 30 40 50

**Time complexity:**

Insert: log(n)

Extract: log(n)

Heapify: log(n)

**Space complexity:** O(n)

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

**Code:**

package problem10;

public class MaxHeap {

private int[] heap;

private int size;

private int capacity;

public MaxHeap(int capacity) {

this.capacity = capacity;

this.heap = new int[capacity];

this.size = 0;

}

public void insert(int value) {

if (size == capacity) {

throw new IllegalStateException("Heap is full");

}

heap[size] = value;

int current = size;

size++;

while (current > 0 && heap[current] > heap[parent(current)]) {

swap(current, parent(current));

current = parent(current);

}

}

public int extractMax() {

if (size == 0) {

throw new IllegalStateException("Heap is empty");

}

int max = heap[0];

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

size--;

heapify(0);

return max;

}

private void heapify(int index) {

int largest = index;

int left = leftChild(index);

int right = rightChild(index);

if (left < size && heap[left] > heap[largest]) {

largest = left;

}

if (right < size && heap[right] > heap[largest]) {

largest = right;

}

if (largest != index) {

swap(index, largest);

heapify(largest);

}

}

private int parent(int index) {

return (index - 1) / 2;

}

private int leftChild(int index) {

return (2 \* index) + 1;

}

private int rightChild(int index) {

return (2 \* index) + 2;

}

private void swap(int index1, int index2) {

int temp = heap[index1];

heap[index1] = heap[index2];

heap[index2] = temp;

}

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 heap1 = new MaxHeap(5);

heap1.insert(12);

heap1.insert(7);

heap1.insert(15);

heap1.insert(5);

System.***out***.println("Extracted Max = " + heap1.extractMax());

System.***out***.print("Max-Heap = ");

heap1.printHeap();

MaxHeap heap2 = new MaxHeap(5);

heap2.insert(8);

heap2.insert(20);

heap2.insert(10);

heap2.insert(3);

System.***out***.println("Extracted Max = " + heap2.extractMax());

System.***out***.print("Max-Heap = ");

heap2.printHeap();

}

}

**O/p:**

Extracted Max = 15

Max-Heap = 12 7 5

Extracted Max = 20

Max-Heap = 10 8 3

**Time complexity:**

Insert: log (n)

Extract : log(n)

Heapify: log(n)

**Space complexity:** O(n)

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

**Code:**

package problem11;

public class HeapSort {

public void heapSort(int[] arr) {

int n = arr.length;

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

heapify(arr, n, i);

}

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

swap(arr, 0, i);

heapify(arr, i, 0);

}

}

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

}

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

largest = right;

}

if (largest != i) {

swap(arr, i, largest);

heapify(arr, n, largest);

}

}

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

HeapSort hs = new HeapSort();

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

hs.heapSort(arr1);

System.***out***.print("Sorted Array 1: ");

for (int num : arr1) {

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

}

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

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

hs.heapSort(arr2);

System.***out***.print("Sorted Array 2: ");

for (int num : arr2) {

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

}

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

}

}

**O/p:**

Sorted Array 1: 1 3 5 9 12

Sorted Array 2: 8 10 15 20

**Time complexity:** O( n log n )

**Space complexity:** O(1)

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

**Code:**

package problem12;

public class KthLargestInStream {

private int[] heap;

private int size;

private int capacity;

public KthLargestInStream(int k) {

this.capacity = k;

this.heap = new int[k];

this.size = 0;

}

public void add(int value) {

if (size < capacity) {

heap[size] = value;

size++;

heapifyUp(size - 1);

} else if (value > heap[0]) {

heap[0] = value;

heapifyDown(0);

}

}

public int getKthLargest() {

if (size == 0) {

throw new IllegalStateException("No elements in the heap");

}

return heap[0];

}

private void heapifyUp(int index) {

while (index > 0 && heap[index] < heap[parent(index)]) {

swap(index, parent(index));

index = parent(index);

}

}

private void heapifyDown(int index) {

int smallest = index;

int left = leftChild(index);

int right = rightChild(index);

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

smallest = left;

}

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

smallest = right;

}

if (smallest != index) {

swap(index, smallest);

heapifyDown(smallest);

}

}

private int parent(int index) {

return (index - 1) / 2;

}

private int leftChild(int index) {

return 2 \* index + 1;

}

private int rightChild(int index) {

return 2 \* index + 2;

}

private void swap(int i, int j) {

int temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

public static void main(String[] args) {

// Test Case 1

int[] stream1 = {3, 10, 5, 20, 15};

int k1 = 3;

KthLargestInStream kth1 = new KthLargestInStream(k1);

for (int num : stream1) {

kth1.add(num);

}

System.***out***.println(" Case 1 Output: " + kth1.getKthLargest()); // Output: 10

// Test Case 2

int[] stream2 = {7, 4, 8, 2, 9};

int k2 = 2;

KthLargestInStream kth2 = new KthLargestInStream(k2);

for (int num : stream2) {

kth2.add(num);

}

System.***out***.println(" Case 2 Output: " + kth2.getKthLargest()); // Output: 8

}

}

**O/p:**

Case 1 Output: 10

Case 2 Output: 8

**Time complexity:** O( n log k)

**Space complexity:** O( k )

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

**Code:**

package problem13;

class PriorityQueue {

private static class Node {

int value;

int priority;

Node(int value, int priority) {

this.value = value;

this.priority = priority;

}

}

private Node[] heap;

private int size;

private int capacity;

public PriorityQueue(int capacity) {

this.capacity = capacity;

this.heap = new Node[capacity];

this.size = 0;

}

public void enqueue(int value, int priority) {

if (size == capacity) {

throw new IllegalStateException("Queue is full");

}

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

size++;

heapifyUp(size - 1);

}

public Node dequeue() {

if (size == 0) {

throw new IllegalStateException("Queue is empty");

}

Node root = heap[0];

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

size--;

heapifyDown(0);

return root;

}

private void heapifyUp(int index) {

while (index > 0 && heap[index].priority > heap[parent(index)].priority) {

swap(index, parent(index));

index = parent(index);

}

}

private void heapifyDown(int index) {

int largest = index;

int left = leftChild(index);

int right = rightChild(index);

if (left < size && heap[left].priority > heap[largest].priority) {

largest = left;

}

if (right < size && heap[right].priority > heap[largest].priority) {

largest = right;

}

if (largest != index) {

swap(index, largest);

heapifyDown(largest);

}

}

private int parent(int index) {

return (index - 1) / 2;

}

private int leftChild(int index) {

return 2 \* index + 1;

}

private int rightChild(int index) {

return 2 \* index + 2;

}

private void swap(int i, int j) {

Node temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

public void printQueue() {

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

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

}

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

}

public static void main(String[] args) {

// Test Case 1

PriorityQueue pq1 = new PriorityQueue(10);

pq1.enqueue(3, 1);

pq1.enqueue(10, 3);

pq1.enqueue(5, 2);

Node dequeued1 = pq1.dequeue();

System.***out***.println("Dequeued element = " + dequeued1.value); // Output: 10

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

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

// Test Case 2

PriorityQueue pq2 = new PriorityQueue(10);

pq2.enqueue(7, 4);

pq2.enqueue(8, 2);

pq2.enqueue(6, 3);

Node dequeued2 = pq2.dequeue();

System.***out***.println("Dequeued element = " + dequeued2.value); // Output: 7

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

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

}

}

**O/p:**

Dequeued element = 10

Priority Queue = 5 3

Dequeued element = 7

Priority Queue = 6 8

**Time complexity:** O( log n)

**Space complexity:** O(n)

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

**Code:**

package problem14;

class MinStack {

private int[] stack;

private int[] minStack;

private int top;

private int minTop;

private int capacity;

public MinStack(int capacity) {

this.capacity = capacity;

stack = new int[capacity];

minStack = new int[capacity];

top = -1;

minTop = -1;

}

public void push(int x) {

if (top == capacity - 1) {

throw new IllegalStateException("Stack is full");

}

stack[++top] = x;

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

minStack[++minTop] = x;

}

}

public int pop() {

if (top == -1) {

throw new IllegalStateException("Stack is empty");

}

int popped = stack[top--];

if (popped == minStack[minTop]) {

minTop--;

}

return popped;

}

public int getMin() {

if (minTop == -1) {

throw new IllegalStateException("Stack is empty");

}

return minStack[minTop];

}

public void printStack() {

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

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

}

System.*out*.println();

}

public static void main(String[] args) {

// Test Case 1

MinStack ms1 = new MinStack(10);

ms1.push(5);

ms1.push(3);

ms1.push(7);

System.*out*.println("Min = " + ms1.getMin()); // Min = 3

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

ms1.printStack();

MinStack ms2 = new MinStack(10);

ms2.push(10);

ms2.push(8);

ms2.push(6);

ms2.push(12);

System.*out*.println("Min = " + ms2.getMin());

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

ms2.printStack();

}

}

**O/p:**

Min = 3

Stack = 5 3 7

Min = 6

Stack = 10 8 6 12

**Time complexity:** O(1)

**Space complexity:** O(n)

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

**Code:**

package problem15;

class CircularQueue {

private int[] queue;

private int front;

private int rear;

private int size;

public CircularQueue(int capacity) {

queue = new int[capacity];

front = -1;

rear = -1;

size = capacity;

}

public boolean isFull() {

return (rear + 1) % size == front;

}

public boolean isEmpty() {

return front == -1;

}

public void enqueue(int value) {

if (isFull()) {

throw new IllegalStateException("Queue is full");

}

if (isEmpty()) {

front = 0;

}

rear = (rear + 1) % size;

queue[rear] = value;

}

public int dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

int dequeued = queue[front];

if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % size;

}

return dequeued;

}

public void printQueue() {

if (isEmpty()) {

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

return;

}

int i = front;

while (true) {

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

if (i == rear) break;

i = (i + 1) % size;

}

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

}

public static void main(String[] args) {

CircularQueue cq1 = new CircularQueue(4);

cq1.enqueue(1);

cq1.enqueue(2);

cq1.enqueue(3);

cq1.enqueue(4);

System.***out***.println("isFull = " + cq1.isFull());

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

cq1.printQueue();

CircularQueue cq2 = new CircularQueue(3);

cq2.enqueue(5);

cq2.enqueue(6);

cq2.dequeue();

cq2.enqueue(7);

System.***out***.println("isEmpty = " + cq2.isEmpty()); //False

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

cq2.printQueue();

}

}

**O/p:**

isFull = true

Queue = 1 2 3 4

isEmpty = false

Queue = 6 7

**Time complexity:** O(1)

**Space complexity:** O(n)