**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

Solution—

**package** assign3;

**class** StackApp{

**static** **final** **int** ***MAX***=10;

**int** a[]=**new** **int**[***MAX***];

**int** top;

StackApp(){

top=-1;

}

**boolean** push(**int** d) {

**if**(top>=***MAX***-1)

{

System.***out***.println("stack overflow");

**return** **false**;

}

a[++top]=d;

**return** **true**;

}

**int** pop() {

**if**(top<0) {

System.***out***.println("stack underflow");

**return** -1;

}

**return** a[top--];

}

**void** display() {

**if**(top<0)

{

System.***out***.println("Empty stack");

}

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

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

**int** r=a[i];

**if**(i==top)

System.***out***.print(r);

**else**

System.***out***.print(r+",");

}

System.***out***.print("],");

}

}

**public** **class** A3\_Q1 {

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

StackApp s=**new** StackApp();

s.push(10);

s.push(20);

**int** p=s.pop();

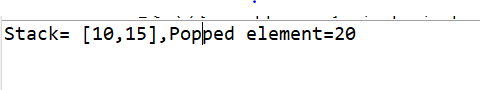
s.push(15);

s.display();

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

}

}



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

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

Solution—

**package** assign3;

**import** java.util.Scanner;

**class** balance{

**static** **final** **int** ***MAX***=10;

**char** a[]=**new** **char**[***MAX***];

**int** top;

**static** **int** *count*=0;

balance(){

top=-1;

}

**boolean** isEmpty() {

**return** (top<0);

}

**boolean** isFull() {

**if**(top>=***MAX***-1)

**return** **true**;

**return** **false**;

}

**int** push(**char** d) {

**if**(isFull())

{

System.***out***.println("stack overflow");

**return** 0;

}

**else** **if**(d== '(' || d=='{' || d=='[' )

{

a[++top]=d;

*count*++;

}

**else**

{

**if**((peek()=='(' && d==')') || (peek()=='{' && d=='}') || (peek()=='[' && d==']') ) {

pop();

*count*--;

}

}

**return** *count*;

}

**char** pop() {

**if**(top<0) {

System.***out***.println("stack underflow");

**return** '0';

}

**return** a[top--];

}

**char** peek() {

**return** top<0?'0':a[top];

}

}

**public** **class** A3\_Q2 {

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

Scanner sc=**new** Scanner(System.***in***);

String s=sc.nextLine();

balance b=**new** balance();

**int** j=-1;

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

j=b.push(s.charAt(i));

}

**if**(j==0)

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

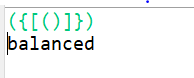
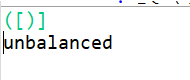
**else**

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

sc.close();

}

}

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

Solution—

**package** assign3;

**import** java.util.Scanner;

**class** Stacks{

**static** **final** **int** ***MAX***=20;

**char** a[]=**new** **char**[***MAX***];

**int** top;

Stacks(){

top=-1;

}

**boolean** push(StringBuffer s) {

**if**(top>=***MAX***-1)

{

System.***out***.println("stack overflow");

**return** **false**;

}

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

a[++top]=s.charAt(i);

}

**return** **true**;

}

**char** pop() {

**if**(top<0) {

System.***out***.println("stack underflow");

**return** '0';

}

**return** a[top--];

}

**void** reverse(StringBuffer s) {

**if**(top<0)

{

System.***out***.println("Empty stack");

**return**;

}

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

**char** ch = pop();

s.setCharAt(i, ch);

}

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

}

}

**public** **class** A3\_Q3 {

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

Scanner sc=**new** Scanner(System.***in***);

String str=sc.nextLine();

StringBuffer s=**new** StringBuffer(str);

Stacks sa=**new** Stacks();

sa.push(s);

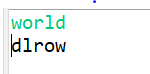
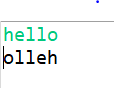
sa.reverse(s);

}

}

**Test Case 1**:  
Input: "hello"  
Output: "olleh"

* **Test Case 2**:  
  Input: "world"  
  Output: "dlrow"



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

**package** assign3;

**import** java.lang.Math;

**import** java.util.\*;

**class** Stackex {

**static** **int** *MAX*;

**int** top;

**int** a[];

Stackex(**int** n) {

top=-1;

*MAX*=n;

a=**new** **int**[*MAX*];

}

**boolean** push(**int** c)

{

**if**(top>=*MAX*-1)

{

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

**return** **false**;

}

a[++top]=c;

**return** **true**;

}

**int** pop() {

**if**(top<0)

{

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

**return** 0;

}

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

**return** p;

}

**void** display() {

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

System.***out***.print(pop());

}

}

**public** **class** A3\_Q4 {

**static** **int** evaluate(String s)

{

Stackex se=**new** Stackex(s.length());

**for**(**int** i=0;i<s.length();i++)

{ **char** c=s.charAt(i);

**if**(Character.*isDigit*(c))

se.push(c-'0');

**else** {

**int** v1=se.pop();

**int** v2=se.pop();

**switch**(c) {

**case** '+':

se.push(v2+v1);

**break**;

**case** '-':

se.push(v2-v1);

**break**;

**case** '\*':

se.push(v2\*v1);

**break**;

**case** '/':

se.push(v2/v1);

**break**;

**case** '^':

se.push(v2^v1);

**break**;

}

}

}

**return** se.pop();

}

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

Scanner sc=**new** Scanner(System.***in***);

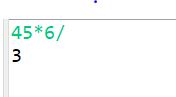
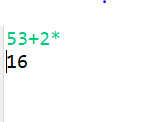
String str=sc.nextLine();

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

sc.close();

}

}



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

**package** assign3;

**import** java.util.Scanner;

**class** Stac{

**int** top;

**char** a[];

**static** **int** *MAX*;

Stac(**int** size)

{ top=-1;

*MAX*=size;

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

}

**boolean** isEmpty() {

**return**(top<0);

}

**boolean** isFull() {

**return**(top>=*MAX*-1);

}

**void** push(**char** c) {

**if**(isFull())

{ System.***out***.println("overflow");

**return**; }

a[++top]=c;

}

**char** pop() { **if**(isEmpty())

{ System.***out***.println("underflow");

**return** '0'; }

**return** a[top--];

}

**char** peek() {

**return** top<0?'0':a[top];

}

}

**class** StackApp1{

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

**switch** (ch) {

**case** '+':

**case** '-':

**return** 1;

**case** '\*':

**case** '/':

**return** 2;

**case** '^':

**return** 3;

}

**return** -1;

}

**static** StringBuffer evaluation(String s) {

Stac st=**new** Stac(s.length());

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

cnew=s.toCharArray();

StringBuffer sb=**new** StringBuffer();

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

**if**(Character.*isLetter*(cnew[i])) {

sb.append(cnew[i]);

}

**else** {

**while**(!st.isEmpty() && *precedence*(cnew[i]) <= *precedence*(st.peek()))

{

sb.append(st.pop());

}

st.push(cnew[i]);

}

}

**while**(!st.isEmpty())

sb.append(st.pop());

**return** sb;

}

}

**public** **class** A3\_Q5 {

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

Scanner sc=**new** Scanner(System.***in***);

String s=sc.nextLine();

StringBuffer sb =**new** StringBuffer();

sb.append(StackApp1.*evaluation*(s));

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

sc.close();

}

}

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

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

**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** assign3;

**class** Q {

**private** **int**[] queue;

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

// Constructor to initialize the queue

**public** Q(**int** capacity) {

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

front = -1;

rear = -1;

size = capacity;

}

// Check if the queue is full

**public** **boolean** isFull() {

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

}

// Check if the queue is empty

**public** **boolean** isEmpty() {

**return** front == -1;

}

// Enqueue operation to add an element

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

**if** (isFull()) {

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

**return**;

}

**if** (isEmpty()) {

front = 0;

}

rear = (rear + 1) % size;

queue[rear] = value;

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

}

// Dequeue operation to remove an element

**public** **int** dequeue() {

**if** (isEmpty()) {

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

**return** -1;

}

**int** result = queue[front];

**if** (front == rear) { // Queue has only one element

front = -1;

rear = -1;

} **else** {

front = (front + 1) % size;

}

System.***out***.println("Dequeued " + result);

**return** result;

}

// Function to print the current state of the queue

**public** **void** printQueue() {

**if** (isEmpty()) {

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

**return**;

}

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

**int** i = front;

**while** (**true**) {

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

**if** (i == rear) **break**;

System.***out***.print(", ");

i = (i + 1) % size;

}

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

}

}

**public** **class** A3\_Q7 {

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

Q cq = **new** Q(5); // Create a queue with capacity 5

// Test Case 1

cq.enqueue(4);

cq.enqueue(5);

cq.enqueue(6);

cq.enqueue(7);

cq.dequeue();

cq.enqueue(8);

cq.printQueue(); }

}

**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;

class QueueUsingStacks {

private Stack<Integer> stack1;

private Stack<Integer> stack2;

public QueueUsingStacks() {

stack1 = new Stack<>();

stack2 = new Stack<>();

}

// Enqueue operation

public void enqueue(int value) {

stack1.push(value);

}

// Dequeue operation

public int dequeue() {

if (stack2.isEmpty()) {

// Move elements from stack1 to stack2

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

// If stack2 is still empty, the queue is empty

if (stack2.isEmpty()) {

throw new RuntimeException("Queue is empty");

}

return stack2.pop();

}

// Peek operation to see the front element

public int peek() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

if (stack2.isEmpty()) {

throw new RuntimeException("Queue is empty");

}

return stack2.peek();

}

// Check if the queue is empty

public boolean isEmpty() {

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

}

// Get the size of the queue

public int size() {

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

}

}

public class Main {

public static void main(String[] args) {

QueueUsingStacks queue = new QueueUsingStacks();

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

System.out.println("Dequeue: " + queue.dequeue()); // Outputs: 1

System.out.println("Peek: " + queue.peek()); // Outputs: 2

System.out.println("Size: " + queue.size()); // Outputs: 2

System.out.println("Dequeue: " + queue.dequeue()); // Outputs: 2

System.out.println("Dequeue: " + queue.dequeue()); // Outputs: 3

System.out.println("Is queue empty? " + queue.isEmpty()); // Outputs: true

}

}

**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.Arrays;

class MinHeap {

private int[] heap;

private int size;

private int capacity;

// Constructor to initialize the heap

public MinHeap(int capacity) {

this.capacity = capacity;

this.size = 0;

heap = new int[capacity];

}

// Helper method to get the index of the parent of a node

private int parent(int index) {

return (index - 1) / 2;

}

// Helper method to get the index of the left child of a node

private int leftChild(int index) {

return (2 \* index) + 1;

}

// Helper method to get the index of the right child of a node

private int rightChild(int index) {

return (2 \* index) + 2;

}

// Swap two elements in the heap

private void swap(int i, int j) {

int temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

// Insert a new value into the heap

public void insert(int value) {

if (size == capacity) {

throw new IllegalStateException("Heap is full");

}

// Insert the new element at the end of the heap

heap[size] = value;

size++;

// Heapify up to maintain the min-heap property

int currentIndex = size - 1;

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

swap(currentIndex, parent(currentIndex));

currentIndex = parent(currentIndex);

}

}

// Heapify down to maintain the min-heap property after extraction

private void heapifyDown(int index) {

int smallest = index;

int left = leftChild(index);

int right = rightChild(index);

// Find the smallest among the node and its children

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

smallest = left;

}

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

smallest = right;

}

// If the smallest is not the current node, swap and continue heapifying

if (smallest != index) {

swap(index, smallest);

heapifyDown(smallest);

}

}

// Extract the minimum element (root) from the heap

public int extractMin() {

if (size == 0) {

throw new IllegalStateException("Heap is empty");

}

int min = heap[0];

// Replace the root with the last element and reduce the heap size

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

size--;

// Heapify down to restore the min-heap property

heapifyDown(0);

return min;

}

// Peek at the minimum element without removing it

public int getMin() {

if (size == 0) {

throw new IllegalStateException("Heap is empty");

}

return heap[0];

}

// Print the heap as an array

public void printHeap() {

System.out.println(Arrays.toString(Arrays.copyOfRange(heap, 0, size)));

}

}

public class Main {

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(10);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

System.out.println("Heap array:");

minHeap.printHeap(); // Output the heap structure

System.out.println("Extracted Min: " + minHeap.extractMin()); // Output: 3

System.out.println("Heap array after extraction:");

minHeap.printHeap();

System.out.println("Current Min: " + minHeap.getMin()); // Output: 5

}

}

**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.Arrays;
* class MaxHeap {
* private int[] heap;
* private int size;
* private int capacity;
* // Constructor to initialize the heap
* public MaxHeap(int capacity) {
* this.capacity = capacity;
* this.size = 0;
* heap = new int[capacity];
* }
* // Helper method to get the index of the parent of a node
* private int parent(int index) {
* return (index - 1) / 2;
* }
* // Helper method to get the index of the left child of a node
* private int leftChild(int index) {
* return (2 \* index) + 1;
* }
* // Helper method to get the index of the right child of a node
* private int rightChild(int index) {
* return (2 \* index) + 2;
* }
* // Swap two elements in the heap
* private void swap(int i, int j) {
* int temp = heap[i];
* heap[i] = heap[j];
* heap[j] = temp;
* }
* // Insert a new value into the heap
* public void insert(int value) {
* if (size == capacity) {
* throw new IllegalStateException("Heap is full");
* }
* // Insert the new element at the end of the heap
* heap[size] = value;
* size++;
* // Heapify up to maintain the max-heap property
* int currentIndex = size - 1;
* while (currentIndex > 0 && heap[currentIndex] > heap[parent(currentIndex)]) {
* swap(currentIndex, parent(currentIndex));
* currentIndex = parent(currentIndex);
* }
* }
* // Heapify down to maintain the max-heap property after extraction
* private void heapifyDown(int index) {
* int largest = index;
* int left = leftChild(index);
* int right = rightChild(index);
* // Find the largest among the node and its children
* if (left < size && heap[left] > heap[largest]) {
* largest = left;
* }
* if (right < size && heap[right] > heap[largest]) {
* largest = right;
* }
* // If the largest is not the current node, swap and continue heapifying
* if (largest != index) {
* swap(index, largest);
* heapifyDown(largest);
* }
* }
* // Extract the maximum element (root) from the heap
* public int extractMax() {
* if (size == 0) {
* throw new IllegalStateException("Heap is empty");
* }
* int max = heap[0];
* // Replace the root with the last element and reduce the heap size
* heap[0] = heap[size - 1];
* size--;
* // Heapify down to restore the max-heap property
* heapifyDown(0);
* return max;
* }
* // Peek at the maximum element without removing it
* public int getMax() {
* if (size == 0) {
* throw new IllegalStateException("Heap is empty");
* }
* return heap[0];
* }
* // Print the heap as an array
* public void printHeap() {
* System.out.println(Arrays.toString(Arrays.copyOfRange(heap, 0, size)));
* }
* }
* public class Main {
* public static void main(String[] args) {
* MaxHeap maxHeap = new MaxHeap(10);
* maxHeap.insert(10);
* maxHeap.insert(4);
* maxHeap.insert(9);
* maxHeap.insert(1);
* maxHeap.insert(7);
* maxHeap.insert(5);
* maxHeap.insert(3);
* System.out.println("Heap array:");
* maxHeap.printHeap(); // Output the heap structure
* System.out.println("Extracted Max: " + maxHeap.extractMax()); // Output: 10
* System.out.println("Heap array after extraction:");
* maxHeap.printHeap();
* System.out.println("Current Max: " + maxHeap.getMax()); // Output: 9
* }
* }

**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;

class HeapSort {

// Helper method to swap two elements in the array

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

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

// Heapify a subtree rooted at index i

// n is the size of the heap

private static void heapify(int[] array, 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 the left child is larger than root

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

largest = left;

}

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

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

largest = right;

}

// If the largest is not root, swap it with the largest and heapify the affected subtree

if (largest != i) {

swap(array, i, largest);

// Recursively heapify the affected subtree

heapify(array, n, largest);

}

}

// Function to perform heap sort

public static void heapSort(int[] array) {

int n = array.length;

// Build a max-heap from the input array

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

heapify(array, n, i);

}

// One by one, extract elements from the heap

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

// Move the current root (largest) to the end

swap(array, 0, i);

// Reduce the heap size and heapify the root element

heapify(array, i, 0);

}

}

public static void main(String[] args) {

int[] array = {12, 11, 13, 5, 6, 7};

System.out.println("Original Array: " + Arrays.toString(array));

heapSort(array);

System.out.println("Sorted Array: " + Arrays.toString(array));

}

}

**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

import java.util.PriorityQueue;

class KthLargestInStream {

private PriorityQueue<Integer> minHeap;

private int k;

// Constructor to initialize the heap and set the value of k

public KthLargestInStream(int k, int[] stream) {

this.k = k;

// A min-heap to store the k largest elements

minHeap = new PriorityQueue<>(k);

// Insert the initial elements from the stream into the heap

for (int num : stream) {

add(num);

}

}

// Function to add a new number from the stream and maintain the heap

public int add(int num) {

// Add the new number to the heap

if (minHeap.size() < k) {

minHeap.offer(num); // Add directly if the heap size is less than k

} else if (num > minHeap.peek()) {

minHeap.poll(); // Remove the smallest element

minHeap.offer(num); // Insert the new number

}

// Return the kth largest element (the root of the heap)

return minHeap.peek();

}

// Function to get the current kth largest element

public int getKthLargest() {

return minHeap.peek();

}

}

public class Main {

public static void main(String[] args) {

int k = 3;

int[] initialStream = {4, 5, 8, 2};

// Initialize the object with the given stream

KthLargestInStream kthLargest = new KthLargestInStream(k, initialStream);

System.out.println("Initial kth largest: " + kthLargest.getKthLargest()); // Should print 4

// Add new elements from the stream and get the kth largest element

System.out.println("Add 3: " + kthLargest.add(3)); // Output: 4

System.out.println("Add 5: " + kthLargest.add(5)); // Output: 5

System.out.println("Add 10: " + kthLargest.add(10)); // Output: 5

System.out.println("Add 9: " + kthLargest.add(9)); // Output: 8

System.out.println("Add 4: " + kthLargest.add(4)); // 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]

import java.util.Arrays;

class PriorityQueueUsingHeap {

private int[] heap;

private int size;

private int capacity;

// Constructor to initialize the priority queue

public PriorityQueueUsingHeap(int capacity) {

this.capacity = capacity;

this.size = 0;

this.heap = new int[capacity];

}

// Helper method to get the index of the parent of a node

private int parent(int index) {

return (index - 1) / 2;

}

// Helper method to get the index of the left child of a node

private int leftChild(int index) {

return (2 \* index) + 1;

}

// Helper method to get the index of the right child of a node

private int rightChild(int index) {

return (2 \* index) + 2;

}

// Swap two elements in the heap

private void swap(int i, int j) {

int temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

// Insert a new element into the priority queue

public void insert(int value) {

if (size == capacity) {

throw new IllegalStateException("Priority Queue is full");

}

// Insert the new element at the end of the heap

heap[size] = value;

size++;

// Heapify up to maintain the min-heap property

int currentIndex = size - 1;

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

swap(currentIndex, parent(currentIndex));

currentIndex = parent(currentIndex);

}

}

// Heapify down to maintain the min-heap property after extraction

private void heapifyDown(int index) {

int smallest = index;

int left = leftChild(index);

int right = rightChild(index);

// Find the smallest among the node and its children

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

smallest = left;

}

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

smallest = right;

}

// If the smallest is not the current node, swap and continue heapifying

if (smallest != index) {

swap(index, smallest);

heapifyDown(smallest);

}

}

// Extract the element with the highest priority (smallest value)

public int extractMin() {

if (size == 0) {

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

}

int min = heap[0];

// Replace the root with the last element and reduce the heap size

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

size--;

// Heapify down to restore the min-heap property

heapifyDown(0);

return min;

}

// Peek at the element with the highest priority (smallest value)

public int peekMin() {

if (size == 0) {

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

}

return heap[0];

}

// Return true if the priority queue is empty

public boolean isEmpty() {

return size == 0;

}

// Print the priority queue as an array

public void printQueue() {

System.out.println(Arrays.toString(Arrays.copyOfRange(heap, 0, size)));

}

}

public class Main {

public static void main(String[] args) {

PriorityQueueUsingHeap pq = new PriorityQueueUsingHeap(10);

pq.insert(3);

pq.insert(10);

pq.insert(5);

pq.insert(1);

pq.insert(7);

System.out.println("Priority Queue:");

pq.printQueue();

System.out.println("Extracted Min: " + pq.extractMin()); // Should return 1

System.out.println("Priority Queue after extraction:");

pq.printQueue();

System.out.println("Current Min: " + pq.peekMin()); // Should return 3

}

}

**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
* **package** assign3;
* **class** Main{
* **int** top;
* **int** a[]=**new** **int**[6];
* **int** min;
* Main() {
* top=-1;
* }
* **void** push(**int** d) {
* **if**(top>=5)
* **return**;
* **if**(top==-1)
* min=d;
* **else**
* **if**(d<min)
* min=d;
* a[++top]=d;
* }
* **void** pop() {
* **if**(top<-1)
* **return**;
* top--;
* **return**;
* }
* **void** getMin() {
* System.***out***.println(min);
* }


* }
* **public** **class** A3\_Q14 {
* **public** **static** **void** main(String[] args) {
* Main s=**new** Main();
* s.push(10);
* s.push(8);
* s.push(6);
* s.push(12);
* s.getMin();
* }
* }

**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