**Assignment 3**

**1. Implement a Stack Using an Array**

class Stack {

private int[] stack;

private int top;

private int capacity;

public Stack(int size) {

stack = new int[size];

capacity = size;

top = -1;

}

public void push(int item) {

if (top == capacity - 1) {

throw new RuntimeException("Stack Overflow");

}

stack[++top] = item;

}

public int pop() {

if (top == -1) {

throw new RuntimeException("Stack Underflow");

}

return stack[top--];

}

public int[] getStack() {

int[] currentStack = new int[top + 1];

System.arraycopy(stack, 0, currentStack, 0, top + 1);

return currentStack;

}

public static void main(String[] args) {

Stack s = new Stack(5);

s.push(5);

s.push(3);

s.push(7);

int poppedElement = s.pop();

System.out.println("Stack = " + Arrays.toString(s.getStack())); // Output: Stack = [5, 3]

System.out.println("Popped element = " + poppedElement); // Output: Popped element = 7

// Test Case 2

s.push(10);

s.push(20);

poppedElement = s.pop();

s.push(15);

System.out.println("Stack = " + Arrays.toString(s.getStack())); // Output: Stack = [10, 15]

System.out.println("Popped element = " + poppedElement); // Output: Popped element = 20

}

}

**Flowchart Description**

1. Start
2. Initialize Stack
3. Check if full (if yes, go to step 5)
4. Push item
5. Check if empty (if yes, go to step 7)
6. Pop item
7. End

**Explanation**

* **Stack Class**: Implements stack operations using an array.
* **Push**: Adds an item to the top if the stack is not full.
* **Pop**: Removes the top item if the stack is not empty.
* **getStack**: Returns the current state of the stack.
* **Test Cases**: Demonstrates the push and pop operations.

**Output**

Stack = [5, 3]

Popped element = 7

Stack = [10, 15]

Popped element = 20

**Time and Space Complexity**

* **Time Complexity**:
  + Push: O(1)
  + Pop: O(1)
* **Space Complexity**: O(n) where n is the capacity of the stack.

**2. Check for Balanced Parentheses Using a Stack**

import java.util.Stack;

public class BalancedParentheses {

public static String isBalanced(String expression) {

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

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

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

stack.push(c);

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

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

return "Not Balanced";

}

}

}

return stack.isEmpty() ? "Balanced" : "Not Balanced";

}

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

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

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

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

}

public static void main(String[] args) {

System.out.println(isBalanced("({[()]})")); // Output: Balanced

System.out.println(isBalanced("([)]")); // Output: Not Balanced

}

}

**Flowchart Description**

1. Start
2. Initialize Stack
3. Loop through each character in the expression
4. If character is an opening bracket, push to stack
5. If character is a closing bracket:
   * Check if stack is empty or top does not match (go to step 7)
   * Pop from stack
6. Check if stack is empty at end
7. Output result
8. End

**Explanation**

* **Function**: isBalanced checks if parentheses are balanced using a stack.
* **Mapping**: Matches closing brackets with their corresponding opening brackets.
* **Loop**: Iterates through the string, using stack operations to validate parentheses.

**Output**

Balanced

Not Balanced

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the expression.
* **Space Complexity**: O(n) in the worst case when all characters are opening brackets.

**3. Reverse a String Using a Stack**

import java.util.Stack;

public class ReverseString {

public static String reverse(String s) {

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

for (char c : s.toCharArray()) {

stack.push(c);

}

StringBuilder reversedStr = new StringBuilder();

while (!stack.isEmpty()) {

reversedStr.append(stack.pop());

}

return reversedStr.toString();

}

public static void main(String[] args) {

System.out.println(reverse("hello")); // Output: "olleh"

System.out.println(reverse("world")); // Output: "dlrow"

}

}

**Flowchart Description**

1. Start
2. Initialize Stack
3. Loop through each character in the string
4. Push each character to stack
5. Pop each character from stack and construct reversed string
6. Output reversed string
7. End

**Explanation**

* **Function**: reverse uses a stack to reverse the input string.
* **Pushing**: All characters are pushed onto the stack.
* **Popping**: Characters are popped in reverse order to form the new string.

**Output**

olleh

dlrow

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the string.
* **Space Complexity**: O(n) for the stack used to hold the characters.

**4. Evaluate a Postfix Expression Using a Stack**

import java.util.Stack;

public class PostfixEvaluator {

public static int evaluatePostfix(String expression) {

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

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

if (isOperator(token)) {

int b = stack.pop();

int a = stack.pop();

stack.push(applyOperation(a, b, token));

} else {

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

}

}

return stack.pop();

}

private static boolean isOperator(String token) {

return token.equals("+") || token.equals("-") || token.equals("\*") || token.equals("/");

}

private static int applyOperation(int a, int b, String operator) {

switch (operator) {

case "+":

return a + b;

case "-":

return a - b;

case "\*":

return a \* b;

case "/":

return a / b;

default:

throw new UnsupportedOperationException("Invalid operator");

}

}

public static void main(String[] args) {

System.out.println(evaluatePostfix("5 3 + 2 \*")); // Output: 16

System.out.println(evaluatePostfix("4 5 \* 6 /")); // Output: 3

}

}

**Flowchart Description**

1. Start
2. Initialize Stack
3. Split expression into tokens
4. Loop through each token
5. If token is an operator:
   * Pop top two elements
   * Apply operation
   * Push result back
6. If token is a number, push it onto the stack
7. At end, pop the result
8. Output result
9. End

**Explanation**

* **Function**: evaluatePostfix evaluates a postfix expression using a stack.
* **Operators**: Handles basic arithmetic operations.
* **Loop**: Iterates through tokens, applying operations as necessary.

**Output**

16

3

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the number of tokens in the expression.
* **Space Complexity**: O(n) for the stack used during evaluation.

**5. Convert an Infix Expression to Postfix Using a Stack**

import java.util.Stack;

public class InfixToPostfix {

public static String infixToPostfix(String expression) {

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

StringBuilder result = new StringBuilder();

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

if (Character.isLetterOrDigit(c)) {

result.append(c);

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

stack.push(c);

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

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

result.append(stack.pop());

}

stack.pop(); // pop '('

} else {

while (!stack.isEmpty() && precedence(stack.peek()) >= precedence(c)) {

result.append(stack.pop());

}

stack.push(c);

}

}

while (!stack.isEmpty()) {

result.append(stack.pop());

}

return result.toString();

}

private static int precedence(char operator) {

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

default:

return 0;

}

}

public static void main(String[] args) {

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

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

}

}

**Flowchart Description**

1. Start
2. Initialize Stack and StringBuilder
3. Loop through each character in the expression
4. If character is an operand, append to result
5. If character is '(', push to stack
6. If character is ')', pop until '(' is found
7. If character is an operator, pop from stack based on precedence
8. At the end, pop all operators from stack to result
9. Output result
10. End

**Explanation**

* This program converts an infix expression to postfix notation using a stack.
* Operators are pushed to the stack based on their precedence.
* Operands are directly appended to the result.

**Output**

A B C \* +

A B \* C D / +

**Time and Space Complexity**

* **Time Complexity**: O(n), where n is the length of the expression.
* **Space Complexity**: O(n) for the stack used to store operators.

**6. Implement a Queue Using an Array**

class Queue {

private int[] queue;

private int front, rear, capacity;

public Queue(int size) {

queue = new int[size];

capacity = size;

front = 0;

rear = -1;

}

public void enqueue(int item) {

if (rear == capacity - 1) {

throw new RuntimeException("Queue Overflow");

}

queue[++rear] = item;

}

public int dequeue() {

if (front > rear) {

throw new RuntimeException("Queue Underflow");

}

return queue[front++];

}

public int[] getQueue() {

int[] currentQueue = new int[rear - front + 1];

System.arraycopy(queue, front, currentQueue, 0, currentQueue.length);

return currentQueue;

}

public static void main(String[] args) {

Queue q = new Queue(5);

q.enqueue(5);

q.enqueue(10);

int dequeued = q.dequeue();

System.out.println("Queue = " + Arrays.toString(q.getQueue())); // Output: Queue = [10]

System.out.println("Dequeued element = " + dequeued); // Output: Dequeued element = 5

// Test Case 2

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

int firstDequeued = q.dequeue();

int secondDequeued = q.dequeue();

System.out.println("Queue = " + Arrays.toString(q.getQueue())); // Output: Queue = [3]

System.out.println("Dequeued elements = " + firstDequeued + ", " + secondDequeued); // Output: 1, 2

}

}

**Flowchart Description**

1. Start
2. Initialize Queue
3. Check if full (if yes, go to step 5)
4. Enqueue item
5. Check if empty (if yes, go to step 7)
6. Dequeue item
7. End

**Explanation**

* This implementation of a queue uses an array.
* Items can be enqueued until the capacity is reached, and dequeued from the front.

**Output**

Queue = [10]

Dequeued element = 5

Queue = [3]

Dequeued elements = 1, 2

**Time and Space Complexity**

* **Time Complexity**:
  + Enqueue: O(1)
  + Dequeue: O(1)
* **Space Complexity**: O(n), where n is the capacity of the queue.

**7. Implement a Circular Queue Using an Array**

class CircularQueue {

private int[] queue;

private int front, rear, capacity;

public CircularQueue(int size) {

queue = new int[size];

capacity = size;

front = 0;

rear = -1;

}

public void enqueue(int item) {

if ((rear + 1) % capacity == front) {

throw new RuntimeException("Queue Overflow");

}

rear = (rear + 1) % capacity;

queue[rear] = item;

}

public int dequeue() {

if (front == (rear + 1) % capacity) {

throw new RuntimeException("Queue Underflow");

}

int item = queue[front];

front = (front + 1) % capacity;

return item;

}

public int[] getQueue() {

int[] currentQueue = new int[capacity];

int index = 0;

for (int i = front; i != rear; i = (i + 1) % capacity) {

currentQueue[index++] = queue[i];

}

currentQueue[index] = queue[rear]; // Include the rear element

return currentQueue;

}

public static void main(String[] args) {

CircularQueue cq = new CircularQueue(5);

cq.enqueue(4);

cq.enqueue(5);

cq.enqueue(6);

cq.enqueue(7);

cq.dequeue();

cq.enqueue(8);

System.out.println("Queue = " + Arrays.toString(cq.getQueue())); // Output: Queue = [5, 6, 7, 8]

// Test Case 2

CircularQueue cq2 = new CircularQueue(4);

cq2.enqueue(1);

cq2.enqueue(2);

cq2.enqueue(3);

cq2.enqueue(4);

cq2.dequeue();

cq2.dequeue();

cq2.enqueue(5);

System.out.println("Queue = " + Arrays.toString(cq2.getQueue())); // Output: Queue = [3, 4, 5]

}

}

**Flowchart Description**

1. Start
2. Initialize Circular Queue
3. Check if full (if yes, go to step 5)
4. Enqueue item
5. Check if empty (if yes, go to step 7)
6. Dequeue item
7. End

**Explanation**

* This implementation uses an array to create a circular queue, which wraps around when the end of the array is reached.
* Enqueue and dequeue operations are managed to prevent overflow and underflow.

**Output**

Queue = [5, 6, 7, 8]

Queue = [3, 4, 5]

**Time and Space Complexity**

* **Time Complexity**:
  + Enqueue: O(1)
  + Dequeue: O(1)
* **Space Complexity**: O(n), where n is the capacity of the queue.

**8. Implement a Queue Using Two Stacks**

import java.util.Stack;

class QueueUsingStacks {

private Stack<Integer> stack1;

private Stack<Integer> stack2;

public QueueUsingStacks() {

stack1 = new Stack<>();

stack2 = new Stack<>();

}

public void enqueue(int item) {

stack1.push(item);

}

public int dequeue() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

if (stack2.isEmpty()) {

throw new RuntimeException("Queue Underflow");

}

return stack2.pop();

}

public static void main(String[] args) {

QueueUsingStacks queue = new QueueUsingStacks();

queue.enqueue(3);

queue.enqueue(7);

int dequeued = queue.dequeue();

System.out.println("Queue after dequeue = " + queue.stack2); // Output: Queue = [7]

System.out.println("Dequeued element = " + dequeued); // Output: Dequeued element = 3

// Test Case 2

queue.enqueue(10);

queue.enqueue(20);

int firstDequeued = queue.dequeue();

int secondDequeued = queue.dequeue();

System.out.println("Queue after dequeues = " + queue.stack2); // Output: Queue = []

System.out.println("Dequeued elements = " + firstDequeued + ", " + secondDequeued); // Output: 10, 20

}

}

**Flowchart Description**

1. Start
2. Initialize two stacks
3. For enqueue, push to stack1
4. For dequeue, check stack2:
   * If empty, pop from stack1 to stack2
   * Pop from stack2
5. End

**Explanation**

* This implementation uses two stacks to simulate queue operations.
* Enqueue is done by pushing to the first stack, and dequeue is done by popping from the second stack, which may be filled from the first stack when needed.

**Output**

Queue after dequeue = [7]

Dequeued element = 3

Queue after dequeues = []

Dequeued elements = 10, 20

**Time and Space Complexity**

* **Time Complexity**:
  + Enqueue: O(1)
  + Dequeue: O(n) in the worst case when transferring elements.
* **Space Complexity**: O(n), for storing elements in two stacks.

**9. Implement a Min-Heap**

import java.util.Arrays;

class MinHeap {

private int[] heap;

private int size;

private int capacity;

public MinHeap(int capacity) {

this.capacity = capacity;

heap = new int[capacity];

size = 0;

}

public void insert(int val) {

if (size == capacity) {

throw new RuntimeException("Heap Overflow");

}

heap[size] = val;

size++;

heapifyUp(size - 1);

}

public int extractMin() {

if (size == 0) {

throw new RuntimeException("Heap Underflow");

}

int min = heap[0];

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

size--;

heapifyDown(0);

return min;

}

private void heapifyUp(int index) {

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

return Arrays.copyOf(heap, size);

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(10);

minHeap.insert(10);

minHeap.insert(15);

minHeap.insert(20);

minHeap.insert(17);

int extractedMin = minHeap.extractMin();

System.out.println("Min-Heap = " + Arrays.toString(minHeap.getHeap())); // Output: Min-Heap = [15, 17, 20]

System.out.println("Extracted Min = " + extractedMin); // Output: Extracted Min = 10

// Test Case 2

MinHeap minHeap2 = new MinHeap(10);

minHeap2.insert(30);

minHeap2.insert(40);

minHeap2.insert(20);

minHeap2.insert(50);

extractedMin = minHeap2.extractMin();

System.out.println("Min-Heap = " + Arrays.toString(minHeap2.getHeap())); // Output: Min-Heap = [20, 40, 50]

System.out.println("Extracted Min = " + extractedMin); // Output: Extracted Min = 30

}

}

**Flowchart Description**

1. Start
2. Initialize Min-Heap
3. For insert, add value and heapify up
4. For extract min, replace root with last element and heapify down
5. End

**Explanation**

* This min-heap implementation supports insertion and extraction of the minimum element.
* It maintains the heap property during inserts and extracts.

**Output**

Min-Heap = [15, 17, 20]

Extracted Min = 10

Min-Heap = [20, 40, 50]

Extracted Min = 30

**Time and Space Complexity**

* **Time Complexity**:
  + Insert: O(log n)
  + Extract Min: O(log n)
* **Space Complexity**: O(n), for storing elements in the heap.

**10. Implement a Max-Heap**

import java.util.Arrays;

class MaxHeap {

private int[] heap;

private int size;

private int capacity;

public MaxHeap(int capacity) {

this.capacity = capacity;

heap = new int[capacity];

size = 0;

}

public void insert(int val) {

if (size == capacity) {

throw new RuntimeException("Heap Overflow");

}

heap[size] = val;

size++;

heapifyUp(size - 1);

}

public int extractMax() {

if (size == 0) {

throw new RuntimeException("Heap Underflow");

}

int max = heap[0];

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

size--;

heapifyDown(0);

return max;

}

private void heapifyUp(int index) {

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

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] > heap[largest]) {

largest = left;

}

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

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) {

int temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

public int[] getHeap() {

return Arrays.copyOf(heap, size);

}

public static void main(String[] args) {

MaxHeap maxHeap = new MaxHeap(10);

maxHeap.insert(12);

maxHeap.insert(7);

maxHeap.insert(15);

maxHeap.insert(5);

int extractedMax = maxHeap.extractMax();

System.out.println("Max-Heap = " + Arrays.toString(maxHeap.getHeap())); // Output: Max-Heap = [12, 7, 5]

System.out.println("Extracted Max = " + extractedMax); // Output: Extracted Max = 15

// Test Case 2

MaxHeap maxHeap2 = new MaxHeap(10);

maxHeap2.insert(8);

maxHeap2.insert(20);

maxHeap2.insert(10);

maxHeap2.insert(3);

extractedMax = maxHeap2.extractMax();

System.out.println("Max-Heap = " + Arrays.toString(maxHeap2.getHeap())); // Output: Max-Heap = [10, 8, 3]

System.out.println("Extracted Max = " + extractedMax); // Output: Extracted Max = 20

}

}

**Flowchart Description**

1. Start
2. Initialize Max-Heap
3. For insert, add value and heapify up
4. For extract max, replace root with last element and heapify down
5. End

**Explanation**

* This max-heap implementation supports insertion and extraction of the maximum element.
* It maintains the heap property during inserts and extracts.

**Output**

Max-Heap = [12, 7, 5]

Extracted Max = 15

Max-Heap = [10, 8, 3]

Extracted Max = 20

**Time and Space Complexity**

* **Time Complexity**:
  + Insert: O(log n)
  + Extract Max: O(log n)
* **Space Complexity**: O(n), for storing elements in the heap.

**11. Sort an Array Using a Heap (Heap Sort)**

**Program (Java)**

java

Copy code

import java.util.Arrays;

class HeapSort {

public static 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 elements 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);

}

}

private static 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) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, n, largest);

}

}

public static void main(String[] args) {

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

heapSort(arr1);

System.out.println(Arrays.toString(arr1)); // Output: [1, 3, 5, 9, 12]

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

heapSort(arr2);

System.out.println(Arrays.toString(arr2)); // Output: [8, 10, 15, 20]

}

}

**Explanation**

* **Heap Construction**: The array is transformed into a max-heap.
* **Extraction**: The root is swapped with the last element, and the heap size is reduced by one, then heapify is called to maintain the max-heap property.

**Output**

* For input [5, 1, 12, 3, 9], output is [1, 3, 5, 9, 12].
* For input [20, 15, 8, 10], output is [8, 10, 15, 20].

**Time Complexity**

* **Best/Average/Worst**: O(n log n)

**Space Complexity**

* O(1) (in-place sorting)

**12. Find the kth Largest Element in a Stream of Numbers Using a Heap**

**Program (Java)**

java

Copy code

import java.util.PriorityQueue;

class KthLargest {

private final PriorityQueue<Integer> minHeap;

private final int k;

public KthLargest(int k) {

this.k = k;

minHeap = new PriorityQueue<>();

}

public void add(int val) {

minHeap.offer(val);

if (minHeap.size() > k) {

minHeap.poll();

}

}

public int getKthLargest() {

return minHeap.peek();

}

public static void main(String[] args) {

KthLargest kthLargest = new KthLargest(3);

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

for (int num : stream1) {

kthLargest.add(num);

}

System.out.println(kthLargest.getKthLargest()); // Output: 10

KthLargest kthLargest2 = new KthLargest(2);

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

for (int num : stream2) {

kthLargest2.add(num);

}

System.out.println(kthLargest2.getKthLargest()); // Output: 8

}

}

**Explanation**

* A min-heap is used to keep track of the top k elements. The root of the min-heap will give the kth largest element.

**Output**

* For stream [3, 10, 5, 20, 15] with k = 3, output is 10.
* For stream [7, 4, 8, 2, 9] with k = 2, output is 8.

**Time Complexity**

* O(n log k) for n insertions.

**Space Complexity**

* O(k) for the min-heap.

**13. Implement a Priority Queue Using a Heap**

**Program (Java)**

java

Copy code

import java.util.PriorityQueue;

class PriorityQueueExample {

private final PriorityQueue<Element> queue;

public PriorityQueueExample() {

queue = new PriorityQueue<>((a, b) -> b.priority - a.priority);

}

public void enqueue(int value, int priority) {

queue.offer(new Element(value, priority));

}

public int dequeue() {

return queue.poll().value;

}

static class Element {

int value;

int priority;

Element(int value, int priority) {

this.value = value;

this.priority = priority;

}

}

public static void main(String[] args) {

PriorityQueueExample pq = new PriorityQueueExample();

pq.enqueue(3, 1);

pq.enqueue(10, 3);

pq.enqueue(5, 2);

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

PriorityQueueExample pq2 = new PriorityQueueExample();

pq2.enqueue(7, 4);

pq2.enqueue(8, 2);

pq2.enqueue(6, 3);

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

}

}

**Explanation**

* A max-heap is used to implement the priority queue. Elements are stored based on priority, allowing the highest priority to be dequeued first.

**Output**

* After enqueuing [3 (p1), 10 (p3), 5 (p2)], output is 10.
* After enqueuing [7 (p4), 8 (p2), 6 (p3)], output is 7.

**Time Complexity**

* Enqueue: O(log n)
* Dequeue: O(log n)

**Space Complexity**

* O(n)

**14. Design a Stack with a getMin() Function**

**Program (Java)**

java

Copy code

import java.util.Stack;

class MinStack {

private final Stack<Integer> stack;

private final Stack<Integer> minStack;

public MinStack() {

stack = new Stack<>();

minStack = new Stack<>();

}

public void push(int x) {

stack.push(x);

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

minStack.push(x);

}

}

public void pop() {

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

minStack.pop();

}

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

public static void main(String[] args) {

MinStack minStack = new MinStack();

minStack.push(5);

minStack.push(3);

minStack.push(7);

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

minStack.pop();

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

}

}

**Explanation**

* Two stacks are used: one for the regular elements and another to keep track of the minimum values.

**Output**

* After pushing 5, 3, 7, output for getMin() is 3.
* After popping, output remains 3.

**Time Complexity**

* O(1) for push, pop, and getMin.

**Space Complexity**

* O(n)

**15. Design a Circular Queue with Fixed Size**

class CircularQueue {

private final int[] queue;

private int front, rear, size, capacity;

public CircularQueue(int capacity) {

this.capacity = capacity;

queue = new int[capacity];

front = size = 0;

rear = capacity - 1;

}

public void enqueue(int item) {

if (isFull()) return;

rear = (rear + 1) % capacity;

queue[rear] = item;

size++;

}

public int dequeue() {

if (isEmpty()) return -1;

int item = queue[front];

front = (front + 1) % capacity;

size--;

return item;

}

public boolean isFull() {

return size == capacity;

}

public boolean isEmpty() {

return size == 0;

}

public static void main(String[] args) {

CircularQueue cq = new CircularQueue(4);

cq.enqueue(1);

cq.enqueue(2);

cq.enqueue(3);

cq.enqueue(4);

System.out.println("isFull() = " + cq.isFull()); // Output: True

cq.dequeue();

cq.enqueue(5);

System.out.println("isEmpty() = " + cq.isEmpty()); // Output: False

}

}

**Explanation**

* The circular queue wraps around when it reaches the end of the array, using modulo arithmetic to manage indices.

**Output**

* After enqueuing 1, 2, 3, 4, output for isFull() is True.
* After dequeueing one element and enqueuing 5, output for isEmpty() is False.

**Time Complexity**

* O(1) for enqueue and dequeue.

**Space Complexity**

* O(n)