

Queue Questions



Basic Level

Implement a Queue using Arrays.

Design and implement a queue using a fixed-size array, with enqueue, dequeue, peek, and isEmpty operations.

Check if a Queue is Palindrome.

Given a queue of integers, check whether it reads the same forwards and backward.

Implement a Circular Queue using Arrays.

Design a circular queue with enqueue, dequeue, and isFull/isEmpty operations using arrays.

Implement a Queue using a Linked List.

Implement a basic queue where the elements are stored using a singly linked list.

Reverse the First K Elements of a Queue.

Given a queue, reverse the first K elements while maintaining the order of the rest.

Intermediate Level

Sort a Queue using Recursion.

Given a queue, sort it in ascending order using recursion without using any other data structures.

Implement a Queue using a Stack.

Design a queue using two stacks where enqueue and dequeue operations have an optimal time complexity.

Find the Maximum Element in a Sliding Window of Size K.

Given an array, find the maximum element in each sliding window of size K.

Generate Binary Numbers from 1 to N using a Queue.

Write a program that generates and prints all binary numbers between 1 and N using a queue.

Find the Minimum Time to Rot All Oranges (Multi-Source BFS).

In a matrix of oranges, where each cell represents an orange that can be fresh, rotten, or empty, find the minimum time required to rot all fresh oranges. Use BFS.

Advanced Level

LRU Cache Implementation using Deque.

Implement an LRU (Least Recently Used) cache using a queue (Deque) with a given cache size.

Design a Queue Supporting Get Min in $O(1)$.

Design a queue data structure that supports the following operations: enqueue, dequeue, and finding the minimum element in $O(1)$ time.

Sum of Minimum Elements in All Subarrays of Size K.

Given an array, find the sum of the minimum elements of all subarrays of size K.

Implement a Double-Ended Queue (Deque) using Arrays.

Design a deque data structure using arrays with `addFirst`, `addLast`, `removeFirst`, `removeLast`, and `peek` operations.

First Negative Integer in Every Window of Size K.

Given an array, for each sliding window of size K, find the first negative integer within the window.

explain problem with example that contains input and output in simple explanation and give java code in optimized and simple form and give time complexity



Memory updated

Let's break down each of the queue-related problems one by one with explanations, examples, and Java code. I'll start with the **basic level** problems.

1. Implement a Queue using Arrays

Problem:

Design a queue using a fixed-size array with operations like `enqueue`, `dequeue`, `peek`, and `isEmpty`.

Example:

- **Input:**
 - `enqueue(10)`, `enqueue(20)`, `dequeue()`, `enqueue(30)`
- **Output:**
 - After `enqueue(10)`: [10]
 - After `enqueue(20)`: [10, 20]
 - After `dequeue()`: [20]
 - After `enqueue(30)`: [20, 30]

Code:

java

```
class ArrayQueue {
    int[] arr;
    int front, rear, size, capacity;

    public ArrayQueue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = size = 0;
        rear = -1;
    }

    // Enqueue operation
    public void enqueue(int x) {
        if (size == capacity) {
            System.out.println("Queue is full");
            return;
        }
    }
}
```

```

        rear = (rear + 1) % capacity;
        arr[rear] = x;
        size++;
    }

    // Dequeue operation
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        int item = arr[front];
        front = (front + 1) % capacity;
        size--;
        return item;
    }

    // Peek at front element
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        return arr[front];
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return size == 0;
    }
}

```

- **Time Complexity:**

- `enqueue`: $O(1)$
- `dequeue`: $O(1)$
- `peek`: $O(1)$
- `isEmpty`: $O(1)$

2. Check if a Queue is Palindrome

Problem:

Given a queue of integers, check whether it reads the same forwards and backward.

Example:

- **Input:** Queue = [1, 2, 3, 2, 1]
- **Output:** True (Palindrome)

Code:

java

```

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class PalindromeQueue {
    public static boolean isPalindrome(Queue<Integer> queue) {
        Stack<Integer> stack = new Stack<>();
        Queue<Integer> tempQueue = new LinkedList<>(queue);
    }
}

```

```
// Push all elements to the stack
while (!tempQueue.isEmpty()) {
    stack.push(tempQueue.poll());
}

// Compare the queue with the stack
for (int item : queue) {
    if (item != stack.pop()) {
        return false;
    }
}
return true;
}
```

- **Time Complexity:** $O(n)$, where n is the number of elements in the queue.

3. Implement a Circular Queue using Arrays

Problem:

Design a circular queue using a fixed-size array with `enqueue`, `dequeue`, `isFull`, and `isEmpty` operations.

Example:

- **Input:**
 - `enqueue(10)`, `enqueue(20)`, `dequeue()`, `enqueue(30)`
- **Output:** Circular queue behaves like a regular queue but wraps around when reaching the array's end.

Code:

java

```
class CircularQueue {
    int[] arr;
    int front, rear, size, capacity;

    public CircularQueue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = size = 0;
        rear = -1;
    }

    public void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = x;
        size++;
    }

    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        front = front % capacity;
        size--;
        return arr[front];
    }
}
```

```

    }
    int item = arr[front];
    front = (front + 1) % capacity;
    size--;
    return item;
}

public boolean isFull() {
    return size == capacity;
}

public boolean isEmpty() {
    return size == 0;
}
}

```

- **Time Complexity:**

- `enqueue`: $O(1)$
- `dequeue`: $O(1)$
- `isFull`: $O(1)$
- `isEmpty`: $O(1)$

4. Implement a Queue using a Linked List

Problem:

Design a queue using a singly linked list, where each element is a node.

Example:

- **Input:** `enqueue(10)`, `enqueue(20)`, `dequeue()`
- **Output:** [20]

Code:

java

```

class LinkedListQueue {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node front, rear;

    public LinkedListQueue() {
        this.front = this.rear = null;
    }

    public void enqueue(int x) {
        Node newNode = new Node(x);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }
}

```

```

    }

    public int dequeue() {
        if (front == null) {
            System.out.println("Queue is empty");
            return -1;
        }
        int item = front.data;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        return item;
    }

    public boolean isEmpty() {
        return front == null;
    }
}

```

- **Time Complexity:** $O(1)$ for both `enqueue` and `dequeue`.

I'll continue with more problems if you'd like!



give remaining



Let's continue with the remaining queue problems.

5. Reverse the First K Elements of a Queue

Problem:

Given a queue, reverse the first K elements while keeping the order of the remaining elements unchanged.

Example:

- **Input:** Queue = [10, 20, 30, 40, 50], K = 3
- **Output:** [30, 20, 10, 40, 50]

Code:

java

```

import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class ReverseKElements {
    public static void reverseFirstK(Queue<Integer> queue, int k) {
        if (queue.isEmpty() || k > queue.size() || k <= 0) {
            return;
        }
    }
}

```

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

// Push the first K elements into the stack
for (int i = 0; i < k; i++) {
    stack.push(queue.poll());
}

// Enqueue back the stack elements (which are now reversed)
while (!stack.isEmpty()) {
    queue.add(stack.pop());
}

// Add the remaining elements back to the queue
int size = queue.size();
for (int i = 0; i < size - k; i++) {
    queue.add(queue.poll());
}
}
```

- **Time Complexity:** $O(K)$ for reversing the first K elements and $O(n - K)$ for maintaining the rest, so overall $O(n)$.

6. Sort a Queue using Recursion

Problem:

Sort a queue in ascending order using recursion without using any additional data structures.

Example:

- **Input:** Queue = [3, 1, 4, 2]
- **Output:** [1, 2, 3, 4]

Code:

java

```
import java.util.LinkedList;
import java.util.Queue;

public class SortQueueRecursion {
    public static void sortQueue(Queue<Integer> queue) {
        if (queue.isEmpty()) {
            return;
        }

        // Remove the first element
        int min = queue.poll();

        // Recursively sort the remaining queue
        sortQueue(queue);

        // Insert the first element back in the sorted position
        insertInSortedOrder(queue, min);
    }

    private static void insertInSortedOrder(Queue<Integer> queue, int element) {
        if (queue.isEmpty() || element <= queue.peek()) {
            queue.add(element);
            return;
        }
    }
}
```

```

    }
    int front = queue.poll();
    insertInSortedOrder(queue, element);
    queue.add(front);
}
}

```

- **Time Complexity:** $O(n^2)$ due to recursion and insertion.

7. Implement a Queue using Two Stacks

Problem:

Design a queue using two stacks where enqueue and dequeue operations are optimized.

Example:

- **Input:** `enqueue(1)`, `enqueue(2)`, `dequeue()`
- **Output:** 1

Code:

```

java

import java.util.Stack;

public class QueueUsingStack {
    Stack<Integer> stack1 = new Stack<>();
    Stack<Integer> stack2 = new Stack<>();

    public void enqueue(int x) {
        stack1.push(x);
    }

    public int dequeue() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.isEmpty() ? -1 : stack2.pop();
    }
}

```

- **Time Complexity:** $O(1)$ for `enqueue`, $O(n)$ for `dequeue` (in worst case).

8. Find the Maximum Element in a Sliding Window of Size K

Problem:

Given an array, find the maximum element in each sliding window of size K.

Example:

- **Input:** arr = [1, 3, -1, -3, 5, 3, 6, 7], K = 3
- **Output:** [3, 3, 5, 5, 6, 7]

Code:

java

```
import java.util.Deque;
import java.util.LinkedList;

public class SlidingWindowMaximum {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || nums.length == 0) return new int[0];

        int[] result = new int[nums.length - k + 1];
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < nums.length; i++) {
            // Remove elements not in the window
            if (!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }

            // Remove smaller elements as they are useless
            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }

            // Add the current element
            deque.offer(i);

            // Get the max for this window
            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peek()];
            }
        }

        return result;
    }
}
```

- **Time Complexity:** O(n) due to the sliding window mechanism with the deque.

9. Generate Binary Numbers from 1 to N using a Queue

Problem:

Write a program to generate binary numbers from 1 to N using a queue.

Example:

- **Input:** N = 5
- **Output:** [1, 10, 11, 100, 101]

Code:

```
java
```

```
import java.util.LinkedList;
import java.util.Queue;

public class BinaryNumbers {
    public static void generateBinary(int n) {
        Queue<String> queue = new LinkedList<>();
        queue.add("1");

        while (n-- > 0) {
            String front = queue.poll();
            System.out.print(front + " ");
            queue.add(front + "0");
            queue.add(front + "1");
        }
    }
}
```

- **Time Complexity:** $O(N)$, where N is the number of binary numbers generated.

10. Find the Minimum Time to Rot All Oranges (Multi-Source BFS)

Problem:

Given a matrix of oranges, where each cell represents an orange that can be fresh, rotten, or empty, find the minimum time required to rot all fresh oranges.

Example:

- **Input:**

```
css
```

```
grid = [ [2, 1, 0, 2, 1],
          [1, 0, 1, 2, 1],
          [1, 0, 0, 2, 1]
        ]
```

- **Output:** 2 (time required)

Code:

```
java
```

```
import java.util.LinkedList;
import java.util.Queue;

class Orange {
    int x, y;
    Orange(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class RotOranges {
```

```
public static int minTimeToRot(int[][] grid) {
    int rows = grid.length;
    int cols = grid[0].length;

    Queue<Orange> queue = new LinkedList<>();
    int freshOranges = 0;

    // Step 1: Add all rotten oranges to the queue and count fresh oranges
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
                queue.add(new Orange(i, j));
            } else if (grid[i][j] == 1) {
                freshOranges++;
            }
        }
    }

    // Step 2: Start BFS from rotten oranges
    int time = 0;
    int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    while (!queue.isEmpty() && freshOranges > 0) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            Orange current = queue.poll();
            for (int[] dir : directions) {
                int newX = current.x + dir[0];
                int newY = current.y + dir[1];

                if (newX >= 0 && newY >= 0 && newX < rows && newY < cols && grid[newX]
[newY] == 1) {
                    grid[newX][newY] = 2; // Rot the fresh orange
                    freshOranges--;
                    queue.add(new Orange(newX, newY));
                }
            }
        }
        time++;
    }

    return freshOranges == 0 ? time : -1;
}
```

- **Time Complexity:** $O(n * m)$ where n is the number of rows and m is the number of columns in the grid.

These are the remaining problems from your list. Let me know if you'd like to go over the advanced-level problems as well!



give remaining



Let's continue with the **advanced level** problems.

11. LRU Cache Implementation using Deque

Problem:

Implement an LRU (Least Recently Used) cache using a deque with a given cache size. When the cache exceeds the capacity, the least recently used element should be removed.

Example:

- **Input:** Cache size = 3, operations = ``put(1,1)``, ``put(2,2)``, ``get(1)``, ``put(3,3)``, ``put(4,4)``, ``get(2)``
- **Output:** ``get(1)`` returns 1, ``get(2)`` returns -1 (not found)

Code:

```
java

import java.util.Deque;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;

class LRUCache {
    private final int capacity;
    private final Map<Integer, Integer> map;
    private final Deque<Integer> deque;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.map = new HashMap<>();
        this.deque = new LinkedList<>();
    }

    public int get(int key) {
        if (!map.containsKey(key)) {
            return -1;
        }
        // Move accessed key to the front (most recently used)
        deque.remove(key);
        deque.addFirst(key);
        return map.get(key);
    }

    public void put(int key, int value) {
        if (map.containsKey(key)) {
            deque.remove(key);
        } else if (deque.size() == capacity) {
            // Remove least recently used key
            int lru = deque.removeLast();
            map.remove(lru);
        }
        deque.addFirst(key);
        map.put(key, value);
    }
}
```

- **Time Complexity:** $O(1)$ for both ``get`` and ``put`` operations since deque and hash map provide constant-time operations.

12. Design a Queue Supporting Get Min in $O(1)$

Problem:

Design a queue that supports enqueue, dequeue, and finding the minimum element, all in $O(1)$ time.

Example:

- **Input:** ``enqueue(3)`, `enqueue(1)`, `getMin()`, `dequeue()`, `getMin()```
- **Output:** After ``getMin()```, it should return 1, and after ``dequeue()```, ``getMin()``` returns 3.

Code:

java

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class MinQueue {
    Queue<Integer> queue = new LinkedList<>();
    Stack<Integer> minStack = new Stack<>();

    public void enqueue(int x) {
        queue.add(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public int dequeue() {
        if (queue.isEmpty()) {
            return -1;
        }
        int removed = queue.poll();
        if (removed == minStack.peek()) {
            minStack.pop();
        }
        return removed;
    }

    public int getMin() {
        return minStack.isEmpty() ? -1 : minStack.peek();
    }
}
```

- **Time Complexity:** $O(1)$ for all operations.

13. Sum of Minimum Elements in All Subarrays of Size K

Problem:

Given an array, find the sum of the minimum elements in all subarrays of size K.

Example:

- **Input:** `arr = [1, 3, 2, 4, 5], K = 3`
- **Output:** 6 (min elements are [1, 2, 2] and their sum is $1 + 2 + 2 = 6$)

Code:

```
java
```

```
import java.util.Deque;
import java.util.LinkedList;

public class SumOfMinInSubarrays {
    public static int sumOfMinInSubarrays(int[] arr, int k) {
        Deque<Integer> deque = new LinkedList<>();
        int sum = 0;

        for (int i = 0; i < arr.length; i++) {
            // Remove elements not in this window
            if (!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }

            // Remove all elements larger than the current element
            while (!deque.isEmpty() && arr[deque.peekLast()] >= arr[i]) {
                deque.pollLast();
            }

            deque.offer(i);

            // Add the minimum element of the window
            if (i >= k - 1) {
                sum += arr[deque.peek()];
            }
        }

        return sum;
    }
}
```

- **Time Complexity:** $O(n)$ because each element is added and removed from the deque at most once.

14. Implement a Double-Ended Queue (Deque) using Arrays

Problem:

Design a deque (double-ended queue) using a fixed-size array that supports `addFirst`, `addLast`, `removeFirst`, `removeLast`, and `peek` operations.

Example:

- **Input:** `addFirst(1)`, `addLast(2)`, `removeFirst()`, `removeLast()`
- **Output:** After `removeFirst()`, deque = [2]; after `removeLast()`, deque = []

Code:

```
java
```

```
class ArrayDeque {
    private int[] arr;
    private int front, rear, size, capacity;

    public ArrayDeque(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = -1;
    }
}
```

```

        rear = -1;
        size = 0;
    }

    public void addFirst(int x) {
        if (isFull()) {
            System.out.println("Deque is full");
            return;
        }
        front = (front == -1) ? 0 : (front - 1 + capacity) % capacity;
        arr[front] = x;
        size++;
    }

    public void addLast(int x) {
        if (isFull()) {
            System.out.println("Deque is full");
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = x;
        size++;
    }

    public int removeFirst() {
        if (isEmpty()) {
            System.out.println("Deque is empty");
            return -1;
        }
        int item = arr[front];
        front = (front + 1) % capacity;
        size--;
        return item;
    }

    public int removeLast() {
        if (isEmpty()) {
            System.out.println("Deque is empty");
            return -1;
        }
        int item = arr[rear];
        rear = (rear == 0) ? capacity - 1 : rear - 1;
        size--;
        return item;
    }

    public boolean isFull() {
        return size == capacity;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}

```

- **Time Complexity:** $O(1)$ for all operations.

15. First Negative Integer in Every Window of Size K

Problem:

Given an array, for each sliding window of size K, find the first negative integer in the window.

Example:

- **Input:** arr = [12, -1, -7, 8, -15, 30, 16, 28], K = 3

- **Output:** [-1, -1, -7, -15, -15, 0]

Code:

java

```
import java.util.LinkedList;
import java.util.Queue;

public class FirstNegativeInWindow {
    public static void firstNegativeInWindows(int[] arr, int k) {
        Queue<Integer> negatives = new LinkedList<>();
        for (int i = 0; i < arr.length; i++) {
            // Add negative numbers to the queue
            if (arr[i] < 0) {
                negatives.add(i);
            }

            // Remove out-of-bound elements
            if (!negatives.isEmpty() && negatives.peek() < i - k + 1) {
                negatives.poll();
            }

            // Print the first negative for this window
            if (i >= k - 1) {
                if (!negatives.isEmpty()) {
                    System.out.print(arr[negatives.peek()] + " ");
                } else {
                    System.out.print("0 ");
                }
            }
        }
    }
}
```

- **Time Complexity:** $O(n)$ since we process each element of the array once.

These are the remaining advanced-level problems. Let me know if you'd like further explanations or modifications!



Expert Level

Maximum of All Subarrays of Size K.

Given an array, find the maximum element of all subarrays of size K using a deque.

Sum of All Elements Between the First and Last Occurrence of a Specific Value in a Queue.

Given a queue, return the sum of all elements between the first and last occurrence of a specified value.

Kth Largest Element in a Stream of Numbers.

Implement a class that continuously adds numbers to a queue and can return the Kth largest element in constant time.

Rearrange Queue Alternately (First-Second Pattern).

Given a queue, rearrange its elements alternately (first half, then second half), e.g., 1 2 3 4 5 6 becomes 1 4 2 5 3 6.

Circular Tour Problem (Gas Station Problem).

Given a set of petrol pumps with a petrol capacity and distance to the next pump, determine if there exists a circular tour that allows completing the journey.



 **Memory updated**

Here are the explanations and solutions for the expert-level queue problems.

1. Maximum of All Subarrays of Size K

Problem:

Given an array, find the maximum element of all subarrays of size K using a deque.

Example:

- **Input:** arr = [10, 5, 2, 7, 8, 7], K = 3
- **Output:** [10, 7, 8, 8]

Code:

java

```
import java.util.Deque;
import java.util.LinkedList;

public class MaxSubarraysOfSizeK {
    public static int[] maxOfSubarrays(int[] arr, int k) {
        int n = arr.length;
        int[] result = new int[n - k + 1];
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < n; i++) {
            // Remove elements not in this window
            if (!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }

            // Remove all elements smaller than the current element
            while (!deque.isEmpty() && arr[deque.peekLast()] <= arr[i]) {
                deque.pollLast();
            }

            deque.offer(i);

            // Add the maximum element of this window
            if (i >= k - 1) {
                result[i - k + 1] = arr[deque.peek()];
            }
        }

        return result;
    }
}
```

- **Time Complexity:** $O(n)$, where n is the length of the array, since each element is added and removed from the deque at most once.

2. Sum of All Elements Between the First and Last Occurrence of a Specific Value in a Queue

Problem:

Given a queue, return the sum of all elements between the first and last occurrence of a specified value.

Example:

- **Input:** Queue = [1, 2, 3, 4, 2, 5, 2, 6], value = 2
- **Output:** 12 (sum of elements between the first and last occurrence of 2: $3 + 4 + 2 + 5 = 12$)

Code:

java

```
import java.util.LinkedList;
import java.util.Queue;

public class SumBetweenOccurrences {
    public static int sumBetweenOccurrences(Queue<Integer> queue, int value) {
        int firstIndex = -1, lastIndex = -1;
        int sum = 0, index = 0;
        Queue<Integer> tempQueue = new LinkedList<>(queue);

        while (!tempQueue.isEmpty()) {
            int element = tempQueue.poll();
            if (element == value) {
                if (firstIndex == -1) {
                    firstIndex = index;
                }
                lastIndex = index;
            }
            index++;
        }

        // Sum elements between the first and last occurrence
        if (firstIndex != -1 && lastIndex != -1 && firstIndex != lastIndex) {
            int i = 0;
            tempQueue = new LinkedList<>(queue);
            while (!tempQueue.isEmpty()) {
                int element = tempQueue.poll();
                if (i > firstIndex && i < lastIndex) {
                    sum += element;
                }
                i++;
            }
        }

        return sum;
    }
}
```

- **Time Complexity:** $O(n)$, where n is the size of the queue.

3. Kth Largest Element in a Stream of Numbers

Problem:

Implement a class that continuously adds numbers to a queue and can return the Kth largest element in constant time.

Example:

- **Input:** Add numbers 3, 10, 5, 7 to the queue, find the 2nd largest element.
- **Output:** 7 (after adding [3, 10, 5, 7], 7 is the 2nd largest element)

Code:

java

```
import java.util.PriorityQueue;

public class KthLargestElement {
    private final PriorityQueue<Integer> minHeap;
    private final int k;

    public KthLargestElement(int k) {
        this.k = k;
        this.minHeap = new PriorityQueue<>();
    }

    public void add(int num) {
        minHeap.offer(num);
        if (minHeap.size() > k) {
            minHeap.poll(); // Keep only K largest elements
        }
    }

    public int getKthLargest() {
        return minHeap.peek(); // Kth largest is the smallest element in the min heap
    }
}
```

- **Time Complexity:** $O(\log K)$ for `add()` and $O(1)$ for `getKthLargest()`.

4. Rearrange Queue Alternately (First-Second Pattern)

Problem:

Given a queue, rearrange its elements alternately (first half, then second half). For example, [1, 2, 3, 4, 5, 6] becomes [1, 4, 2, 5, 3, 6].

Example:

- **Input:** Queue = [1, 2, 3, 4, 5, 6]
- **Output:** [1, 4, 2, 5, 3, 6]

Code:

```
java
```

```
import java.util.LinkedList;
import java.util.Queue;

public class RearrangeQueue {
    public static Queue<Integer> rearrangeQueue(Queue<Integer> queue) {
        int size = queue.size();
        Queue<Integer> firstHalf = new LinkedList<>();
        Queue<Integer> secondHalf = new LinkedList<>();

        for (int i = 0; i < size / 2; i++) {
            firstHalf.add(queue.poll());
        }
        while (!queue.isEmpty()) {
            secondHalf.add(queue.poll());
        }

        Queue<Integer> result = new LinkedList<>();
        while (!firstHalf.isEmpty() || !secondHalf.isEmpty()) {
            if (!firstHalf.isEmpty()) {
                result.add(firstHalf.poll());
            }
            if (!secondHalf.isEmpty()) {
                result.add(secondHalf.poll());
            }
        }

        return result;
    }
}
```

- **Time Complexity:** $O(n)$, where n is the size of the queue.

5. Circular Tour Problem (Gas Station Problem)

Problem:

Given a set of petrol pumps with a petrol capacity and the distance to the next pump, determine if there exists a circular tour that allows completing the journey.

Example:

- **Input:**
 - Petrol: [4, 6, 7, 4]
 - Distance: [6, 5, 3, 5]
- **Output:** Start at pump 1

Code:

```
java
```

```
public class CircularTour {
    public static int findStartingPoint(int[] petrol, int[] distance) {
        int totalPetrol = 0, currentPetrol = 0;
        int start = 0;

        for (int i = 0; i < petrol.length; i++) {
            totalPetrol += petrol[i] - distance[i];
        }
    }
}
```

```

        currentPetrol += petrol[i] - distance[i];

        if (currentPetrol < 0) {
            start = i + 1;
            currentPetrol = 0;
        }

        return totalPetrol >= 0 ? start : -1;
    }
}

```

- **Time Complexity:** $O(n)$, where n is the number of petrol pumps.

These are the expert-level queue problems with their explanations and solutions. Let me know if you'd like further clarifications!

ChatGPT can make mistakes. Check important info.