Queue

Queue: Definition and Key Principle

A **Queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle, meaning the element added first will be removed first. It is similar to a real-life queue (e.g., a line at a ticket counter).

Key Operations:

1. **Enqueue**: Adding an element to the end of the queue.
2. **Dequeue**: Removing an element from the front of the queue.
3. **Peek/Front**: Getting the front element without removing it.
4. **IsEmpty**: Checking if the queue is empty.
5. **IsFull**: (For fixed-size queues) Checking if the queue is full.

Types of Queues

Queues come in different variations based on their structure and behavior. Below are the main types of queues with their descriptions and examples:

---

1. Simple Queue (Linear Queue)

- **Definition**: A basic queue that follows the **First In, First Out (FIFO)** principle. Elements are added at the rear and removed from the front.
- **Key Operations**:
  - **Enqueue**: Add an element at the rear.
  - **Dequeue**: Remove an element from the front.
- **Limitation**: Once the queue is full, no more elements can be added even if some are removed (unless implemented as circular).
- **Use Case**: Customer service ticketing system.

Example in Java:

```
Queue<Integer> queue = new LinkedList<>();
queue.add(10);  // Enqueue
queue.add(20);
System.out.println(queue.poll()); // Dequeue: 10
```

---

2. Circular Queue

- **Definition**: A queue where the rear pointer wraps around to the front when the end of the array is reached, making better use of storage.
- **Key Difference**: Unlike a simple queue, it efficiently utilizes the array by reusing freed spaces.
- **Use Case**: CPU scheduling, memory management.

Illustration:

```
Enqueue: Rear → Index wraps back to 0
Dequeue: Front → Index wraps back to 0
```

Example in Java:

```
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;
```

```java
        this.rear = -1;
        this.size = 0;
    }

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

    public int dequeue() {
        if (size == 0) {
            System.out.println("Queue is empty");
            return -1;
        }
        int data = queue[front];
        front = (front + 1) % capacity;
        size--;
        return data;
    }
}
```

---

3. Priority Queue

- **Definition**: A queue where each element has a priority. Elements with higher priority are dequeued before those with lower priority, regardless of their order of arrival.
- **Behavior**:
    - **Highest Priority**: Removed first.

    - **Same Priority**: Follow FIFO for elements of equal priority.
- **Use Case**: Task scheduling, network packet management.

Example in Java:

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(15); // Priority 1
pq.add(10); // Priority 2
pq.add(20); // Priority 0 (smallest value gets highest priori
ty)
System.out.println(pq.poll()); // Output: 10
```

---

4. Deque (Double-Ended Queue)

- **Definition**: A queue where elements can be added or removed from both ends (front and rear).
- **Variants**:
    - **Input-restricted deque**: Insertion allowed at one end, deletion allowed at both ends.
    - **Output-restricted deque**: Deletion allowed at one end, insertion allowed at both ends.
- **Use Case**: Palindrome checking, caching mechanisms (e.g., LRU Cache).

Example in Java:

```
import java.util.ArrayDeque;

ArrayDeque<Integer> deque = new ArrayDeque<>();
deque.addFirst(10); // Add at front
deque.addLast(20);  // Add at rear
```

```
System.out.println(deque.removeFirst()); // Remove from front
System.out.println(deque.removeLast());  // Remove from rear
```

5. Double Priority Queue

- **Definition**: A priority queue that allows removal of both the highest and lowest-priority elements.
- **Behavior**:
    - Removal happens based on either maximum or minimum priority.
- **Use Case**: Complex scheduling systems.

6. Concurrent Queue

- **Definition**: A thread-safe queue designed for use in multi-threaded environments.
- **Types in Java**:
    - `ConcurrentLinkedQueue`: Non-blocking queue.
    - `BlockingQueue`: Blocks when trying to enqueue/dequeue in full/empty conditions.
- **Use Case**: Producer-consumer problems in multithreading.

Example in Java:

```
import java.util.concurrent.ConcurrentLinkedQueue;

ConcurrentLinkedQueue<Integer> cq = new ConcurrentLinkedQueue
<>();
cq.add(10);
cq.add(20);
```

```
System.out.println(cq.poll()); // Output: 10
```

---

7. Double-Ended Priority Queue (DEPQ)

- **Definition**: A combination of deque and priority queue where elements can be added or removed from both ends with priority considerations.
- **Use Case**: Advanced data management systems.

Comparison Table

Type	Insertion	Deletion	Priority Supported	Thread Safety
Simple Queue	Rear	Front	No	No
Circular Queue	Rear	Front	No	No
Priority Queue	Rear	Based on priority	Yes	No
Deque	Both ends	Both ends	No	No
Concurrent Queue	Rear	Front	No	Yes

Let me know if you'd like a deep dive into any specific type!

simple queue Implementation using array

```
class SimpleQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int capacity;

    // Constructor to initialize the queue
    public SimpleQueue(int capacity) {
        this.capacity = capacity;
```

```
        this.queue = new int[capacity];
        this.front = 0;
        this.rear = -1;
    }

    // Enqueue: Add an element to the rear
    public void enqueue(int data) {
        if (rear == capacity - 1) { // Check if the queue is ful
            System.out.println("Queue is full. Cannot enqueue "
            return;
        }
        queue[++rear] = data;
        System.out.println(data + " enqueued to the queue.");
    }

    // Dequeue: Remove an element from the front
    public int dequeue() {
        if (front > rear) { // Check if the queue is empty
            System.out.println("Queue is empty. Cannot dequeue."
            return -1;
        }
        int data = queue[front++];
        System.out.println(data + " dequeued from the queue.");
        return data;
    }

    // Peek: Get the front element
    public int peek() {
        if (front > rear) { // Check if the queue is empty
            System.out.println("Queue is empty. No front element
            return -1;
        }
        return queue[front];
    }

    // Is the queue empty?
```

```java
    public boolean isEmpty() {
        return front > rear;
    }

    // Display the elements of the queue
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
            return;
        }
        System.out.print("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            System.out.print(queue[i] + " ");
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a simple queue with a capacity of 5
        SimpleQueue queue = new SimpleQueue(5);

        // Perform queue operations
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.display();

        queue.dequeue();
        queue.display();

        queue.enqueue(40);
        queue.enqueue(50);
        queue.display();
```

```
    }
}
```

Circular Queue Implementation using array

```
class CircularQueue {
    private int[] queue; // Array to hold queue elements
    private int front;   // Index of the front element
    private int rear;    // Index of the rear element
    private int size;    // Current size of the queue
    private int capacity; // Maximum capacity of the queue

    // Constructor to initialize the queue
    public CircularQueue(int capacity) {
        this.capacity = capacity;
        this.queue = new int[capacity];
        this.front = 0;
        this.rear = -1;
        this.size = 0;
    }

    // Enqueue: Add an element to the rear of the queue
    public void enqueue(int data) {
        if (isFull()) {
            System.out.println("Queue is full. Cannot enqueue "
            return;
        }
        rear = (rear + 1) % capacity; // Circular increment
        queue[rear] = data;
        size++;
        System.out.println(data + " enqueued to the queue.");
    }

    // Dequeue: Remove an element from the front of the queue
```

```
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot dequeue.'
            return -1;
        }
        int data = queue[front];
        front = (front + 1) % capacity; // Circular increment
        size--;
        System.out.println(data + " dequeued from the queue.");
        return data;
    }

    // Peek: Get the front element without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty. No front element
            return -1;
        }
        return queue[front];
    }

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

    // isFull: Check if the queue is full
    public boolean isFull() {
        return size == capacity;
    }

    // Display the elements of the queue
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
            return;
```

```
        }
        System.out.print("Queue elements: ");
        for (int i = 0; i < size; i++) {
            System.out.print(queue[(front + i) % capacity] + " "
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a circular queue with a capacity of 5
        CircularQueue queue = new CircularQueue(5);

        // Perform queue operations
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.display();

        queue.dequeue();
        queue.display();

        queue.enqueue(40);
        queue.enqueue(50);
        queue.enqueue(60); // This will be rejected as the queue
        queue.display();
    }
}
```

Implementation of a **Deque** (Double-Ended Queue) in Java

A **deque** (short for "double-ended queue") is a data structure where elements can be added or removed from both ends. It can operate as both a stack and a queue.

Key Operations in a Deque

1. **Add at the Front:** `addFirst()`
2. **Add at the Rear:** `addLast()`
3. **Remove from the Front:** `removeFirst()`
4. **Remove from the Rear:** `removeLast()`
5. **Peek Front Element:** `peekFirst()`
6. **Peek Rear Element:** `peekLast()`
7. **Check if Empty:** `isEmpty()`

```
class Deque {
    private int[] deque;
    private int front, rear, size, capacity;

    // Constructor
    public Deque(int capacity) {
        this.capacity = capacity;
        this.deque = new int[capacity];
        this.front = -1;
        this.rear = 0;
        this.size = 0;
    }

    // Add an element to the front
    public void addFirst(int data) {
        if (isFull()) {
            System.out.println("Deque is full. Cannot add " + da
            return;
        }
        if (front == -1) { // First element
            front = 0;
            rear = 0;
```

```java
        } else {
            front = (front - 1 + capacity) % capacity; // Circul
        }
        deque[front] = data;
        size++;
        System.out.println(data + " added to the front.");
    }

    // Add an element to the rear
    public void addLast(int data) {
        if (isFull()) {
            System.out.println("Deque is full. Cannot add " + da
            return;
        }
        if (front == -1) { // First element
            front = 0;
            rear = 0;
        } else {
            rear = (rear + 1) % capacity; // Circular increment
        }
        deque[rear] = data;
        size++;
        System.out.println(data + " added to the rear.");
    }

    // Remove an element from the front
    public int removeFirst() {
        if (isEmpty()) {
            System.out.println("Deque is empty. Cannot remove fr
            return -1;
        }
        int data = deque[front];
        if (front == rear) { // Single element
            front = -1;
            rear = -1;
        } else {
```

```
            front = (front + 1) % capacity; // Circular incremen
        }
        size--;
        System.out.println(data + " removed from the front.");
        return data;
    }

    // Remove an element from the rear
    public int removeLast() {
        if (isEmpty()) {
            System.out.println("Deque is empty. Cannot remove fr
            return -1;
        }
        int data = deque[rear];
        if (front == rear) { // Single element
            front = -1;
            rear = -1;
        } else {
            rear = (rear - 1 + capacity) % capacity; // Circular
        }
        size--;
        System.out.println(data + " removed from the rear.");
        return data;
    }

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

    // Check if the deque is full
    public boolean isFull() {
        return size == capacity;
    }

    // Display elements of the deque
```

```
    public void display() {
        if (isEmpty()) {
            System.out.println("Deque is empty.");
            return;
        }
        System.out.print("Deque elements: ");
        for (int i = 0; i < size; i++) {
            System.out.print(deque[(front + i) % capacity] + " "
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a deque with capacity 5
        Deque deque = new Deque(5);

        // Perform deque operations
        deque.addFirst(10);
        deque.addLast(20);
        deque.addLast(30);
        deque.display();

        deque.removeFirst();
        deque.addFirst(40);
        deque.display();

        deque.removeLast();
        deque.display();
    }
}
```

Interview Questions

Basic Questions

1. **What is a Queue?**
    - Explain the queue data structure and its FIFO principle.
2. **What are the key operations of a queue?**
    - Explain operations like enqueue, dequeue, peek, and isEmpty.
3. **What is the difference between a Queue and a Stack?**
    - Compare their behavior and use cases.
4. **What are the different types of queues?**
    - Discuss types like Simple Queue, Circular Queue, Priority Queue, and Deque.
5. **Explain the real-life use cases of a queue.**
    - Examples: Printer task scheduling, call handling, CPU scheduling, etc.

Implementation Questions

1. **Implement a queue using an array.**
    - Write a program to implement basic queue operations like enqueue, dequeue, and display.
2. **Implement a queue using a linked list.**
    - Show how to dynamically manage the queue size.
3. **Implement a circular queue using an array.**
    - Demonstrate efficient space utilization with a circular structure.
4. **How would you implement a queue using two stacks?**
    - Explain and implement both the enqueue and dequeue operations.
5. **How would you implement a stack using two queues?**
    - Write the logic for this conversion.

Intermediate Questions

1. **What is the difference between a Circular Queue and a Simple Queue?**
    - Explain their behavior and how a circular queue overcomes the limitations of a simple queue.
2. **What is a Priority Queue? How is it implemented in Java?**
    - Discuss the `PriorityQueue` class in Java and its use cases.
3. **What is a Deque, and how is it different from a Queue?**
    - Discuss operations like `addFirst`, `addLast`, `removeFirst`, and `removeLast`.
4. **How does Java's `Queue` interface work?**
    - Explain its methods (`add`, `offer`, `poll`, `peek`, etc.) and implementations like `LinkedList` and `PriorityQueue`.
5. **What is a BlockingQueue?**
    - Explain its use in multithreading and methods like `put` and `take`.
6. **What is a ConcurrentLinkedQueue?**
    - Discuss its non-blocking, thread-safe nature and real-world use cases.

Advanced Questions

1. **How would you implement a queue that supports retrieving the maximum element in O(1) time?**
    - Use an auxiliary data structure to track the maximum.
2. **How would you implement a sliding window maximum using a deque?**
    - Solve this common problem using an efficient approach with a deque.
3. **How does Breadth-First Search (BFS) use a queue?**
    - Explain BFS logic and its dependency on the queue data structure.
4. **Explain the time complexity of queue operations.**
    - Discuss the time complexity for operations like enqueue, dequeue, and peek.

5. **How is a circular queue implemented in hardware or embedded systems?**
    - Discuss its application in buffering and resource management.
6. **How would you design a task scheduler using a queue?**
    - Outline the logic to schedule tasks based on their priority or arrival order.
7. **What is the difference between a Priority Queue and a Heap?**
    - Explain their implementation and use cases.
8. **How would you design a system to handle multiple queues with different priorities?**
    - Discuss how to manage tasks efficiently in such a system.

Problem-Solving Questions

1. **Reverse the first `k` elements of a queue.**
    - Use a stack or a deque to solve the problem efficiently.
2. **Check if a sequence of enqueue and dequeue operations is valid.**
    - Validate the sequence using a stack and a queue.
3. **Generate binary numbers from 1 to N using a queue.**
    - Use a queue to generate the binary representation of numbers.
4. **Implement a cache using a queue.**
    - Use a deque to implement an LRU (Least Recently Used) cache.
5. **Given a queue, find the minimum element without sorting.**
    - Use an auxiliary queue to track the minimum.
6. **Simulate a call center system using a queue.**
    - Use multiple queues to simulate customers, agents, and priority calls.

Real-Life Use Case Questions

1. **How are queues used in task scheduling?**
    - Discuss their application in job schedulers, like CPU task management.

2. **Explain the role of queues in message brokering systems.**
   - Use examples like RabbitMQ, Kafka, or ActiveMQ.
3. **How is a queue used in load balancing?**
   - Describe its application in distributing tasks among servers.
4. **How would you implement a queue in a distributed system?**
   - Discuss challenges like consistency, fault tolerance, and synchronization.
5. **How do queues work in operating system process scheduling?**
   - Explain concepts like multi-level queue scheduling.