Queue Implementation Using Array

Generate DSA Notes for implement QUEUE using Array

Here are the **DSA Notes for implementing a Queue using an Array**:

Queue Data Structure

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle, where the element inserted first is the first to be removed.

Operations in a Queue

1. **Enqueue (Insert)**: Adds an element to the end of the queue.
2. **Dequeue (Remove)**: Removes an element from the front of the queue.
3. **Peek/Front**: Retrieves the element at the front without removing it.
4. **IsEmpty**: Checks if the queue is empty.
5. **IsFull**: Checks if the queue is full (for array-based queues).

Array Implementation of Queue

An array-based queue implementation maintains the following:

- **Front**: Index of the first element in the queue.
- **Rear**: Index of the last element in the queue.
- **Size**: Maximum capacity of the queue.

Challenges in Array Implementation

- After repeated enqueue and dequeue operations, elements may shift, and space may remain unused. To resolve this, we use **circular queues**.

Steps to Implement a Queue Using Array

1. **Define the Array**: Create an array to store elements of the queue.
2. **Initialize Front and Rear**: Initially set to `-1` to indicate an empty queue.
3. **Enqueue Operation**:
    - Check if the queue is full (`rear == size - 1` for a simple queue).
    - Increment the `rear` and insert the element at `rear`.
    - If the queue was empty, update `front` to `0`.
4. **Dequeue Operation**:
    - Check if the queue is empty (`front == -1` or `front > rear`).
    - Retrieve the element at `front` and increment `front`.
    - If `front` surpasses `rear`, reset both to `-1` (queue is empty).
5. **Peek Operation**:
    - Return the element at the `front` if the queue is not empty.
6. **Handle Edge Cases**:
    - Queue overflow and underflow.
    - Reset indices when the queue becomes empty.

Pseudocode

Initialization:

```java
class Queue {
    int[] arr;  // Array to hold queue elements
    int front, rear, size, capacity;

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

Enqueue Operation:

java

```java
void enqueue(int value) {
    if (rear == capacity - 1) {  // Queue is full
        System.out.println("Queue Overflow");
        return;
    }
    if (front == -1) front = 0; // First element
    arr[++rear] = value;
    size++;
}
```

Dequeue Operation:

java

```java
int dequeue() {
    if (front == -1 || front > rear) { // Queue is empty
        System.out.println("Queue Underflow");
        return -1;
    }
    int dequeuedValue = arr[front++];
    size--;
    if (front > rear) { // Reset queue after last element is dequeued
        front = rear = -1;
    }
    return dequeuedValue;
}
```

Peek Operation:

java

```java
int peek() {
    if (front == -1) { // Queue is empty
        System.out.println("Queue is Empty");
        return -1;
    }
    return arr[front];
}
```

IsEmpty:

java

```
boolean isEmpty() {
    return front == -1;
}
```

Circular Queue Optimization

In a circular queue:

- When `rear` reaches the last index, it wraps around to `0` (if there's space at the front).
- Formula for the next position: `(rear + 1) % capacity`.

Key Changes for Circular Queue:

1. **Enqueue:**

   java

   ```
   void enqueue(int value) {
       if ((rear + 1) % capacity == front) {  // Queue is full
           System.out.println("Queue Overflow");
           return;
       }
       if (front == -1) front = 0; // First element
       rear = (rear + 1) % capacity;
       arr[rear] = value;
       size++;
   }
   ```

2. **Dequeue:**

   java

```
int dequeue() {
    if (front == -1) { // Queue is empty
        System.out.println("Queue Underflow");
        return -1;
    }
    int dequeuedValue = arr[front];
    if (front == rear) { // Last element
        front = rear = -1;
    } else {
        front = (front + 1) % capacity;
    }
    size--;
    return dequeuedValue;
}
```

Time Complexity

- **Enqueue**: O(1)
- **Dequeue**: O(1)
- **Peek**: O(1)

Space Complexity

- **Array**: O(n), where `n` is the queue capacity.

Advantages of Array-based Queue

- Easy to implement.
- Predictable memory usage.

Disadvantages

- Fixed size (requires resizing or circular implementation for efficiency).
- Shifting elements (in a simple queue, not circular).

Let me know if you'd like to explore linked-list-based implementation for a queue or need deeper explanations for circular queues.