Implementation of Circular Queue

Mr. Vikas Kumar

Assistant Professor

Industry Embedded Program

Lesson Plan

Subject/Course	**Competitive Coding**
Lesson Title	**Implementation of Circular Queue**

Lesson Objectives
To explain the concept and purpose of a circular queue and how it differs from a linear queue.
To implement circular queue operations (enqueue, dequeue, front, rear, isFull, isEmpty) using an array-based approach in code.
Evaluate the time and space complexity of common circular queue operations.
To identify suitable scenarios in computer science or real-life where Circualr Queue is applicable and efficient.

Problem Statement:

Write a program to design a circular queue(k) which Should implement the below functions

a. Enqueue

b. Dequeue

c. Front

d. Rear

Introduction to the Problem

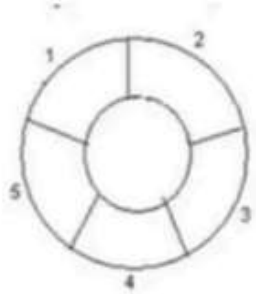
- A Circular Queue is a linear data structure that follows the FIFO (First In First Out) principle.
- Unlike a normal queue, the last position is connected back to the first position to make a circle.
- This structure efficiently utilizes memory by reusing empty spaces created by dequeued elements.

Concept and Background

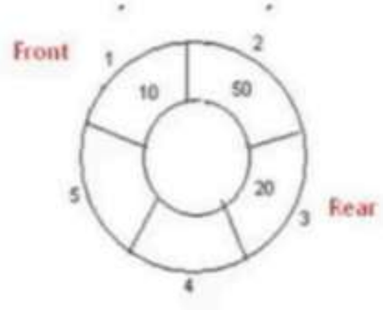
- Circular Queue connects the end of the queue back to the front.
- It prevents wastage of space in a linear queue when elements are dequeued.
- It can be implemented using Arrays, Linked List etc.
- Two pointers are maintained:
  - front → points to the first element.
  - rear → points to the last element.
- The queue is full when (rear + 1) % size == front.
- The queue is empty when front == -1.

Visualization

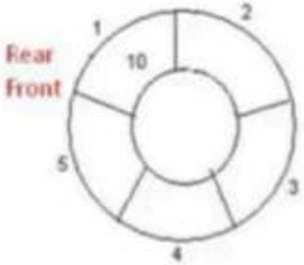
1.



4.Insert 20,Rear=3, Front =1

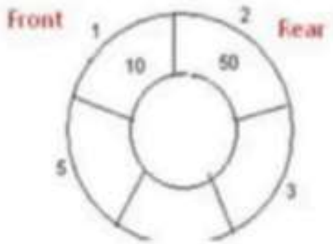


2. Insert 10, Rear = 1, Front = 1.

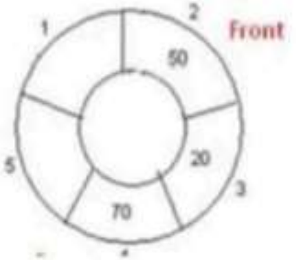


5. Insert 70, Rear = 4, Front = 1.

3. Insert 50, Rear = 2, Front = 1.



6. Delete front, Rear = 4, Front = 2.



Queue Operations

1. Enqueue (Insert): Add an element at the rear.

- Check if the queue is full.
- Update rear = (rear + 1) % size.
- Insert element at rear.

2. Dequeue (Delete): Remove an element from the front.

- Check if the queue is empty.
- Update front = (front + 1) % size.

3. Peek: Retrieve the front element without removing it.

4. Display: Print elements from front to rear circularly.

Conditions for Queue State

- **Queue Full Condition:**
  (rear + 1) % size == front

- **Queue Empty Condition:**
  front == -1

- **Reset Condition after last Dequeue:**
  If front == rear, then set front = rear = -1

Algorithm/Logic

❑ **ENQUEUE(x):**

1. If (rear + 1) % size == front → Overflow
2. Else if front == -1 → front = rear = 0
3. Else → rear = (rear + 1) % size
4. queue[rear] = x

❑ **DEQUEUE():**

1. If front == -1 → Underflow
2. x = queue[front]
3. If front == rear → front = rear = -1
4. Else → front = (front + 1) % size

Algorithm/Logic

- **Front(x):**

1. If front != -1, then queue[front] is the front item.

- **Rear():**

1. If rear != -1, then queue[rear] is the rear item.

Code Implementation

```java
public class Practical4_CircularQueue {

    static class CircularQueue {
        int[] queue;
        int front, rear, size, capacity;

        CircularQueue(int k) {
            capacity = k;
            queue = new int[k];
            front = 0;
            rear = -1;
            size = 0;
        }

        boolean enqueue(int value) {
            if (isFull()) return false;
            rear = (rear + 1) % capacity;
            queue[rear] = value;
            size++;
            return true;
        }
```

```java
        boolean dequeue() {
            if (isEmpty()) return false;
            front = (front + 1) % capacity;
            size--;
            return true;
        }

        int Front() { return isEmpty() ? -1 : queue[front]; }
        int Rear() { return isEmpty() ? -1 : queue[rear]; }
        boolean isEmpty() { return size == 0; }
        boolean isFull() { return size == capacity; }
    }

    public static void main(String[] args) {
        CircularQueue q = new CircularQueue(3);
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        System.out.println("Front Element: " + q.Front());
        q.dequeue();
        q.enqueue(40);
        System.out.println("Rear Element: " + q.Rear());
    }
}
```

Output :

```
Front Element: 10

Rear Element: 40
```

Output

```
Front: 10
Rear: 70
Deleted: 10
Front: 50
Rear: 70
Queue Elements: 50 20 70
```

Time & Space Complexity

- Enqueue Operation: O(1)
- Dequeue Operation: O(1)
- Peek Operation: O(1)
- Display Operation: O(n)
- Space Complexity: O(n)

Circular Queue provides efficient memory use and constant-time operations.

Summary

- Implemented a Circular Queue of capacity k supporting operations: enqueue, dequeue, front, and rear.
- Used modular arithmetic to connect the last index back to the first, avoiding wasted space.
- Handled overflow (queue full) and underflow (queue empty) conditions effectively.
- Demonstrated the difference between linear queue and circular queue, emphasizing better memory utilization.
- Strengthened understanding of queue data structure, front/rear pointers, and FIFO principle in data handling.

Practice Questions:

1.Design Circular Deque -- LeetCode#641
https://leetcode.com/problems/design-circular-deque/

2.Find the Winner of the Circular Game -- LeetCode#1823
https://leetcode.com/problems/find-the-winner-of-the-circular-game/

Thanks