Day 24: Circular Queue

Gati Goyal

"Queues are always first in, first out. But circular queues bring that back to the start when it reaches the end."
— Anonymous

---

1 Introduction

A **Queue** is a linear data structure that follows the **First In First Out (FIFO)** principle, meaning the first element inserted into the queue is the first one to be removed. A circular queue is a variation of a normal queue that overcomes the problem of wasted space in a regular queue. In a circular queue, when the rear of the queue reaches the end of the array, it wraps around to the front of the array.

This is achieved using the modulo operator, which allows us to access elements circularly by wrapping around when necessary.

2 Normal Queue vs Circular Queue

The main difference between a normal queue and a circular queue lies in how the rear pointer is handled when the queue is full:

- **Normal Queue:**
  - If the queue is full and an element needs to be added, it results in an overflow even if there is space at the front due to the queue being linear.
- **Circular Queue:**
  - The rear pointer wraps around to the front when it reaches the end of the array, allowing for efficient use of space.

3 Circular Queue Operations

The following operations can be performed on a circular queue:

- **Enqueue:** Insert an element into the queue.
- **Dequeue:** Remove an element from the front of the queue.

- **Front:** Get the element at the front of the queue without removing it.
- **IsEmpty:** Check whether the queue is empty.
- **IsFull:** Check whether the queue is full.

The **Modulo Operator** is used for circular indexing. The rear and front pointers are incremented modulo the queue size, ensuring they wrap around when they reach the end of the array.

4 Code Implementation

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 5 // Maximum size of the circular queue

// Define the Circular Queue structure
struct CircularQueue {
    int front, rear;
    int data[MAX];
};

// Function to initialize the queue
void initializeQueue(struct CircularQueue* queue) {
    queue->front = -1;
    queue->rear = -1;
}

// Function to check if the queue is full
int isFull(struct CircularQueue* queue) {
    return (queue->rear + 1) % MAX == queue->front;
}

// Function to check if the queue is empty
int isEmpty(struct CircularQueue* queue) {
    return queue->front == -1;
}

// Function to enqueue an element
void enqueue(struct CircularQueue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue %d.\n", value);
        return;
    }

    if (isEmpty(queue)) {
        queue->front = 0; // If the queue is empty, set front to 0
    }

```

```
    queue->rear = (queue->rear + 1) % MAX; // Move rear to the next position
    queue->data[queue->rear] = value; // Insert the element
    printf("Enqueued %d\n", value);
}

// Function to dequeue an element
int dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    int dequeuedValue = queue->data[queue->front];

    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1; // Queue is empty now
    } else {
        queue->front = (queue->front + 1) % MAX; // Move front to the next position
    }

    printf("Dequeued %d\n", dequeuedValue);
    return dequeuedValue;
}

// Function to print the queue
void printQueue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

    int i = queue->front;
    printf("Queue: ");
    while (i != queue->rear) {
        printf("%d ", queue->data[i]);
        i = (i + 1) % MAX;
    }
    printf("%d\n", queue->data[queue->rear]);
}

int main() {
    struct CircularQueue queue;
    initializeQueue(&queue); // Initialize the queue

    // Enqueue elements
    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);
    enqueue(&queue, 40);
```

```
    enqueue(&queue, 50); // At this point, the queue is full

    // Try to enqueue into a full queue
    enqueue(&queue, 60);

    // Print the current queue
    printQueue(&queue);

    // Dequeue elements
    dequeue(&queue);
    dequeue(&queue);

    // Print the queue after dequeue operations
    printQueue(&queue);

    // Enqueue more elements
    enqueue(&queue, 60);
    enqueue(&queue, 70);

    // Print the final queue
    printQueue(&queue);

    return 0;
}
```

5 Output of Circular Queue

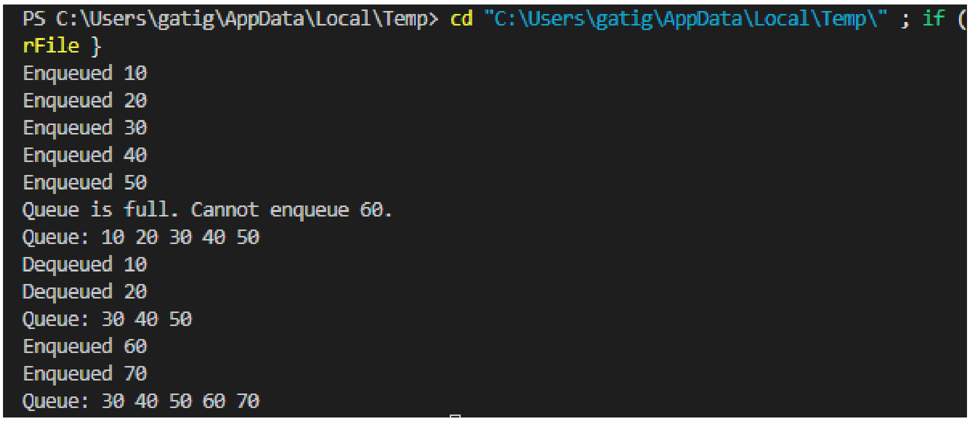

Figure 1: Circular Enqueue

6 Conclusion

The circular queue offers a solution to the problem of wasted space in a normal queue. By using modulo arithmetic, it allows efficient use of the array, with the rear pointer wrapping around when it reaches the end. This ensures that a queue can operate efficiently even when there are empty slots at the beginning of the array.