Day 21: Implement Queue

Gati Goyal

"Queue: A way to handle the first task first."
— Anonymous

---

1 Introduction

A **Queue** is a linear data structure that follows the **First In First Out (FIFO)** principle. The first element inserted into the queue is the first one to be removed. It supports the following operations:

- **Enqueue:** Add an element to the rear of the queue.
- **Dequeue:** Remove the front element of the queue.
- **Peek/Front:** View the front element without removing it.

2 Applications of Queue

- Scheduling tasks in operating systems.
- Buffering data in streaming services.
- Managing requests in a web server.

3 Code

```c
#include <stdio.h>

#define MAX 100

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

// Initialize the queue
```

```c
void initQueue(Queue *queue) {
    queue->front = -1;
    queue->rear = -1;
}

// Enqueue operation (pass by reference to modify the original queue)
void enqueue(Queue *queue, int value) {
    if (queue->rear == MAX - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (queue->front == -1) {
        queue->front = 0;
    }
    queue->data[++queue->rear] = value;

    // Display the queue after the enqueue
    printf("Queue after enqueue: ");
    for (int i = queue->front; i <= queue->rear; i++) {
        printf("%d ", queue->data[i]);
    }
    printf("\n");
}

// Dequeue operation (pass by reference to modify the original queue)
int dequeue(Queue *queue) {
    if (queue->front == -1) {
        printf("Queue Underflow\n");
        return -1;
    }
    int dequeuedValue = queue->data[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front++;
    }

    // Display the queue after the dequeue
    printf("Queue after dequeue: ");
    for (int i = queue->front; i <= queue->rear; i++) {
        printf("%d ", queue->data[i]);
    }
    printf("\n");

    return dequeuedValue;
}

// Main function to test queue operations
int main() {
```

```c
    Queue queue;
    initQueue(&queue); // Initialize the queue

    // Enqueue elements into the queue
    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);

    // Dequeue elements from the queue
    printf("Dequeued: %d\n", dequeue(&queue)); // Should print 10
    printf("Dequeued: %d\n", dequeue(&queue)); // Should print 20
    printf("Dequeued: %d\n", dequeue(&queue)); // Should print 30

    // Try dequeuing from an empty queue
    printf("Dequeued: %d\n", dequeue(&queue)); // Should print "Queue Underflow"

    return 0;
}
```

4 Queue operations: Visual Representation and Output

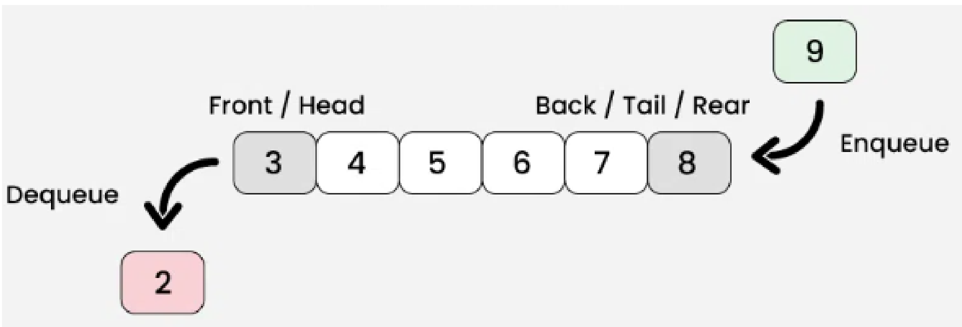

Figure 1: Queue Operations: Enqueue and Dequeue

5 Conclusion

The queue data structure is crucial in applications where elements need to be processed in the order they arrive, such as in task scheduling and buffer management. It is a simple yet highly effective tool for implementing FIFO logic.

```
PS C:\Users\gatig\AppData\Local\Temp> cd "C:\Users\gatig\AppData\Local\Temp\" ; if (
rFile }
Queue after enqueue: 10
Queue after enqueue: 10 20
Queue after enqueue: 10 20 30
Queue after dequeue: 20 30
Dequeued: 10
Queue after dequeue: 30
Dequeued: 20
Queue after dequeue: -1
Dequeued: 30
Queue Underflow
Dequeued: -1
```

Figure 2: Program Output for Queue