Lecture Notes on Queue ADT and Its Implementations

1. The Queue ADT - Definition and Operations

A Queue is a linear data structure that follows the First In First Out (FIFO) principle. Elements are inserted from the rear and deleted from the front.

Basic Operations

- **enqueue(x)**: Insert element `x` at the rear.
- **dequeue()**: Remove and return the element from the front.
- **peek()**: View the front element without removing it.
- **isEmpty()**: Check if the queue is empty.
- **isFull()**: Check if the queue is full (for fixed-size arrays).

2. Array-based Queue Implementation (Simple Array)

```
#include<iostream>
using namespace std;

#define MAX 100

struct Queue {
    int arr[MAX];
    int front = 0;
    int rear = -1;
};

bool isEmpty(Queue &q) {
    return q.front > q.rear;
}

bool isFull(Queue &q) {
    return q.rear == MAX - 1;
```

```
}

void enqueue(Queue &q, int val) {
    if (isFull(q)) {
        cout << "Queue Overflow\n";
        return;
    }
    q.arr[++q.rear] = val;
}

int dequeue(Queue &q) {
    if (isEmpty(q)) {
        cout << "Queue Underflow\n";
        return -1;
    }
    return q.arr[q.front++];
}
```

3. Circular Queue

A Circular Queue connects the end of the array back to the beginning, allowing reuse of freed space.

Problem with Normal Array-Based Queue

In a simple queue implemented with a fixed-size array, once the `rear` pointer reaches the end of the array, no more insertions can be done—even if there is space at the beginning (due to dequeued elements).

Example: Normal Queue Limitation

Suppose we have an array of size 5:

```
[10, 20, 30, 40, 50]
front = 0, rear = 4
```

Now we perform 3 dequeues:

```
[ _, _, _, 40, 50]
front = 3, rear = 4
```

Even though there is space at the beginning, we **cannot insert** more elements because `rear = SIZE - 1`.

Circular Queue: The Solution

A **Circular Queue** solves this problem by connecting the last index back to the first index, forming a logical circle. If there's empty space at the beginning, we can insert there once `rear` reaches the end.

How It Works:

We use the following formula to update positions:

```
rear = (rear + 1) % SIZE
front = (front + 1) % SIZE
```

Example: Circular Queue in Action

Let's take the same array of size 5.

1. Insert 5 elements:

   ```
   [10, 20, 30, 40, 50]
   front = 0, rear = 4
   ```

2. Dequeue 3 elements:

   ```
   [_, _, _, 40, 50]
   front = 3, rear = 4
   ```

3. Insert 2 new elements (using circular property):

   ```
   [60, 70, _, 40, 50]
   front = 3, rear = 1
   ```

   Note: The new values 60 and 70 are inserted at positions 0 and 1 due to the modulo operation.

Advantages of Circular Queue

- Efficient utilization of memory.
- No need to shift elements as in a normal array queue.
- Ideal for buffers, real-time data streams, and operating system queues.

```
#include<iostream>
using namespace std;

#define SIZE 5

struct CircularQueue {
    int arr[SIZE];
    int front = -1;
    int rear = -1;
};
```

```
bool isEmpty(CircularQueue &q) {
    return q.front == -1;
}

bool isFull(CircularQueue &q) {
    return (q.rear + 1) % SIZE == q.front;
}

void enqueue(CircularQueue &q, int val) {
    if (isFull(q)) {
        cout << "Queue Full\n";
        return;
    }
    if (isEmpty(q)) q.front = 0;
    q.rear = (q.rear + 1) % SIZE;
    q.arr[q.rear] = val;
}

int dequeue(CircularQueue &q) {
    if (isEmpty(q)) {
        cout << "Queue Empty\n";
        return -1;
    }
    int val = q.arr[q.front];
    if (q.front == q.rear) q.front = q.rear = -1;
    else q.front = (q.front + 1) % SIZE;
    return val;
}
```

4. Priority Queue

A Priority Queue stores elements such that the element with the highest (or lowest) priority is removed first.

4.1 Ascending Priority Queue

Lower values have higher priority.

```
#include<iostream>
using namespace std;

#define SIZE 100

struct PriorityQueue {
    int arr[SIZE];
    int count = 0;
};
```

```
void enqueue(PriorityQueue &pq, int val) {
    int i = pq.count - 1;
    while (i >= 0 && pq.arr[i] > val) {
        pq.arr[i + 1] = pq.arr[i];
        i--;
    }
    pq.arr[i + 1] = val;
    pq.count++;
}

int dequeue(PriorityQueue &pq) {
    if (pq.count == 0) {
        cout << "Queue Empty\n";
        return -1;
    }
    return pq.arr[--pq.count];
}
```

4.2 Descending Priority Queue

Higher values have higher priority.

```
void enqueueDescending(PriorityQueue &pq, int val) {
    int i = pq.count - 1;
    while (i >= 0 && pq.arr[i] < val) {
        pq.arr[i + 1] = pq.arr[i];
        i--;
    }
    pq.arr[i + 1] = val;
    pq.count++;
}

int dequeueDescending(PriorityQueue &pq) {
    if (pq.count == 0) {
        cout << "Queue Empty\n";
        return -1;
    }
    return pq.arr[--pq.count];
}
```

5. Dynamic Array-Based Queue

This version uses heap allocation for the queue array.

```
#include<iostream>
using namespace std;

struct DynamicQueue {
```

```
    int* arr;
    int front;
    int rear;
    int capacity;
};

void initQueue(DynamicQueue &q, int size) {
    q.arr = new int[size];
    q.capacity = size;
    q.front = 0;
    q.rear = -1;
}

bool isEmpty(DynamicQueue &q) {
    return q.front > q.rear;
}

void enqueue(DynamicQueue &q, int val) {
    if (q.rear == q.capacity - 1) {
        cout << "Queue Full\n";
        return;
    }
    q.arr[++q.rear] = val;
}

int dequeue(DynamicQueue &q) {
    if (isEmpty(q)) {
        cout << "Queue Empty\n";
        return -1;
    }
    return q.arr[q.front++];
}
```