Queues

This C++ file is a fantastic guide to the **Queue data structure**, famous for its **First-In, First-Out (FIFO)** behavior, just like a real-world checkout line. The file is expertly structured to demonstrate three distinct and important implementations, allowing for a clear comparison of their strengths and weaknesses.

1. **Simple Array Implementation:** It starts with the most basic approach, using a standard array and two indices, front and rear. While easy to understand, this version has a major flaw: it can quickly run out of space even when the queue is not full, leading to wasted memory.
2. **Circular Queue (Array):** This is the optimized array-based solution. By using the modulo operator (%), it treats the array as a circle, allowing the rear pointer to "wrap around" to the beginning. This reclaims empty space left at the start of the array and is the most common and efficient way to implement a queue with a fixed size.
3. **Linked List Implementation:** Finally, it shows the dynamic approach using a Linked List. This version is highly flexible because its size is not fixed; it can grow and shrink as needed at runtime. It's the perfect choice when you can't predict the maximum number of items the queue will need to hold.

**Queue using 2 pointers**

This section implements a basic Queue using a standard array. It uses two integer indices, front and rear, to keep track of the start and end of the queue. While simple, this implementation is inefficient with memory.

* q->front = q->rear = -1; This initializes both front and rear to **-1**. This is a common way to signify that the queue is **empty**. The first element added will set rear to index 0.
* if(q->rear == q->size-1){ printf("Queue Overflow\n"); } This is the condition to check if the queue is full. However, it has a **major flaw**: it only checks if the rear pointer has reached the end of the array. If you dequeue a few items, space opens up at the front, but this implementation can't use it. This leads to wasted space.
* q->front++; x = q->Q[q->front]; The deQueue operation doesn't actually remove data. It simply **increments the front pointer**, making the old front element inaccessible. The data remains in the array but is now considered "empty" space that, in this simple implementation, can never be reused.

**Circular Queue using Arrays**

This is the optimized array-based implementation that solves the wasted space problem of the simple queue. It treats the array as if it were a circle, allowing it to reuse empty spots at the beginning of the array. This is the standard and most efficient way to build a fixed-size queue. 🔄

* q->front = q->rear = 0; In this implementation, front and rear are initialized to **0**. The condition for an empty queue becomes front == rear.
* if ((q->rear + 1) % q->size == q->front) This is the elegant condition for checking if the queue is full. The **modulo operator (%)** is the key. It calculates where the *next* rear position would be. If that new position is the same as the current front position, it means there's no space left.
* q->rear = (q->rear + 1) % q->size; This is how an element is added (enqueue). The rear pointer is incremented, and the modulo operator makes it automatically **"wrap around"** from the end of the array back to the beginning if space is available there.
* do { ... } while (i != (q.rear + 1) % q.size); Displaying a circular queue requires a do-while loop. This ensures the loop runs at least once and correctly handles the case where the queue's elements wrap around from the end of the array to the beginning.

**Implementing Queues using Linked List**

This section shows how to build a Queue using a dynamic Linked List. This approach is highly flexible as it doesn't have a fixed size; it can grow as large as system memory allows.

* \*front = NULL, \*rear = NULL; An empty queue is represented by two global pointers, front and rear, both pointing to **NULL**. front will always point to the first node, and rear will always point to the last node for quick access.
* if(front==NULL){ front = rear = t; } This is the special case for adding an element to an **empty queue**. The newly created node t becomes both the front and the rear of the queue.
* else{ rear->next = t; rear = rear->next; } This is the standard enQueue logic. The next pointer of the current last node (rear) is updated to point to the new node t. Then, the rear pointer itself is moved forward to t. Because we always have a direct pointer to the rear, this is a very fast **O(1)** operation.
* x = front->data; t = front; front = front->next; free(t); The deQueue logic is also very efficient (**O(1)**). It retrieves the data from the front node, moves the front pointer to the next node in the chain, and then frees the memory of the original front node to prevent memory leaks.