1 Circular Queue

A **queue** is a First-In-First-Out (FIFO) data structure in which elements are inserted (*enqueued*) at the rear end and removed (*dequeued*) from the front end. A **circular queue** is a variant of a linear queue in which the last position is conceptually connected back to the first position to form a circle. This allows more efficient usage of the underlying storage.

1.1 Array-based Implementation

- Typically implemented using a fixed-size array.
- Two indices (or pointers) are maintained:
  - `front`: Points to the position where the next element will be dequeued.
  - `rear`: Points to the position where the last element was enqueued (depending on convention).

1.2 Initialization

- The queue is often initialized with `front = -1` and `rear = -1`, indicating an empty queue.

1.3 Enqueue (Insertion)

- Check if the queue is full. For a circular queue of size N, the condition for being full is:
$$(\texttt{rear} + 1) \bmod N = \texttt{front}.$$
- If not full:
  1. If $\texttt{front} = -1$, set $\texttt{front} = 0$ (first insertion).
  2. Update $\texttt{rear} = (\texttt{rear} + 1) \bmod N$.
  3. Place the new element at `queue`[`rear`].

1.4 Dequeue (Deletion)

- Check if the queue is empty, i.e., $\texttt{front} = -1$.
- If not empty:
  1. Retrieve the element at `queue`[`front`].
  2. If $\texttt{front} = \texttt{rear}$, set both `front` and `rear` to -1 (queue becomes empty).
  3. Otherwise, update $\texttt{front} = (\texttt{front} + 1) \bmod N$.

1.5 Peek

- Returns the element at the front of the queue without removing it.
- If `front` $= -1$, the queue is empty (nothing to peek).

1.6 Checking if the Queue is Full/Empty

- **Empty**: `front` $= -1$.
- **Full**: $(\texttt{rear} + 1) \bmod N = \texttt{front}$.

1.7 Key Advantage of Circular Queue

- In a standard (linear) queue using a fixed-size array, once `rear` reaches the end of the array, no more elements can be inserted even if there is space in the front.
- A circular queue "wraps around" so this free space can be reused.

2 Example: Step-by-Step Enqueue and Dequeue Operations in a Circular Queue

In this example, we have a circular queue of size $N = 5$. We denote:

$$\texttt{front} = -1, \quad \texttt{rear} = -1, \quad \text{queue} = [\_, \_, \_, \_, \_].$$

An underscore ($\_$) indicates an empty position.

2.1 Enqueue Operation (Insertion)

We will insert several elements step by step:

Enqueue 10

1. **Check if full:**
$$(\texttt{rear} + 1) \bmod N = \texttt{front}?$$
Since `front` $= -1$ and `rear` $= -1$, the queue is clearly not full.
2. **If** `front` $= -1$ **(empty queue), set** `front` $= 0$. Thus, `front` $\leftarrow 0$.
3. **Increment** `rear` **(circularly):**
$$\texttt{rear} = (\texttt{rear} + 1) \bmod N = (-1 + 1) \bmod 5 = 0.$$
4. **Insert 10 at** `queue[rear]`**:** `queue`$[0] \leftarrow 10$.
5. **Final state**:
$$\texttt{front} = 0, \quad \texttt{rear} = 0, \quad \text{queue} = [\,10, \_, \_, \_, \_].$$

Enqueue 20

1. **Check if full:**

$$(\texttt{rear} + 1) \bmod 5 = (\ 0 + 1) \bmod 5 = 1 \neq \texttt{front}(0).$$

   Not full.

2. $\texttt{front} \neq -1$, so we do not set $\texttt{front} = 0$ again.

3. $\texttt{rear} = (0 + 1) \bmod 5 = 1$.

4. $\texttt{queue}[1] \leftarrow 20$.

5. **Final state**:

$$\texttt{front} = 0, \quad \texttt{rear} = 1, \quad \text{queue} = [\,10,\ 20,\ \_,\ \_,\ \_].$$

Enqueue 30

1. **Check if full:**

$$(\ 1 + 1) \bmod 5 = 2 \neq \texttt{front}(0).$$

   Not full.

2. $\texttt{rear} = (1 + 1) \bmod 5 = 2$.

3. $\texttt{queue}[2] \leftarrow 30$.

4. **Final state**:

$$\texttt{front} = 0, \quad \texttt{rear} = 2, \quad \text{queue} = [\,10,\ 20,\ 30,\ \_,\ \_].$$

Enqueue 40, Enqueue 50

Following the same logic:

$$\texttt{front} = 0, \quad \texttt{rear} = 4, \quad \text{queue} = [\,10,\ 20,\ 30,\ 40,\ 50].$$

Now the queue is *full*, because:

$$(\texttt{rear} + 1) \bmod 5 = (4 + 1) \bmod 5 = 0,$$

which equals `front`.

2.2 Dequeue Operation (Removal)

We remove elements from the front, which currently points to the oldest inserted element.

Dequeue the first element

1. **Check if empty:** `front` $= -1$? Not empty since `front` $= 0$.
2. **Retrieve** `queue`$[0] = 10$.
3. **Update** `front`. Since `front` $\neq$ `rear`, we do:

$$\texttt{front} = (\texttt{front} + 1) \bmod 5 = (0 + 1) \bmod 5 = 1.$$

4. **Final state after removing 10**:

$$\texttt{front} = 1, \quad \texttt{rear} = 4, \quad \text{queue} = [\,10,\ 20,\ 30,\ 40,\ 50].$$

   Logically, `queue`$[0]$ is no longer in the active queue.

Dequeue the next element

1. **Check if empty:** `front` $= 1 \neq -1$. Not empty.
2. **Retrieve** `queue`$[1] = 20$.
3. **Update** `front`:

$$\texttt{front} = (1 + 1) \bmod 5 = 2.$$

4. **Final state**:

$$\texttt{front} = 2, \quad \texttt{rear} = 4, \quad \text{queue} = [\,10,\ 20,\ 30,\ 40,\ 50].$$

   Now the active queue elements are logically [30, 40, 50].

Continue dequeuing until empty

- **Dequeue 30**: `front` $\leftarrow 3$.
- **Dequeue 40**: `front` $\leftarrow 4$.
- **Dequeue 50**: Now `front` $=$ `rear` $= 4$. After removing the last element, set both:

$$\texttt{front} = -1, \quad \texttt{rear} = -1.$$

  The queue is empty again.

2.3 Wrap-Around Example

Suppose, at some point, the queue is:

$$\texttt{front} = 2, \quad \texttt{rear} = 4, \quad \text{queue} = [\,10,\ 20,\ 30,\ 40,\ 50].$$

Logically, the active elements are [30, 40, 50].

1. **Dequeue** once more (removing 30):

$$\texttt{front} = (2 + 1) \bmod 5 = 3.$$

2. **Enqueue 60**:

$$\texttt{rear} = (4 + 1) \bmod 5 = 0 \quad \Longrightarrow \quad \texttt{queue}[0] \leftarrow 60.$$

3. Now $\texttt{front} = 3$, $\texttt{rear} = 0$. The queue logically has

$$[\,40,\ 50,\ 60] \quad (\text{indices } 3, 4, 0).$$

The rear "wrapped around" to index 0 once it reached the end of the array.

2.4 Summary of Key Points

- **Enqueue check**:

$$(\texttt{rear} + 1) \bmod N = \texttt{front} \quad \Longrightarrow \quad \text{Queue is full.}$$

- **Dequeue check**:

$$\texttt{front} = -1 \quad \Longrightarrow \quad \text{Queue is empty.}$$

- When the queue is initially empty ($\texttt{front} = -1$), set $\texttt{front} = 0$ on the first enqueue.
- If after a dequeue, $\texttt{front} = \texttt{rear}$, set both to -1 to indicate the queue is now empty.
- Both `front` and `rear` wrap around using modulo arithmetic, ensuring efficient reuse of the array space.

3 Test Cases for a Circular Queue

Assume a circular queue of size $N = 5$. (You can generalize the same steps for any N.)

3.1 Test Case 1: Basic Enqueue and Dequeue

1. **Initial Condition**: `front` $= -1$, `rear` $= -1$ (empty queue).
2. **Operations**:
   - Enqueue 10
   - Enqueue 20
   - Dequeue (expected to remove 10)
   - Peek (expected to see 20)
3. **Expected Final State**:
   - After two enqueues, `front` $= 0$, `rear` $= 1$.
   - After one dequeue, `front` $= 1$, `rear` $= 1$, queue contains [20].
   - Peek shows 20.

3.2 Test Case 2: Filling the Queue to Capacity

1. **Operations**:
   - Enqueue 1, 2, 3, 4, 5
   - Attempt to enqueue 6 (should fail — queue is full)
2. **Expected Result**:
   - Queue is full after fifth enqueue: `front` $= 0$, `rear` $= 4$.
   - Attempt to enqueue 6 returns an error ("Queue is full").

3.3 Test Case 3: Dequeue from Full Queue and Then Enqueue to Test Wrap-Around

1. **Precondition**: Queue is [1, 2, 3, 4, 5] with `front` $= 0$, `rear` $= 4$.
2. **Operations**:
   - Dequeue twice (removes 1, then 2).
   - Enqueue 6 (wraps around to index 0).
   - Enqueue 7 (wraps around to index 1).
3. **Expected Result**:
   - After dequeuing 1 and 2, `front` $= 2$, `rear` $= 4$.
   - After enqueueing 6, `rear` $= 0$.
   - After enqueueing 7, `rear` $= 1$.
   - Queue is full again, effectively contains [3, 4, 5, 6, 7].

3.4 Test Case 4: Dequeue Until Empty and Check States

1. **Precondition**: Queue is full, e.g., [3, 4, 5, 6, 7], `front` = 2, `rear` = 1.
2. **Operations**:
   - Dequeue all elements one by one.
3. **Expected Result**:
   - Dequeue order: 3, 4, 5, 6, 7.
   - After final dequeue, `front` = -1 and `rear` = -1.
   - Queue is empty.

3.5 Test Case 5: Edge Conditions

- **Dequeue on empty queue**: Should indicate "Queue is empty".
- **Enqueue on full queue**: Should indicate "Queue is full".
- **Peek on empty queue**: Should indicate "Queue is empty".

3.6 Test Case 6: Randomized Operations

- Perform a random series of enqueues and dequeues (without exceeding capacity).
- Verify that the sequence of items dequeued matches the FIFO order.
- Check that `front` and `rear` update correctly.

3.7 Test Case 7: Stress Testing

- Repeatedly enqueue and dequeue a large number of elements.
- Ensure no index-out-of-bounds or infinite-loop errors.
- Confirm each operation is $\mathcal{O}(1)$ and the pointers wrap correctly.

4 Conclusion

A **circular queue** uses modular arithmetic to wrap the `front` and `rear` pointers within the bounds of the queue's size, making efficient use of storage. Properly tracking these pointers and checking for full or empty conditions is key. The test cases outlined here cover basic functionality, boundary conditions, and wrap-around behavior to ensure a robust implementation.