C.q. Intro 1

Introduction:

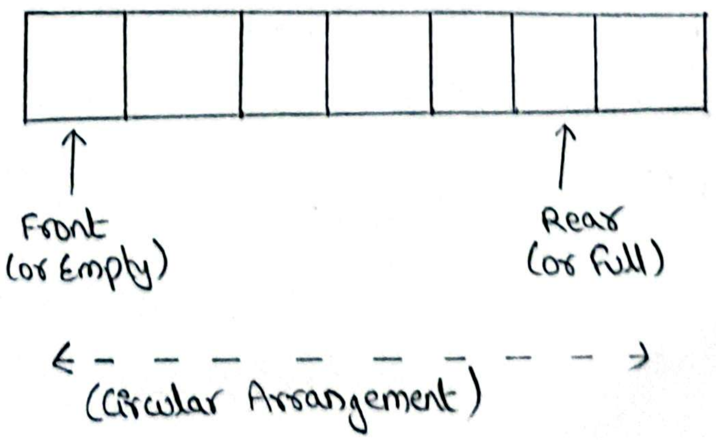
A circular queue is a linear data structure that operates on the First-In-First-Out (FIFO) principle, but with a twist: the last position is connected back to the first position, forming a circle. This design efficiently utilizes memory, especially when dealing with continuous data streams.

Key Characteristics:-

- **Circular Nature:**
  → Unlike a regular queue where the rear pointer stops at the end of the array, a circular queue wraps around. This prevents memory wastage that occurs in linear queues when elements are dequeued.
- **FIFO Principle:**
  → Elements are added at the rear and removed from the front, maintaining the first-in-first-out order.
- **Pointers:**
  → It uses two pointers:
  (a) Front: Points to the first element in the queue.
  (b) Rear: Points to the last element in the queue.
- **Efficient Memory Usage:**
  → It reuses empty spaces created by dequeuing elements, making it more memory-efficient than a standard queue.

Diagram:

Here's a conceptual diagram of a circular queue:



Operations :

→ Enqueue (Insertion):
- Adds an element to the rear of the queue.
- The rear pointer is updated to the next available position, wrapping around to the begining if necessary.
- Special attention must be paid to checking if the queue is full.

→ Dequeue (Deletion):
- Removes an element from the front of the queue.
- The front pointer is updated to the next element, wrapping around if necessary.
- Special attention must be paid to checking if the queue is empty.

Why Use a Circular Queue?

→ Buffer Management
It's ideal for implementing buffers, such as those used in operating systems or data streaming applications.

→ CPU Scheduling
Used in operating systems for CPU scheduling algorithms.

→ Traffic Management
Simulating traffic flow.

Algorithm of Circular Queue Type :-

1) Initialization (__init__):
→ Input: capacity (the maximum number of elements the queue can hold).
→ Steps:
- Create a list queue of size capacity to store the queue elements.
- Initialize front and rear pointers to -1, indicating an empty queue.

2) Check if Empty (is_empty):
→ Steps:
- Return True if front is -1, otherwise return False.

3) Check if Full (is_full):
• Steps:
→ Calculate (rear+1) % capacity.
→ Return True if the results equals front, otherwise return False. This formula effectively checks if the next position after rear (wrapping around) is front.

4) Enqueue (enqueue):
→ Input: item (the element to be added)
→ Steps:
• If queue is full (is_full is True), print an error message and return False.
• If the queue is empty (is_empty is True), set front to 0.
• Update rear to the next available position using rear = (rear+1) % capacity. This handles the circular wrapping.
• Store item in queue [rear].
• Return True.

5) Dequeue (dequeue):
→ Steps:
• If the ~~dt~~ queue is empty (is_empty is True), print an error message and return None.
• Store the element at queue [front] in item.
• If front and rear are equal (only one element in the queue), reset front and rear to -1.
• Otherwise update front to the next element using front = (front+1) % capacity.
• Return item.

6) Display (display):
→ Steps:
• If the queue is empty, print a message and return.
• Initialize a counter i to front.
• Iterate through the queue elements starting from front until i reaches rear, handling the circular wrapping using i = (i+1) % capacity.
• Print each element.