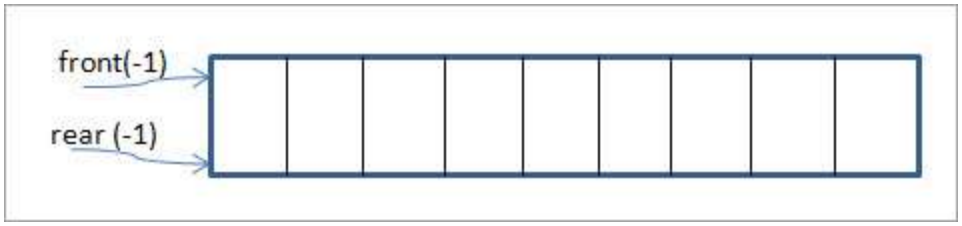
Queue

In software terms, the queue can be viewed as a set or collection of elements as shown below. The elements are arranged linearly.



We have two ends i.e. “front” and “rear” of the queue. When the queue is empty, then both the pointers are set to -1.

The “rear” end pointer is the place from where the elements are inserted in the queue. The operation of adding /inserting elements in the queue is called “enqueue”.

The “front” end pointer is the place from where the elements are removed from the queue. The operation to remove/delete elements from the queue is called “dequeue”.

When the rear pointer value is size-1, then we say that the queue is full. When the front is null, then the queue is empty.

Basic Operations

The queue data structure includes the following operations:

- **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
- **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of the queue without removing it.

Enqueue

In this process, the following steps are performed:

- Check if the queue is full.
- If full, produce overflow error and exit.
- Else, increment ‘rear’.
- Add an element to the location pointed by ‘rear’.
- Return success.

Dequeue

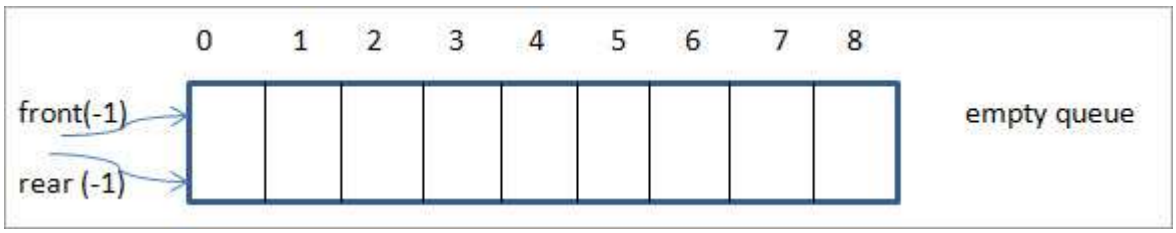
Dequeue operation consists of the following steps:

- Check if the queue is empty.
- If empty, display an underflow error and exit.
- Else, the access element is pointed out by ‘front’.

- Increment the ‘front’ to point to the next accessible data.
- Return success.

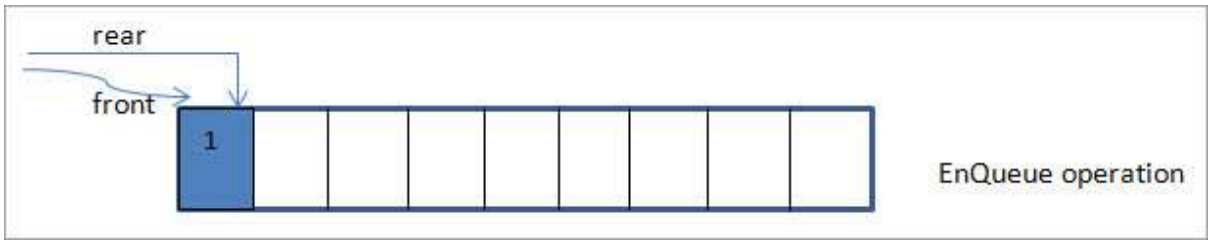
Next, we will see a detailed illustration of insertion and deletion operations in queue.

Illustration

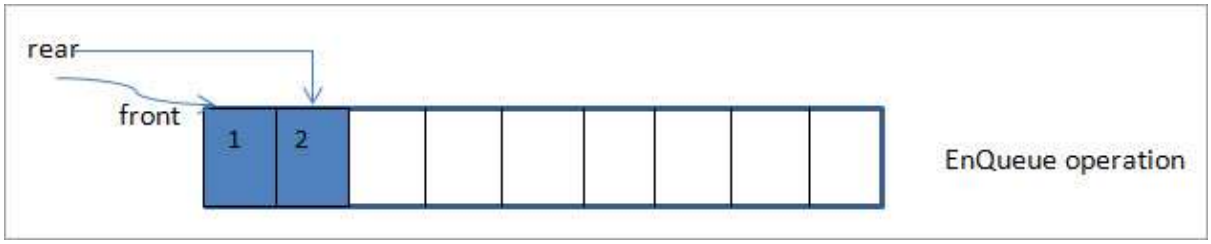


This is an empty queue and thus we have rear and empty set to -1.

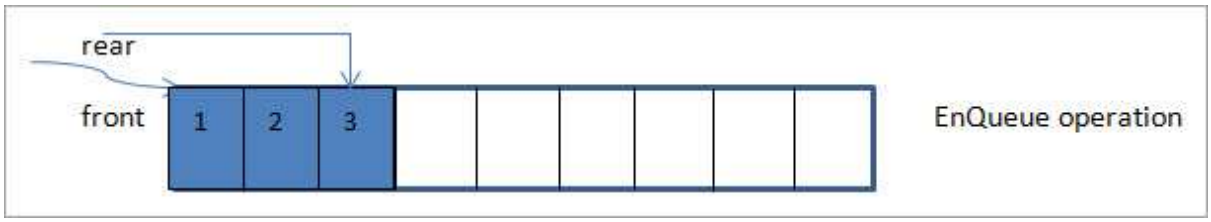
Next, we add 1 to the queue and as a result, the rear pointer moves ahead by one location.



In the next figure, we add element 2 to the queue by moving the rear pointer ahead by another increment.

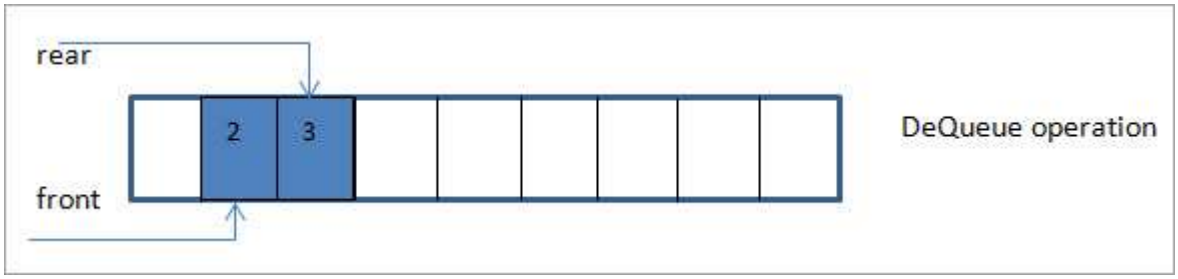


In the following figure, we add element 3 and move the rear pointer by 1.



At this point, the rear pointer has value 2 while the front pointer is at the 0^{th} location.

Next, we delete the element pointed by the front pointer. As the front pointer is at 0, the element that is deleted is 1.



Thus the first element entered in the queue i.e. 1 happens to be the first element removed from the queue. As a result, after the first dequeue, the front pointer now will be moved ahead t0 the next location which is 1.

Queue operations

Enqueue

The Enqueue operation is used to add an element to the front of the queue.

Steps of the algorithm:

1. Check if the Queue is full.
2. Set the front as 0 for the first element.
3. Increase rear by 1.
4. Add the new element at the rear index.

Dequeue

The Dequeue operation is used to remove an element from the rear of the queue.

Steps of the algorithm:

1. Check if the Queue is empty.
2. Return the value at the front index.
3. Increase front by 1.
4. Set front and rear as -1 for the last element.

Peek

The Peek operation is used to return the front most element of the queue.

Steps of the algorithm:

1. Check if the Queue is empty.
2. Return the value at the front index.

isFull
The isFull operation is used to check if the queue is full or not.

Steps of the algorithm:

1. Check if rear == MAXSIZE - 1.
2. If they are equal, return “Queue is Full.”
3. If they are not equal, return “Queue is not Full.”

isEmpty
The isEmpty operation is used to check if the queue is empty or not.

Steps of the algorithm:

1. Check if the rear and front are pointing to null memory space, i.e., -1.
2. If they are pointing to -1, return “Queue is empty.”
3. If they are not equal, return “Queue is not empty.”

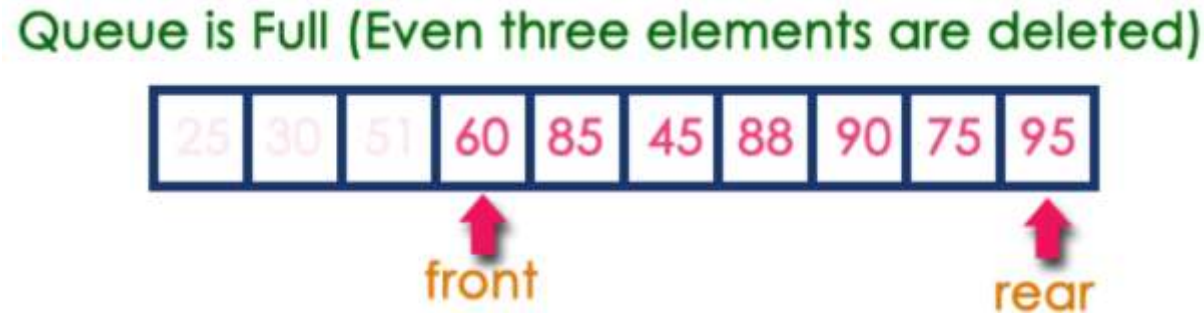
Applications of Queue

1. Printers: Queue data structure is used in printers to maintain the order of pages while printing.
2. Interrupt handling in computes: The interrupts are operated in the same order as they arrive, i.e., interrupt which comes first, will be dealt with first.
3. Process scheduling in Operating systems: Queues are used to implement round-robin scheduling algorithms in computer systems.
4. Customer service systems like ticket booking counters etc.

Circular queue

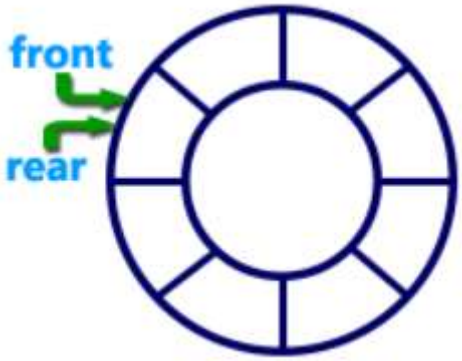
In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.

Now consider the following situation after deleting three elements from the queue...



This situation also says that Queue is Full and we cannot insert the new element because 'rear' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

A **circular queue** is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

1. Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
2. Declare all user defined functions used in circular queue implementation.
3. Create a one dimensional array with above defined SIZE (int cQueue[SIZE])
4. Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
5. Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- **Step 1 -** Check whether **queue** is **FULL**. ((**rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4 -** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue.

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5 -** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.

- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

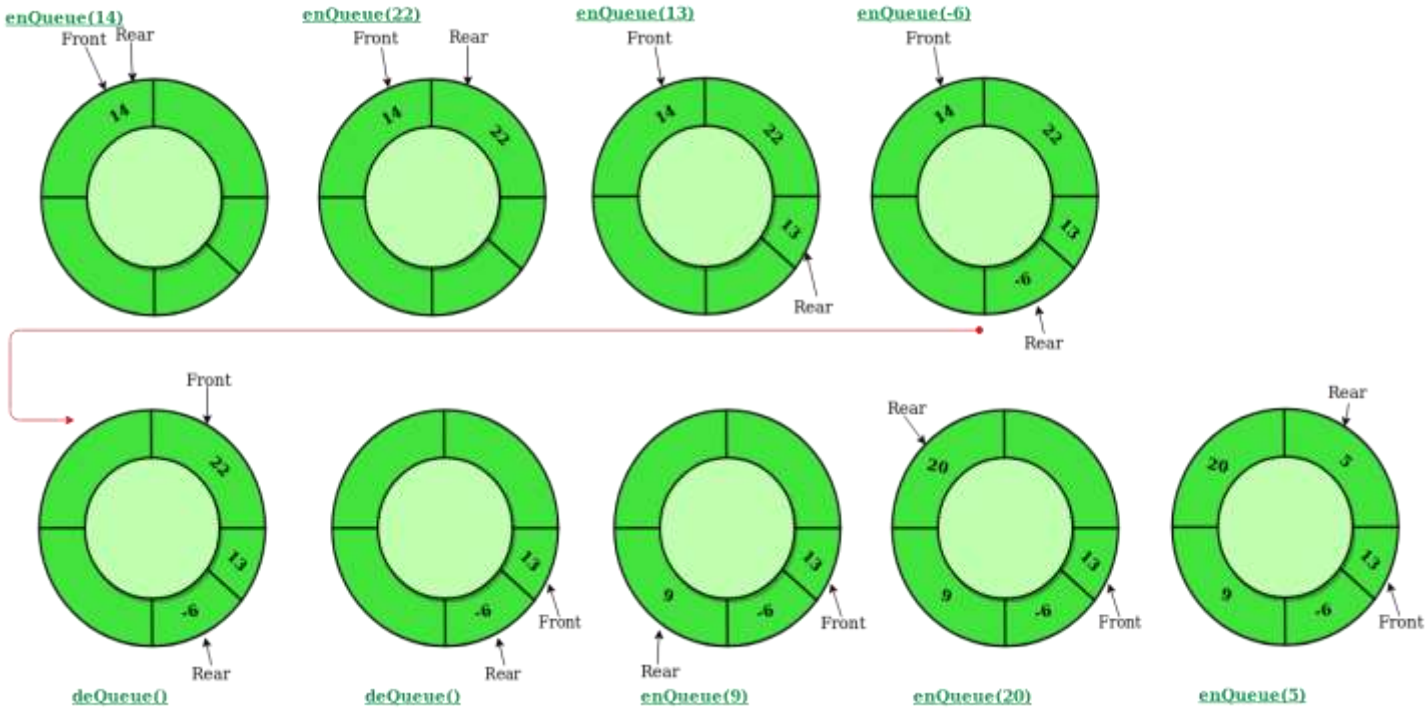


Image reference : https://www.geeksforgeeks.org/introduction-to-circular-queue/