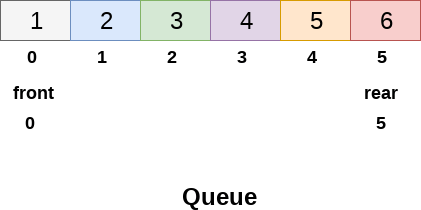
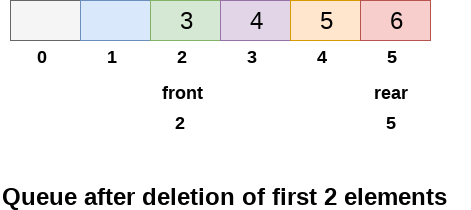
Circular Queue

Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.

Consider the queue shown in the following figure.

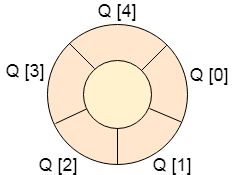
  
**Fig: Full Queue**

* The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition **rear == max - 1 becomes true**.
* However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition **rear = max -1 still holds**.
* This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.



**Queue after deleted two elements**

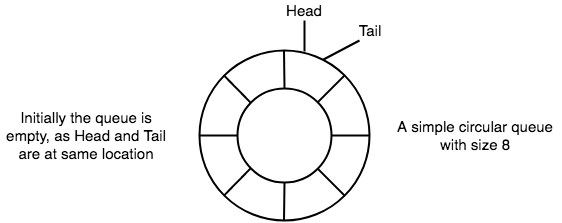
* One of the solution of this problem is circular queue.
* In the circular queue, the first index comes right after the last index.
* You can think of a circular queue as shown in the following figure.



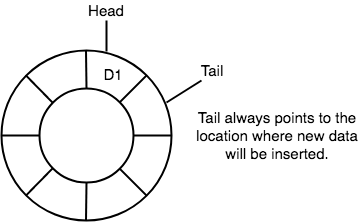
Empty Queue: Front = Rear = -1

Basic features of Circular Queue

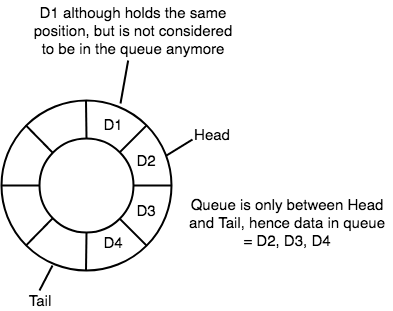
1. In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.
   * Here Initially Head = tail = -1(Empty condition)
   * First insertion Head = tail = 0, insert the element at Tail Position and Increment the tail
   * Tail is pointing to the empty position



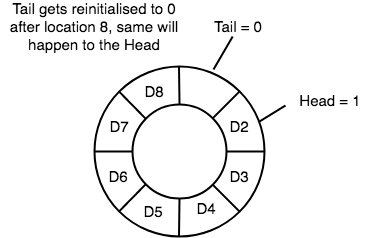
1. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.



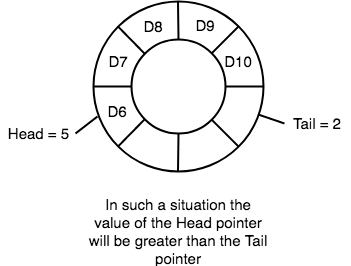
1. In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



1. The head and the tail pointer will get reinitialised to **0** every time they reach the end of the queue.



1. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialised upon reaching the end of the queue.



**Going Round and Round**

* Another very important point is keeping the value of the tail and the head pointer within the maximum queue size.
* In the diagrams above the queue has a size of 8, hence, the value of tail and head pointers will always be between 0 and 7.
* This can be controlled either by checking everytime whether tail or head have reached the maxSize and then setting the value 0 or, we have a better way, which is, for a value x if we divide it by 8, the remainder will never be greater than 8, it will always be between 0 and 8, which is exactly what we want.
* So the formula to increment the head and tail pointers to make them **go round and round** over and again will be, head = (head+1) % maxSize or tail = (tail+1) % maxSize
* There are three scenario of inserting an element in a queue.
* **If (rear + 1)%maxsize = front**, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
* **If rear != max - 1**, then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
* **If front != 0 and rear = max - 1**, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.
  + If Max = 9 (0..8) and rear =8 (insert is called)
    1. Insert on 8th position
    2. (Rear+1) mod 9 = 0….. next value of rear = 0
    3. For next move, Front = Rear = 0 so queue is full
    4. If Front = 2 (two elements are already removed so pointing the third position) two more insert rear = 2

Rear+1 %9 = 3 > Front….. so overflow

**Enqueue()**

* check if the queue is full
* for the first element, set value of FRONT to 0
* circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
* add the new element in the position pointed to by REAR

**Algorithm to insert an element in circular queue**

**Step 1:** if (rear+1)%MAX = front  
Write " OVERFLOW "  
Goto step 4  
[Endif]

**Step 2: if front = -1 and rear = -1**set front = rear = 0  
else if rear = MAX - 1 and front ! = 0  
set rear = 0  
else  
set rear = (rear + 1) % MAX  
[Endif]

**Step 3:** set queue[rear] = VAL

**Step 4:** exit

Algorithm to delete an element from a circular queue

To delete an element from the circular queue, we must check for the three following conditions.

1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm

**Step 1:** if front = -1  
Write " UNDERFLOW "  
Goto Step 4  
[END of if]

**Step 2:** set VAL = queue[front]

**Step 3:** if front = rear

set front = rear = -1  
else  
if front = MAX -1  
 set front = 0  
else

set front = front + 1

[Endif]  
[Endif]

**Step 4:** exit

**Time Complexity**

|  |  |
| --- | --- |
| **Front** | O(1) |
| **Rear** | O(1) |
| **enQueue()** | O(1) |
| **deQueue()** | O(1) |

**Application of Circular Queue**

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.
3. Multiprogramming (Time Sharing)
4. Asynchronous data transfer (file IO, pipes, sockets) [ Round Robin].
5. Waiting times of customers at call center.

**Multi-Queue Data Structures**

**Multiple queues**

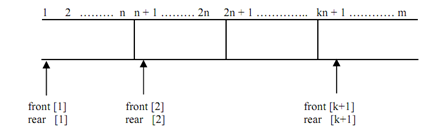
Thus far, we have seen the demonstration of a single queue, but several practical applications in computer science needs several queues.

Multi queue is data structure in which multiple queues are maintained.

This type of data structures are utilized for process scheduling.

We might use one dimensional array or multidimensional array to illustrated a multiple queue.

A multi queue implementation by using a single dimensional array along m elements is illustrated in Figure



**Figure: Multiple queues in an array**

Each of queues contains n elements that are mapped to a liner array of m elements.