Linear Queue

A queue is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, FIFO), while in a stack, as we've seen, the last item inserted is the first to be removed (LIFO). A queue works like the line you wait in.

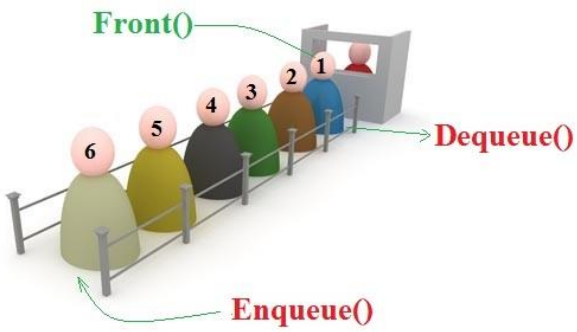
The queue abstract data type defines a collection that keeps objects in a sequence, where:

- *element access and deletion* are restricted to the *first element in the sequence*, which is called the *front of the queue*, and
- *element insertion* is restricted to the *end of the sequence*, which is called the *rear of the queue*.

In a linear queue, the **traversal** through the queue is possible only **once**,i.e.,**once an element is deleted, we cannot insert another element in its position.** This disadvantage of a linear queue is overcome by a **circular queue**, thus saving memory.

This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle. The queue supports the following two fundamental methods:

- ***Enqueue(e):*** Insert element e at the rear of the queue (end).
- ***Dequeue():*** Remove and return from the queue the object at the front; an error occurs if the queue is empty.



Additionally, similar to the case with the Stack ADT, **the queue ADT includes the following supporting methods:**

size(): Return the number of objects in the queue.

isEmpty(): Return a Boolean value that indicates whether the queue is empty.

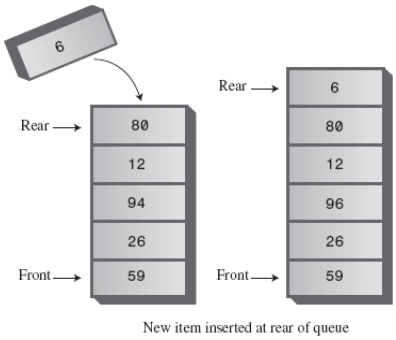
front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

Linear Queue Implementation

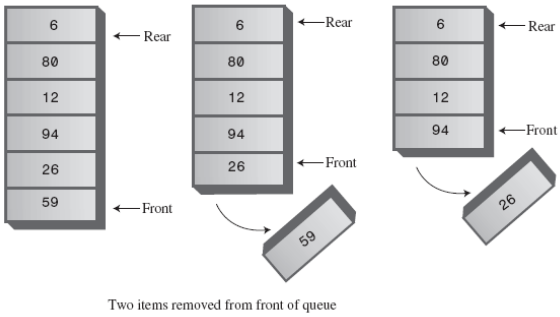
Queue can be implemented in two different ways:

1. ***Contiguous linear queue***: the queue is implemented as an **array**.
2. ***Linked linear queue***: **pointers** and **dynamic memory allocation** is used to implement the queue.

❖ A Simple Array-Based Linear Queue Implementation



New item inserted at rear of queue



Two items removed from front of queue

Operations of the Queue

Enqueue (item):

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE. ITEM is the value to be inserted.

1. If (REAR == N-1) Then *[Check for overflow]*
2. Print: Overflow
3. Else
4. If (FRONT and **REAR** = = -1) Then *[Check if QUEUE is empty]*
   (a) Set FRONT = 0
   (b) Set REAR = 0
5. Else
6. Set REAR = REAR + 1 *[Increment REAR by 1]*

   [End of Step 4 If]
7. QUEUE [REAR] = ITEM
8. Print: ITEM inserted

   [End of Step 1 If]
9. Exit

Dequeue():

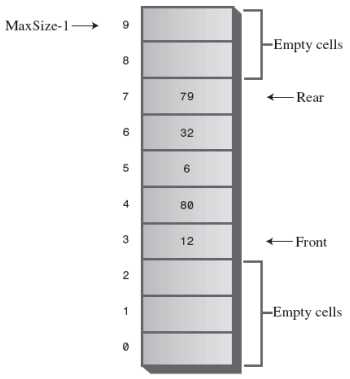
Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE.

1. If (FRONT == -1) *[Check for underflow]*
2. Print: Underflow
3. Else
4. ITEM = QUEUE [FRONT]
5. If (FRONT == REAR) Then *[Check if only one element is left]*
   (a) Set FRONT = -1 *[after delete it queue becomes empty]*
   (b) Set REAR = -1
6. Else

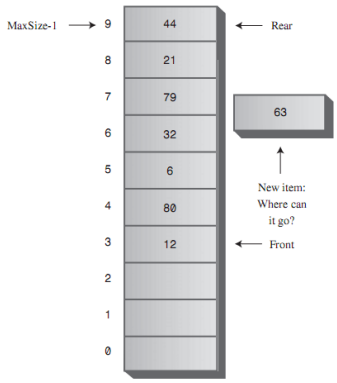
7. Set FRONT = FRONT + 1 *[Increment FRONT by 1]*
[End of Step 5 If]
8. Print: ITEM deleted
[End of Step 1 If]
9. Exit

- When you **insert** a new item in the Queue, the **rear** arrow moves upward, when you **remove** an item, **front** also moves upward.

The trouble with this arrangement is that pretty soon *the rear of the queue is at the end of the array (the highest index). Even if there are empty cells at the beginning of the array, because you've removed them, you still can't insert a new item because Rear can't go any further*. This situation is shown below.



A Queue with some items removed



Rear arrow at the end of the array

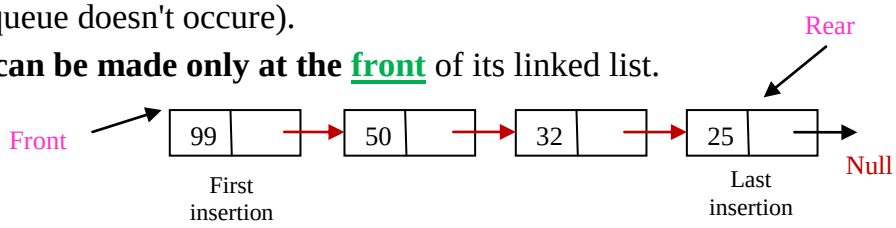
To avoid the problem of not being able to insert more items into the queue even when it's not full, **the Front and Rear arrows wrap around to the beginning of the array. The results a *circular queue* (sometimes called a *ring buffer*)**.

❖ Linked Linear Queue

The queue ADT is defined by the following operations:
Constructor : Create a new, empty queue.
Insert : Add a new item to the queue.
Remove : Remove and return an item from the queue. The item that is returned is the first one that was added.
Empty: Check whether the queue is empty.

We know that a **linear queue** is a "first in first out " data structure,i.e.,

- **Insertion (enqueue)** can be **made only at the end** of its linked list. In linked queue, there is no maximum size (full queue doesn't occure).
- **Deletion (dequeue) can be made only at the front** of its linked list.



```
public class LQueue
 {
        int data;                    // value of element
        LQueue  next;                // reference to next node

     public LQueue  (int d)          // constructor
        {
            data = d ;
        }
 }
```

Where the initial values are:

Front = null
Rear = null
Size = 0

This means a queue is empty

Front and **Rear** : *two pointers of linked queue*

Size : *the number of elements in linked queue (no. of nodes)*

- **The functions are built as bellow:**

```
public void enqueue ( )                  // insertlast to linked queue
  {
      input value to k (i.e. data to be inserted)
      LQueue   f = new LQueue (k) ;
      If  (Front = = null  and  Rear = = null)        // first insert to empty queue
        Front = f ;
        Rear = f ;
      Else                                // any insert except the first one
        Rear . next = f ;
        Rear = f ;

       size ++;
  }
////////////////////////////////////////////////
public void  dequeue ( )                  // deletefirst  from linked queue
   {
      If (Front = = null)        print   "The queue is empty" ;
      Else
        {
              item = Front . data ;
                 If  (Front = = Rear)          // There is only one element
                   Front = null ;
                   Rear= null ;
                 Else     Front = Front . next ;
```

```
            s.o.p (item) ;
            size - -;

        }
    }
```

Examples of Linear Queue Applications:

1. Queue used to model real-world situations such as people waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.
2. There are various queues quietly doing their job in your computer's (or the network's) operating system.
3. There's a printer queue where print jobs wait for the printer to be available.
4. Stores, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle transaction processing for such applications. For example, it would be a natural choice for handling calls to the reservation center of an airline.