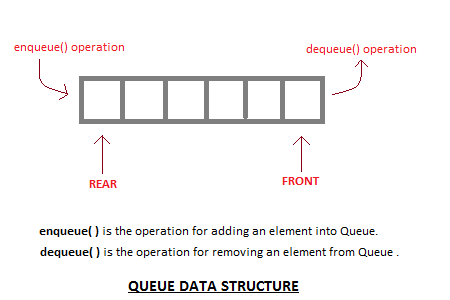
**QUEUES**

**Queue** is also an abstract data type or a linear data structure, just like [stack data structure](https://www.studytonight.com/data-structures/stack-data-structure), in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



**Basic features of Queue**

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

**Applications of Queue**

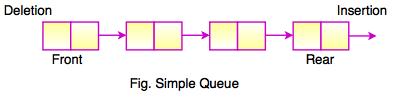
Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

**Types of Queue**

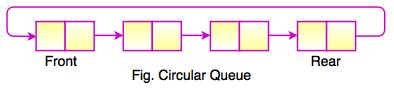
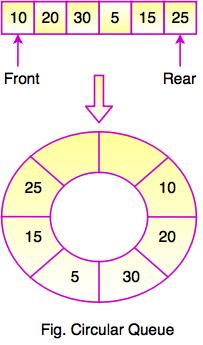
**There are four types of Queue:**  
  
1. Simple Queue or Classical Queue  
2. Circular Queue  
3. Priority Queue  
4. Dequeue (Double Ended Queue)

1. Simple Queue

Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.  
  


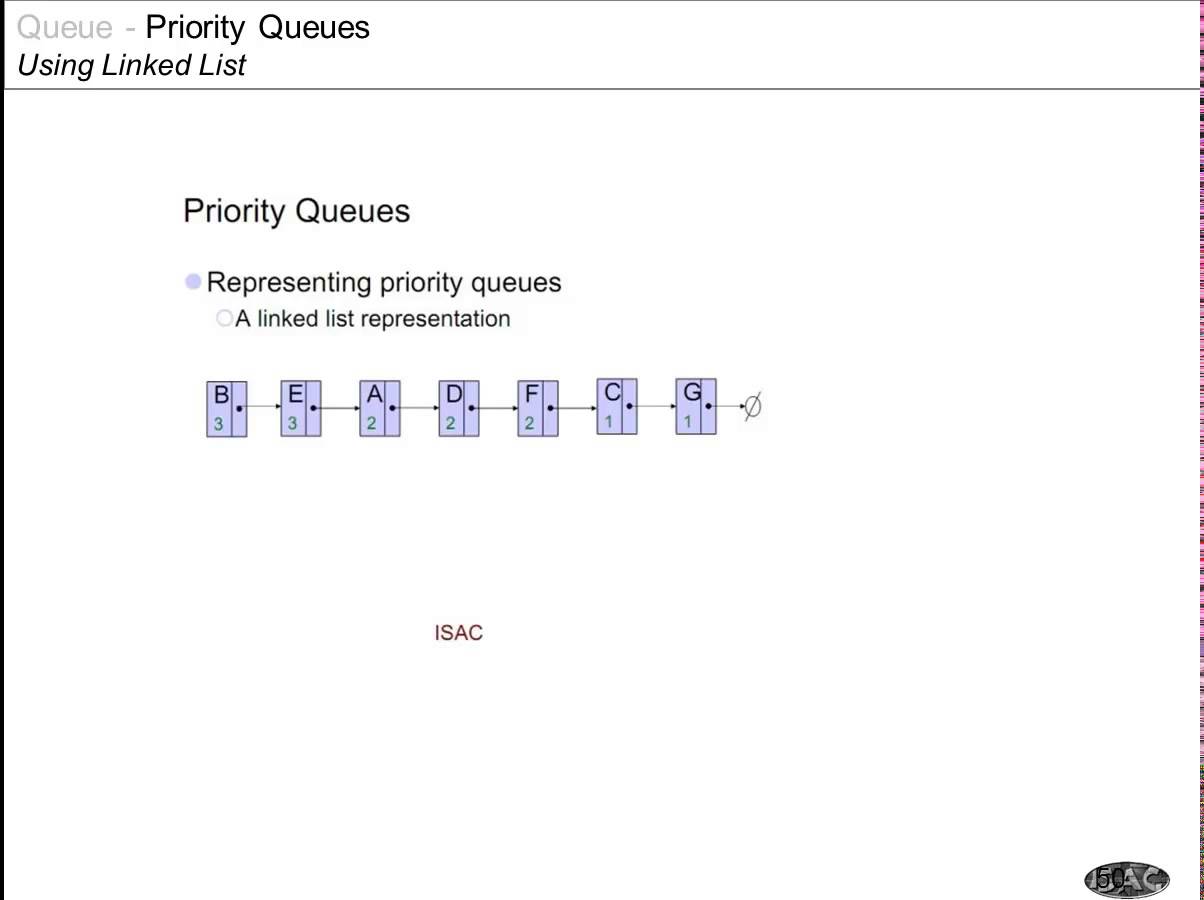
2. Circular Queue

* In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
* Circular queue is also called as**Ring Buffer.**
* It is an abstract data type.
* Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

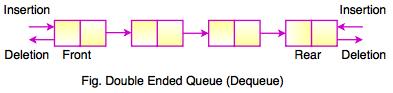
  
  
  
  
The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

3. Priority Queue

* Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is removed first.
* In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.



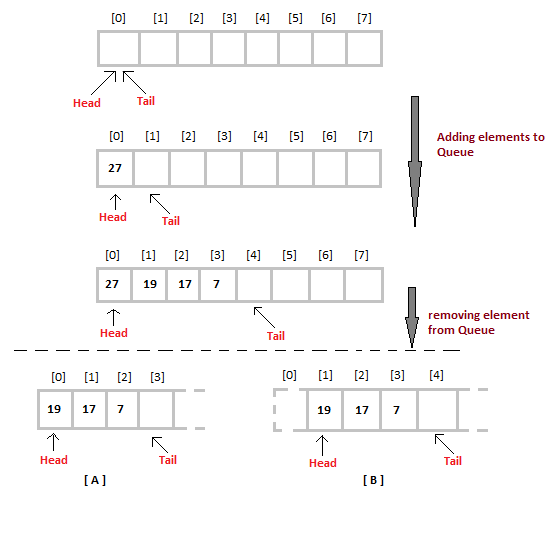
4. Dequeue (Double Ended Queue)

In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue.  
  


**Implementation of Queue Data Structure**

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

## Operations of Queue

* **enqueue()** − add (store) an item to the queue.
* **dequeue()** − remove (access) an item from the queue.
* **peek()** − Gets the element at the front of the queue without removing it.
* **isfull()** − Checks if the queue is full.
* **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

**Algorithm**

begin procedure peek

return queue[front]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return queue[front];

}

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

begin procedure isfull

if rear equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(rear == MAXSIZE - 1)

return true;

else

return false;

}

isempty()

Algorithm of isempty() function −

**Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else

return false

endif

end procedure

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

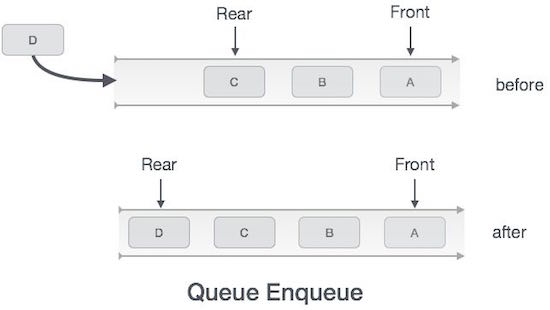
}

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

* **Step 1** − Check if the queue is full.
* **Step 2** − If the queue is full, produce overflow error and exit.
* **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
* **Step 4** − Add data element to the queue location, where the rear is pointing.
* **Step 5** − return success.



Algorithm for enqueue operation

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

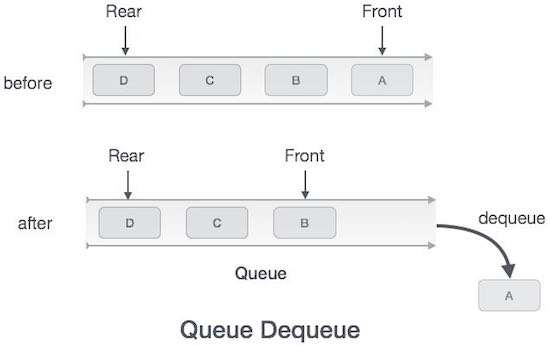
return 1;

end procedure

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

* **Step 1** − Check if the queue is empty.
* **Step 2** − If the queue is empty, produce underflow error and exit.
* **Step 3** − If the queue is not empty, access the data where **front** is pointing.
* **Step 4** − Increment **front** pointer to point to the next available data element.
* **Step 5** − Return success.



Algorithm for dequeue operation

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

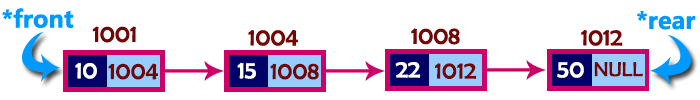
}

**Implementation of Queue using Linked List**

A queue can be easily implemented using linked list. In singly linked list implementation, enqueueing happenes at the tail of the list and dequeueing happens at the head of the list.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

Example

In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

**enQueue(value) - Inserting an element into the Queue**

We can use the following steps to insert a new node into the queue...

Step 1: Create a newNode with given value and set 'newNode → next' to NULL.

Step 2: Check whether queue is Empty (rear == NULL)

Step 3: If it is Empty then, set front = newNode and rear = newNode.

Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

**deQueue() - Deleting an Element from Queue**

We can use the following steps to delete a node from the queue...

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

**display() - Displaying the elements of Queue**

We can use the following steps to display the elements (nodes) of a queue...

Step 1: Check whether queue is Empty (front == NULL).

Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.

Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).

Step 4: Finally! Display 'temp → data ---> NULL'.

**Advantages of Queues**

1. **Multiple Clients**

While queues are more complex than stacks, the array makes queues easy by placing the newest element at the end and moving each element over one step when one piece of data is removed from the queue. Queues are helpful when multiple consumers share a particular process. For example, a website might only have so much space to allow consumers to download a particular file. With a stack, some consumer might have to wait much longer to download the file than the newest consumers. Queues are also useful when the client does not necessarily receive the data at the same time the data is sent.

1. **Circular Queues**

Queues can lead to empty spaces in the data structure, since a bigger array is needed than the total number of pieces of data. However, programmers can use circular queues to use the empty space. Programmers can set up time outs where jobs wait until the entry reaches the data queue.

1. **Speed**

Data queues are a fast method of inter-process communication. Data queues free up jobs from performing some work, which can lead to a better response time and an overall improvement in system performance. Data queues serve as the fastest form of asynchronous communication between two different tasks, since there is less overhead than with database files and data areas.

1. **Flexibility**

Queues are flexible, requiring no communications programming. The programmer does not need any knowledge of inter-process communication. Data queues allow computers to handle multiple tasks. The queue can remain active when there are no entries, ready to process data entries when necessary.

1. **Multiple Jobs**

Some jobs have performance restraints and cannot handle all the entries, so the data entries are spread out across multiple jobs. For example, only one customer service representative can help a customer at a time, so the queue can spread customer service requests among the representatives, for quicker processing.

**Disadvantages of queues**

A major disadvantage of a classical queue is that a new element can only be inserted when *all* of the elements are deleted from the queue.

As an example, consider the queue:

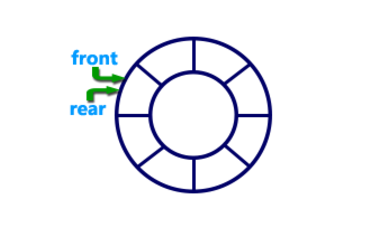
https://qph.fs.quoracdn.net/main-qimg-36f821ba65bbe40a29823c7967eaace9

Now, if the first three members are de-queued from the front (left hand side) of the queue, we get:

https://qph.fs.quoracdn.net/main-qimg-14ce7ef0fb7ab1086daa78bb37497291

Where the queue remains full but we can not insert a new element because, the back of the queue (right hand side) remains as it was before. this is the major limitation of a classical queue, i.e. even if there is space available at the front of the queue we can not use it.

So to overcome the problem above, we can use a *circular queue*. With reference to [Circular Queue - Data Structures](http://btechsmartclass.com/DS/U2_T10.html), this can be defined as 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.” We can represent this circular queue as:



If a queue is considered circular, when a de-queue operation occurs, re-pointing the head of the queue to the next element is a simple assignment. This also avoids extensive re-buffering when all the elements would otherwise *move one to the left.*