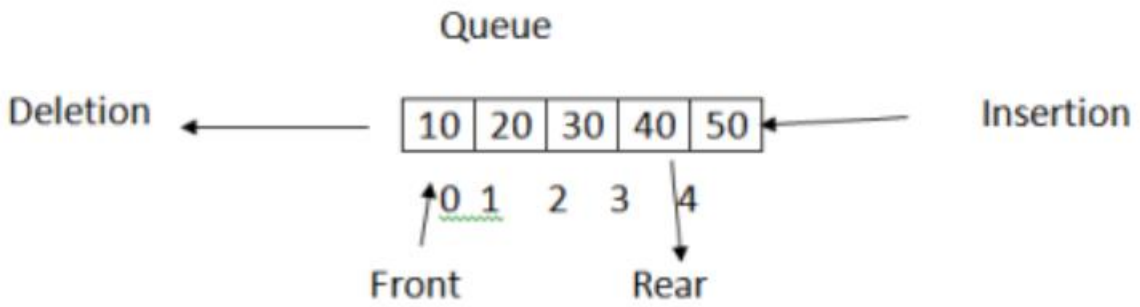
A linear queue

A linear queue is a linear data structure that serves the request first, which has been arrived first. It consists of data elements which are connected in a linear fashion. It has two pointers, i.e., front and rear, where the insertion takes place from the front end, and deletion occurs from the front end. thus, queues are also called First-in First-Out lists (FIFO) or Last-In-Last-Out lists (LILO)



.

Operations on Linear Queue

There are two operations that can be performed on a linear queue:

- **Enqueue:** The enqueue operation inserts the new element from the rear end.
- **Dequeue:** The dequeue operation is used to delete the existing element from the front end of the queue.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **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.

Let's first learn about supportive functions of a queue –

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-1
      return true
   else
      return false
   endif

end procedure
```

isempty()

Algorithm of isempty() function −

Algorithm

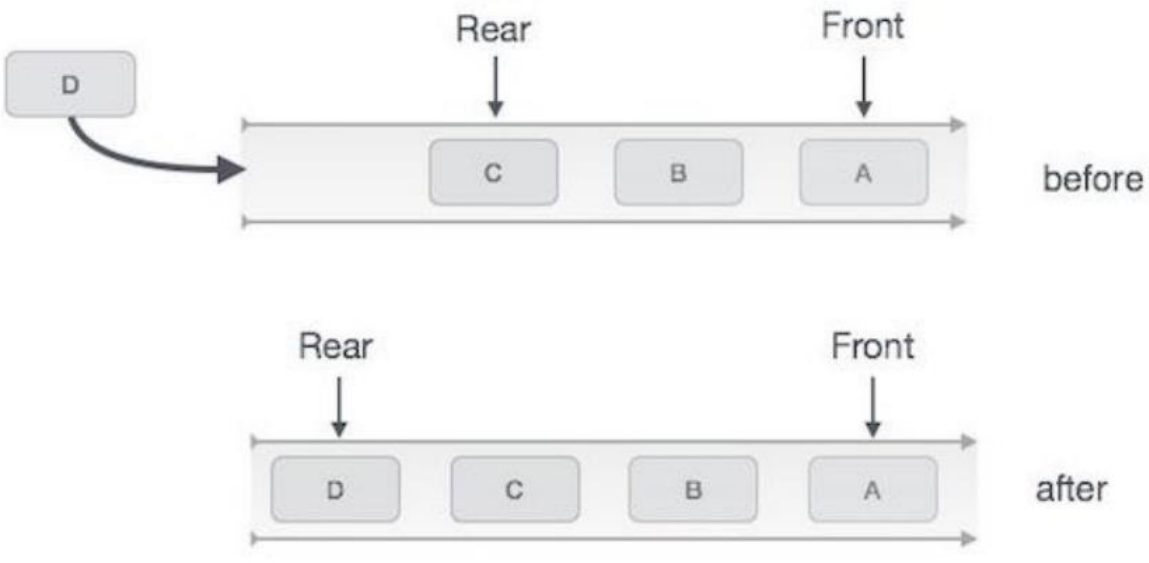
```
begin procedure isempty

   if ((rear==-1 and front==-1)or front==size-1)
or(front==rear)

      return true
   else
      return false
   endif

end procedure
```

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.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

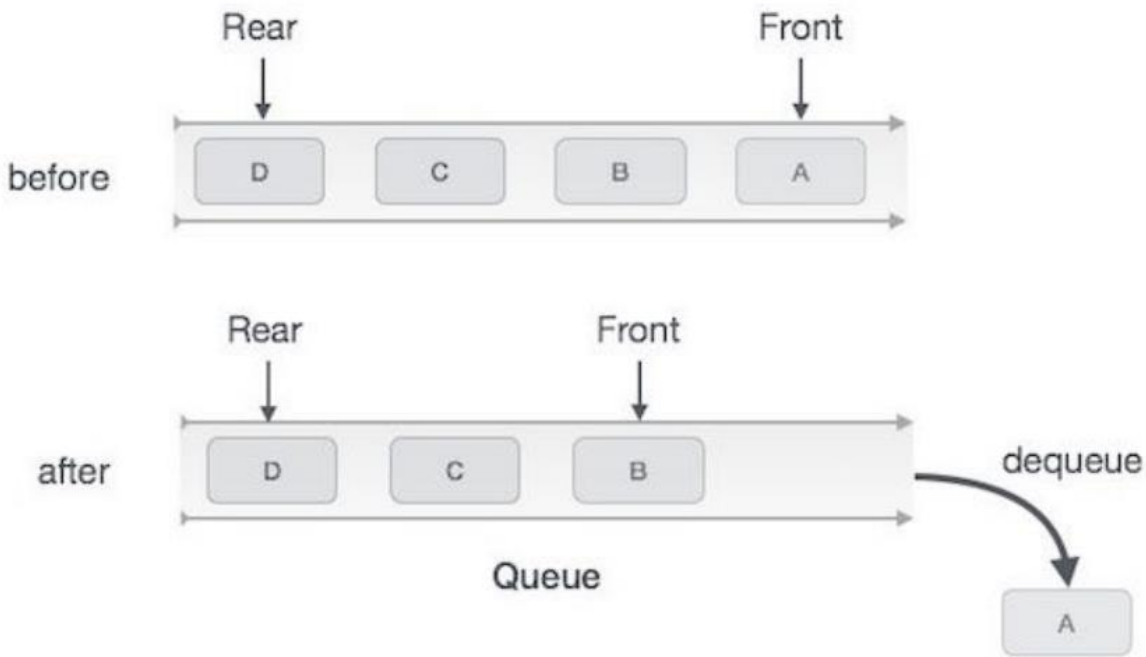
Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

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, Increment **front** pointer to point to the next available data element.
- **Step 4** − access the data where **front** is pointing.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is emptyA
      return underflow
   end if
   front ← front + 1
   data = queue[front]

   return true

end procedure
```

```
#include<iostream>
#include<conio.h>
using namespace std;
const int size=8;
template <class T>
class linearqueue
{
private:
int front;
int rear;
int count;
T arr[size];
public:
linearqueue()
{
front=-1;
rear=-1;
count=0;
}
```

```
bool isEmpty()
{
if((rear==-1 && front==-1)|| front==size-1|| (front==rear))
return true;
else
return false;
}
bool isFull()
{
if((rear==size-1))
return true;
else
return false;
}
void enqueue (T item)
{
if(isFull())
{
cout<<"Queue is Full\n";
}
else
{
rear++;
arr[rear]=item;
count++;
}}
T dequeue ()
{
T dequeueitem;
if(isEmpty())
{
cout<<"Queue is Empty\n";
return 0;
}
```

```
else
{
front++;
dequeueitem=arr[front];
count--;
return dequeueitem;-->return arr[++front]
}
}
int numberofitems()
{
return count;
}

void print_rearandfront()
{cout<<"\nrear="<<rear<<"    front="<<front<<endl;
}

void print_queue()
{
            for (int i = front+1; i <= rear; i++)
cout << "Position : " << i << " , Value  : " << arr[i]<<endl;

}};
void main()
{int s;
linearqueue <int> q1;
q1.dequeue();           //empty
q1.print_rearandfront();  //r=-1  f=-1
q1.enqueue(1);
q1.enqueue(2);
q1.enqueue(3);
q1.enqueue(4);
q1.enqueue(5);
q1.enqueue(6);
```

```
q1.print_queue(); //position (0)=1 2 3 4 5 6

q1.print_rearandfront();//r=5  f=-1
 cout<<"number of items in the queue:
"<<q1.numberofitems()<<endl; //6
 s=q1.numberofitems();
 for(int i=0;i<s;i++)
cout << q1.dequeue() << endl;
q1.print_rearandfront(); //r=5  f=5
cout<<"number of items in the queue:
"<<q1.numberofitems()<<endl;//0
q1.enqueue(77);
q1.print_rearandfront();  //r=6  f=5
q1.enqueue(88);
q1.print_rearandfront();//r=7  f=5
q1.enqueue(99);  //full
q1.print_queue();  // position (6)=88 99
cout<<"number of items in the queue:
"<<q1.numberofitems()<<endl; //2
cout << q1.dequeue() << endl; //77
q1.print_rearandfront();  //r=7  f=6
cout<<"number of items in the queue:
"<<q1.numberofitems()<<endl; //1
cout << q1.dequeue() << endl; //88
q1.print_rearandfront();  //r=7  f=7
cout<<"number of items in the queue:
"<<q1.numberofitems()<<endl; //0
q1.enqueue(99);  //full
q1.print_queue();
cout << q1.dequeue() << endl; //empty

getch();
}
```

```
Queue is Empty

rear=-1    front=-1
Position : 0 , Value  : 1
Position : 1 , Value  : 2
Position : 2 , Value  : 3
Position : 3 , Value  : 4
Position : 4 , Value  : 5
Position : 5 , Value  : 6

rear=5    front=-1
number of items in the queue: 6
1
2
3
4
5
6

rear=5    front=5
number of items in the queue: 0

rear=6    front=5

rear=7    front=5
Queue is Full
Position : 6 , Value  : 77
Position : 7 , Value  : 88
number of items in the queue: 2
77

rear=7    front=6
number of items in the queue: 1
88

rear=7    front=7
number of items in the queue: 0
Queue is Full
Queue is Empty
```