Queue Implementation

Queue

- Queue follows the **First In First Out(FIFO)** rule i.e., the data item stored first will be accessed first.
- Queue is an abstract data structure
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue)

Queue Specifications

- ✓ A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:
- ✓ Enqueue: Add element to end of queue
- ✓ Dequeue: Remove element from front of queue
- ✓ IsEmpty: Check if queue is empty
- ✓ IsFull: Check if queue is full
- ✓ Peek: Get the value of the front of queue without removing it

- **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.

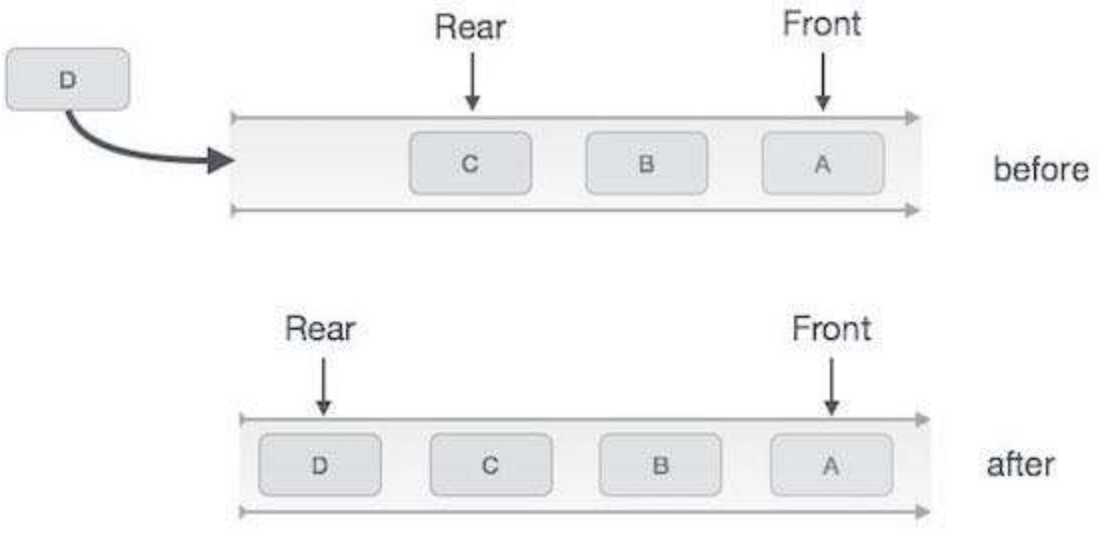
Enqueue Operation

- Queues maintain two data pointers, **front** and **rear**.

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.

Enqueue representation



Queue Enqueue

Algorithm for enqueue operation

```
procedure enqueue(data)
        if queue is full
            return overflow
        endif
        rear ← rear + 1
        queue[rear] ← data
        return true
end procedure
```

Enqueue

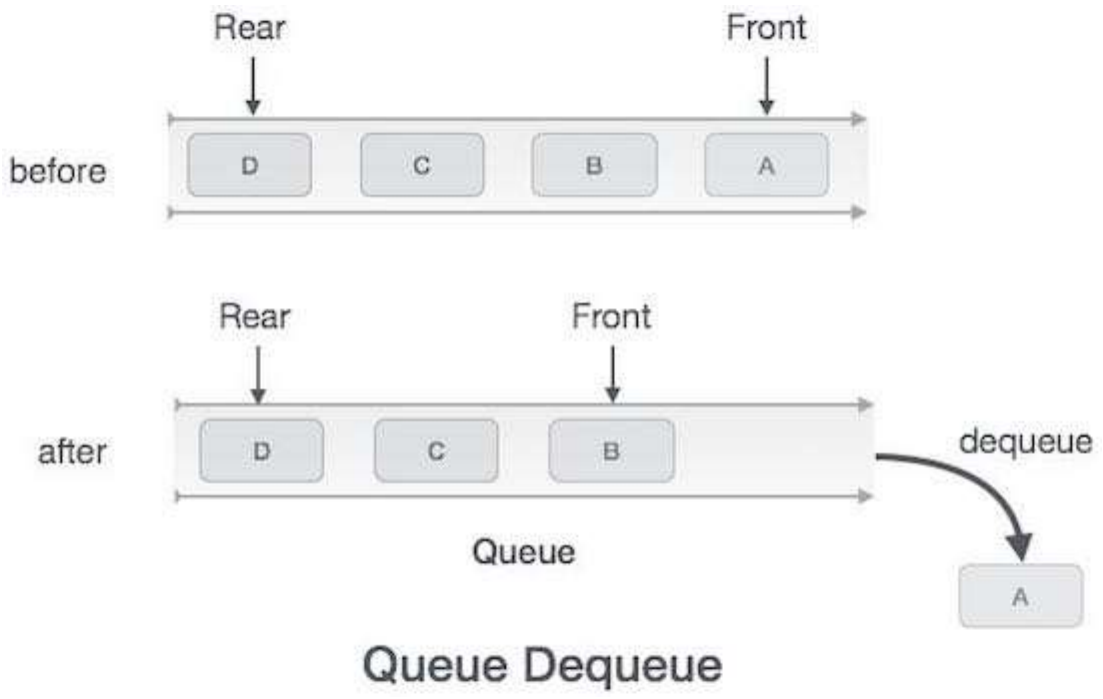
Example

```
int enqueue(int data)
if(isfull())
        return 0;
        rear = rear + 1;
        queue[rear] = data;
return 1;
```

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.
- **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.

Dequeue reprsentation



Algorithm for dequeue operation

Algorithm for dequeue operation

```
procedure dequeue
  if queue is empty
           return underflow
  end if
data = queue[front]
front ← front + 1
     return true
end procedure
```

Dequeue operation

Example

```
int dequeue()
 {
   if(isempty())
         return 0;
int data = queue[front];
front = front + 1;
return data;
}
```

Implementation using C programming

```
#include<stdio.h>
 #define SIZE 5
 void enQueue(int);
void deQueue();
 void display();
int items[SIZE], front = -1, rear = -1;
int main()
{
//enQueue 5 elements
enQueue(1);
 enQueue(2);
 enQueue(3);
 enQueue(4);
enQueue(5);
display();
//deQueue removes element entered first i.e. 1
deQueue();
//Now we have just 4 elements
display();
return 0;
}
void enQueue(int value)
{
 if(rear == SIZE-1)
    printf("\nQueue is Full!!");
else
{
if(front == -1)
front = 0;
rear++;
items[rear] = value;
printf("\nInserted -> %d", value);
 }
 }
```

```
void deQueue(){
    if(front == -1)
        printf("\nQueue is Empty!!");
    else{
        printf("\nDeleted : %d", items[front]);
        front++;
        if(front > rear)
            front = rear = -1;
    }
}
```

```
void display(){
    if(rear == -1)
        printf("\nQueue is Empty!!!");
    else{
        int i;
        printf("\nQueue elements are:\n");
        for(i=front; i<=rear; i++)
            printf("%d\t",items[i]);
    }
}
```