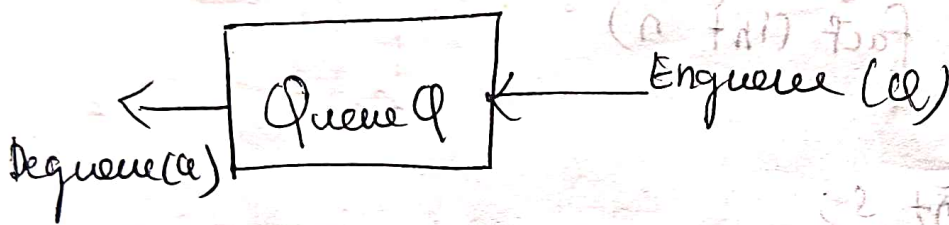


The Queue ADT

Queue is a linear data structure which follows First In First out principle in which insertion is performed at rear end and deletion is performed at front end.

Model of a queue



operations on Queue

Enqueue :-

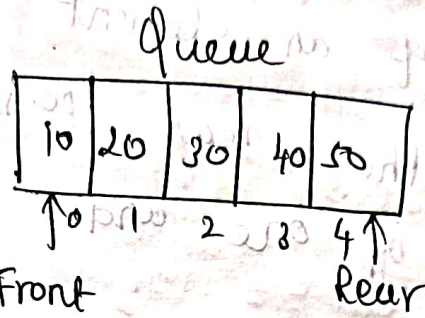
which inserts an element at the end of the list (called the rear).

Dequeue :-

which deletes the element at the start of the list (known as front)

* overflow!:-

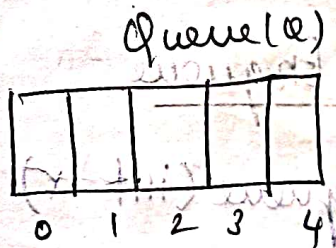
Attempt to insert an element when the queue is full is said to be overflow.



Enqueue (60) [it] causes overflow

* Underflow

Attempt to delete an element when the queue is empty is said to be underflow.



Front = -1

Rear = -1

Dequeue (Q) causes underflow

Queue Implementation

There are two ways for implementing

the queue

* Array Implementation

* Linked list implementation

Array Implementation

There are two pointers used for implementing. one is Rear and other one is Front pointer.

* To insert an element 'x' on to the queue(Q), the rear pointer is incremented by one and said

$Queue[rear] = x;$

* To delete an element $Q[front]$ is returned and the front pointer is incremented

Routine to Enqueue

```
void enqueue(int x)
```

```
{
```

```
    if (rear > max - Arraysize)
```

```
        printf("Queue overflow");
```

```
    else
```

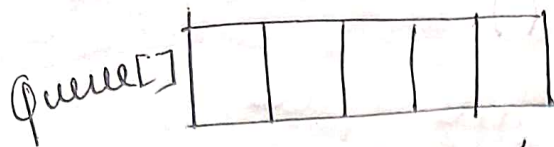
```
    {
```

```
        rear = rear + 1;
```

```
        Queue[rear] = x;
```

```
    }
```

Example :-



Rear = -1 0 1 2 3 4

Front = 0

Enqueue (10)

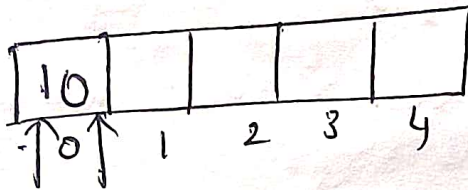
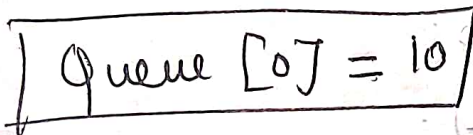
if (rear > max Array size)

-1 > 5

else

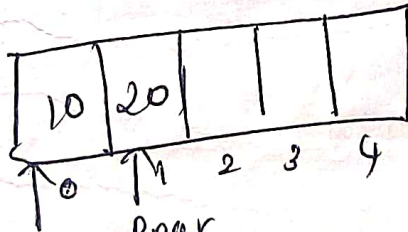
rear = rear + 1

rear = -1 + 1 = 0



Rear Front

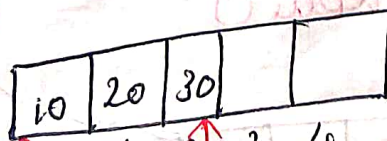
Enqueue (20)



Front

Rear

Enqueue (30)



Front

Rear

Dequeue (Q)

Void delete ()

```
{  
  if (front < 0)  
    printf ("Underflow");
```

```
  else
```

```
  {
```

```
    x = Queue [front];
```

```
    if (front == rear)
```

```
    {
```

```
      front = 0;
```

```
      rear = -1;
```

```
    }
```

```
  } else
```

```
    front = front + 1;
```

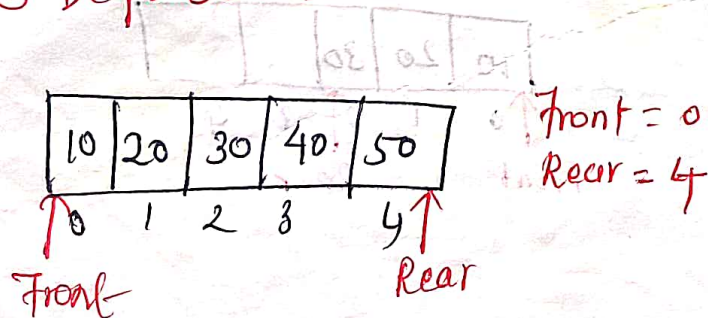
```
  }
```

```
  return x;
```

```
}
```

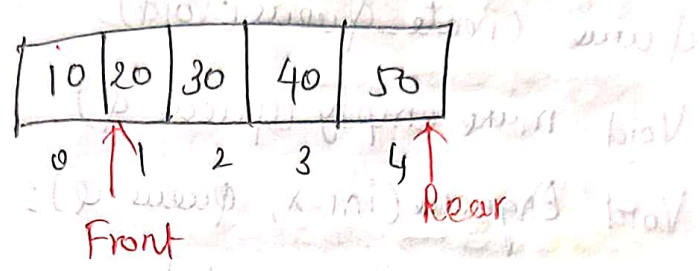
Example

① Dequeue ()



$x = \text{Queue}[\text{Front}]$
 $x = \text{Queue}[0]$
 $x = 10$

$\text{front} = \text{front} + 1$
 $\boxed{\text{front} = 0 + 1 = 1}$

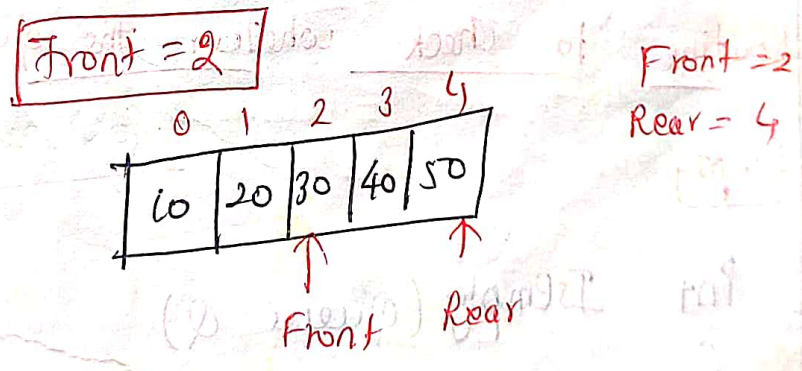


② Dequeue()

$\text{Front} = 1$
 $\text{Rear} = 4$

$x = \text{Queue}[1]$
 $x = 20$

$\text{Front} = \text{Front} + 1$
 $= 1 + 1$



Linked List Implementation of Queue

Enqueue operation is performed at end of the list. Dequeue operation is performed at Front of the list.

Routine to Enqueue an Element

```
Void Enqueue(int x)
{
    struct node *newnode;
    newnode = malloc(sizeof(struct node));
    if(rear == NULL)
    {
        newnode -> data = x;
        newnode -> next = NULL;
        front = newnode;
        rear = newnode;
    }
    else
    {
        newnode -> data = x;
        newnode -> next = NULL;
        rear -> next = newnode;
        rear = newnode;
    }
}
```

Example

Enqueue x=10



rear = NULL

front = NULL

if (rear == NULL) (True)

newnode → data > 10

10	
----	--

newnode → next = NULL
front = newnode

10	/
----	---

rear = newnode

10	/
----	---

10	/
----	---

Enqueue X = 20

newnode =

--	--

 = front

- if (rear == NULL) (False)
else

newnode → data = 20 ⇒

20	
----	--

newnode → next = NULL ⇒

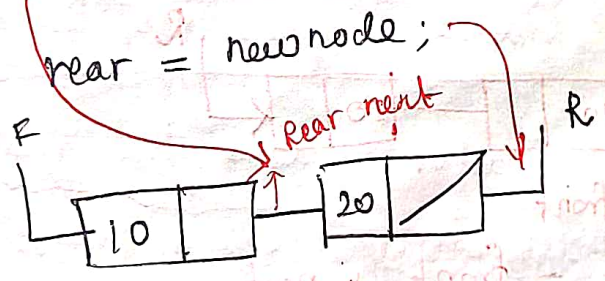
20	/
----	---

 R

rear → next = newnode;

20	/
----	---

 R



Routine to Dequeue

Void dequeue (Queue Q)

{ struct node *temp;

if (front == NULL);

Error ("Underflow");

else

{ temp = front;


```
if (front == rear)
```

```
{
```

```
    front = NULL;
```

```
    rear = NULL;
```

```
}
```

```
else
```

```
{
```

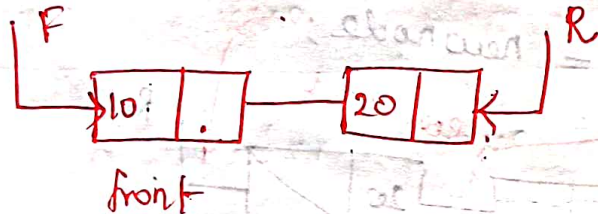
```
    front = front → next;
```

```
    printf("temp → data");
```

```
    free(temp);
```

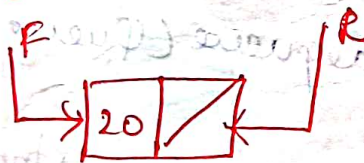
```
}
```

Example



```
front = front → next;
```

```
front = 20
```



Displays the deleted data & deallocates
the memory allocated for the
variable temp