

A ■ P ■ I ■ I ■ T

ASIA PACIFIC INSTITUTE OF
INFORMATION TECHNOLOGY

QUEUES

QUEUES

Q- is an ordered collection of items

Items are deleted from the **head** (**Front**)

Items are added at the **tail** (rear)

A Queue is some times called a **FIFO** list as opposed to a stack.

Five parameters of a Q

(Perspective of an Operational Researchers)

Arrival pattern

Service Mechanism

No of servers

Q discipline

Q capacity

Three Primitive Operations on QUEUE

Insert(Q name, element)

Which insert **item x** at the **rear** of the **queue Q**

x = **remove(Q-name)** - underflow for an empty Q

Which deletes the front elements from the queue q and set x to its content

empty(Q-name)



Which returns false or true depending on whether or not the queue contains any elements

Example : | | | A | B | C | | |

front ↗
↘ rear

| | | B | C | | |

front ↘ rear ↙

| | | B | C | D | E | |

Sequence of operations ?

Assume that the queue is initially empty.

1. Insert(q,A)
2. insert(q,B)
- 3.insert(q,C)
4. x = remove(q,E)
5. insert(q,E)

The **insert** operation can always be performed, since there is no limit to the no. of elements a queue may contains.

The **remove** operations can be applied only if the queue is nonempty.

The result of an illegal attempt to remove an element from an empty queue is called **UNDERFLOW**

Implementing a Queue in C

A Queue can be implemented in C by means of an **array** and two integer variable viz. **Front** and **Rear**

```
#define MAXQUEUE 100  
  
:  
:  
  
struct queue  
{  
    int items[MAXQUEUE];  
    int front,rear;  
}
```

Meaning of Operations on a Queue

Initial settings for an empty Q

* $q.rear = -1$

* $q.front = 0$

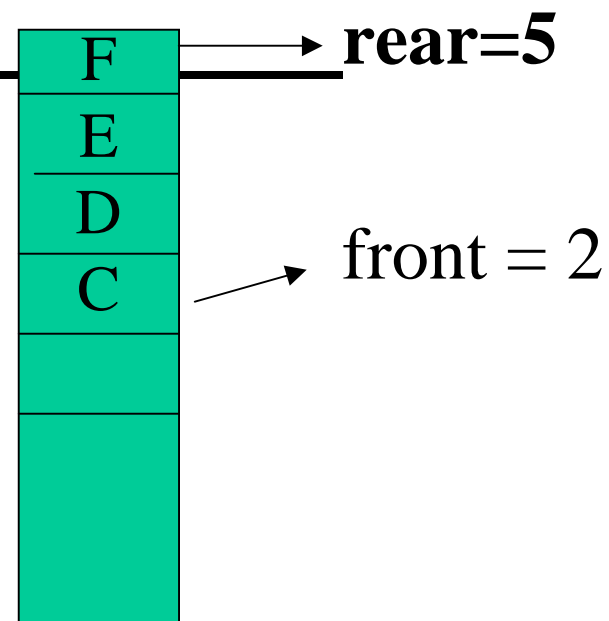
$insert(q, x) \rightarrow q.items[++q.rear] = x;$

$x = remove(q) \rightarrow x = q.items[q.front++]$

$empty(q) \rightarrow q.rear < q.front$

Note : No of elements in $q = q.rear - q.front + 1$

Disadvantage of this approach



After a few operations $q.rear$ becomes $MAXQUEUE - 1$ but array locations may remain unutilized below $q.front$

ie We will not be able to add. Sometimes even an empty array appears full.

How to overcome this ?

One solution

Shift all elements in an array after each **deletion** operation to the beginning of the array

`x = remove(q)` can modified as

```
x=q.items[0];
```

```
for(i=0 ; i<q.rear; i++)
```

```
    q.items[i] = q.items[ i+1];
```

```
    q.rear--;
```

Note : In this case, `q.front` becomes irrelevant

`q.rear == -1` indicate that `q` is empty

Obviously, this method is computational explosive

How to overcome the computational problem ?

[Using Circular List Representation]

Another is to view the array that holds the queue as a circle rather than as a straight line.

Ie the first element of the array as immediately following its last element

means if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty.

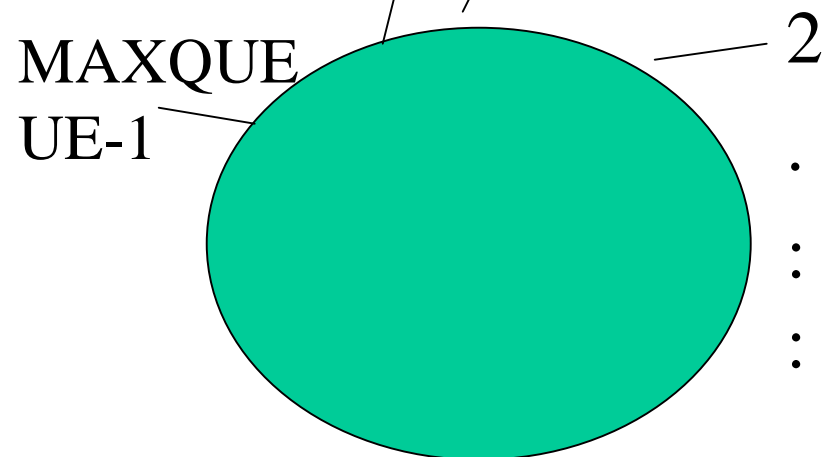
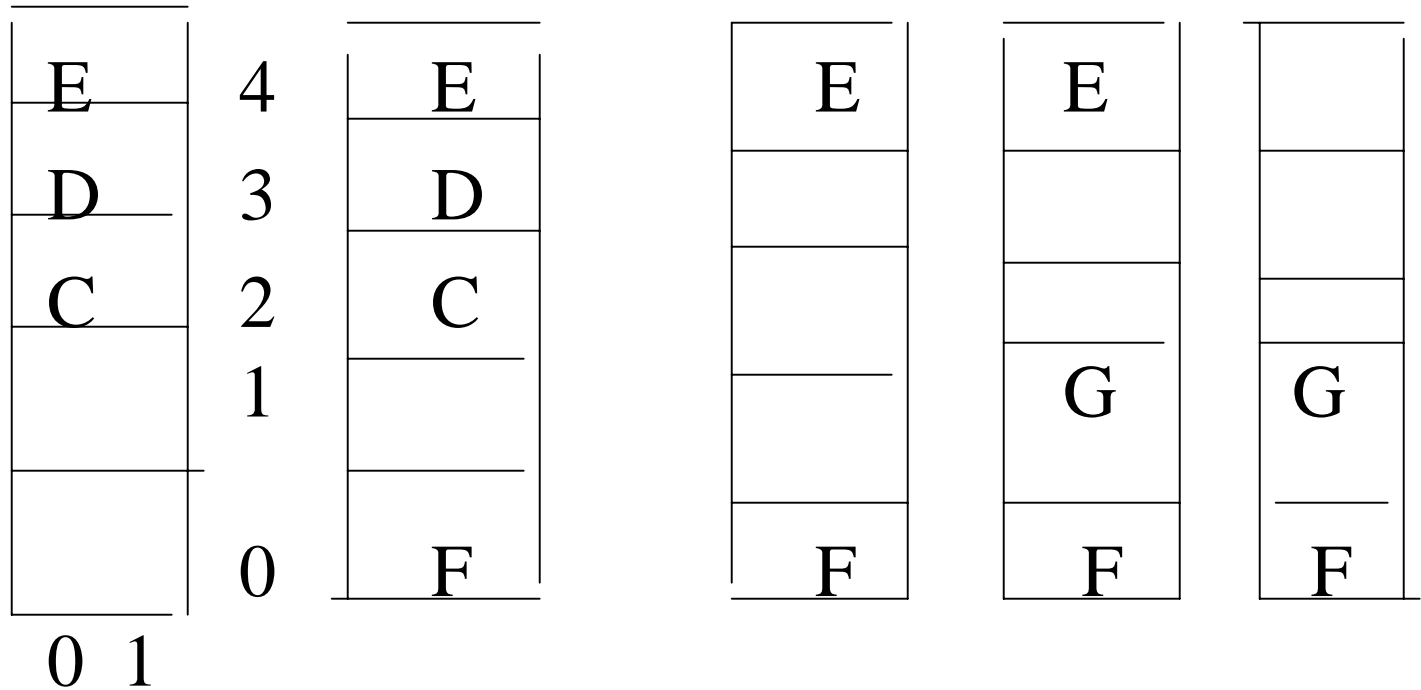
Problem in the Queue Management

Under this representation it is difficult to determine when the queue is empty.

The condition $q.rear < q.front$ is no longer valid as a test to the empty queue



How to avoid the computation ?



Note : The condition
 $q.rear < q.front$ cannot be
 used to check for empty Q.

A queue of integers may therefore be declared and
initialized by

```
#define MAXQUEUE 100;

struct queue
{
    int items[MAXQUEUE];
    int front,rear;
};
```

```
struct queue q;
```

```
a.front = q.rear = MAXQUEUE-1;
```

Note : q.front and q.rear are initialized to the last index of the
array, rather than to -1 or 0, because the last element of
the array immediately precedes the first one within the
Queue

A test for the empty queue is implemented by the statement

```
#define MAXQUEUE 100;

struct queue
{ int items[MAXQUEUE];
  int front,rear;
};
```

empty(pq)

```
struct queue *pq;
{
  return((pq -> front == pq -> rear) ? TRUE : FALSE);
}
```

Usage :-

```
if(empty(pq))  
    /* Q is empty */  
else  
    /*Q is not empty
```

The function remove(q) can be coded as

```
remove(pq)  
struct queue *pq;  
{ if (empty(pq))  
    { printf(“\n Q underflow”;  
      exit(1);  
    }  
}
```

```
If(pq -> front == MAXQUEUE-1)
```

```
    pq -> front = 0;
```

```
else
```

```
    (pq -> front) ++;
```

```
    return (pq->items[pq -> front]);
```

```
}
```

Note :

pq->front must be **updated** before an element is extracted.

Insert Operation

How to differentiate between empty and full Queues ?

4	E	E	E
3	D	D	D
2	C	C	C
1			G
0		F	F

$$Q.FRONT = Q.REAR = 1$$

This indicate that q.front equals q.rear which is precisely the indication of Underflow.

It seems that there is no way to distinguish between the **empty queue** and **the full queue** under this implementation.

How to overcome this ?

__ One solution is to **sacrifice one element of the array** and to allow a queue to grow only as large as one less than the size of the array.

Use only **MAXQUEUE-1** elements of the array

Now the insert function can be coded as

```

insert( pq, x)
struct queue *pq;
int x;
{
    /* make room for new element */
    if(pq -> rear == MAXQUEUE-1)
        pq -> rear =0;
    else
        (pq -> rear)++;
    /* Check for overflow */
    if(pq -> rear == pq.front)
        {printf(“\n Queue Overflow”); exit(1); }
    pq -> items[pq -> rear] = x;  return;
}

```

Note

- * The test for overflow in `insert(q,x)` occurs after incrementing `pq -> rear`
- * The test for underflow occurs in `remove(q)` before updating `pq -> front`

PRIORITY QUEUE

Stack / Queue

- * Elements are ordered based on the sequence in which they have been inserted
- * pop operation retrieves the last element inserted and the remove operation remove the first element inserted

The Priority Queue is a data structure in which the intrinsic(real/ actual) ordering of the element does determine the results of its basic operations.

PRIORITY QUEUE

There are two types of Priority Queue

1) An Ascending Priority Queue

is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed .

For an example, if `apq` is an ascending priority queue, the operation `pqinsert(apq,x)` inserts element `x` into `apq` and `pqmindelete(apq)` removes the minimum element from `apq` and returns its value

2) A Descending Priority Queue

PRIORITY QUEUE

There are two types of Priority Queue

2) A Descending Priority Queue

is similar but allows deletion of only the largest item.

THE END OF IVth LESSON