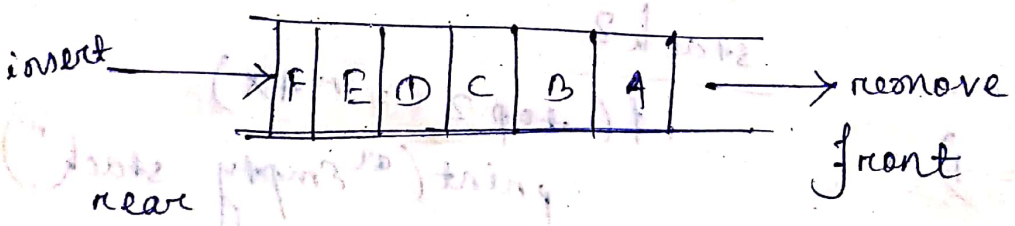
Q. Write a C program for housing 2 stacks into one array.

20/3/15

Queue

- It is a special type of linear list in which insertion is restricted to occur at one end of the list, called the rear and deletion is restricted to occur at the other end of the list, called the front.



- The insertion op" is called "insert"
- If the deletion " " " "remove".
- The operating principle of queue is "First in First out" etc. (FIFO), i.e. the elements are removed in the order in which they were inserted into the queue.

(* stack & queue are dynamic data structure)

Let the arrival times

A → t_1

B → t_2

C → t_3

D → t_4

E → t_5

F → t_6

Let t_7 → total time

waiting time

A = $t_7 - t_1$

B = $t_7 - t_2$

C = $t_7 - t_3$

⋮

F = $t_7 - t_6$

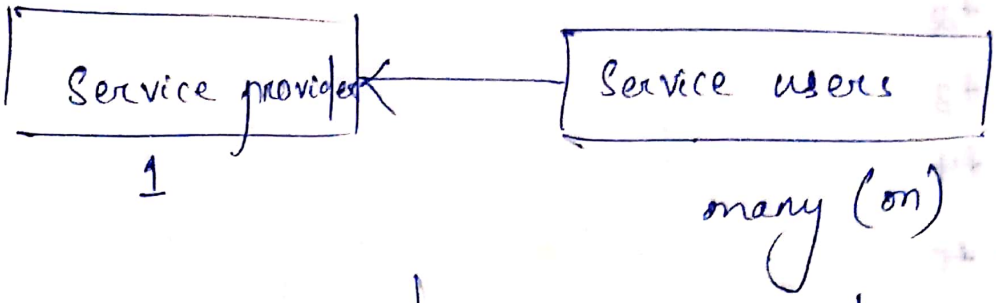
~~∴ The~~

- The element at the rear of the queue has been in the queue for least amount of time & the element at the front of the queue has been in the queue for longest period of time.

Why queue is required?

- Whenever queue is concerned there are 2 things : - Service provider
  - " user

- Queue is always required when the service provider is one but the service users are many.



- That means the service can be provided to all the users but not simultaneously.
- So this case queue is required.

(* All the comp. resources can be shared. But one resource can't be used by multiple users simultaneously. That is the sharing of resources are mutually exclusive in time.)

* When multiple prog.s request a CPU time, the OS forms a queue.
* So case of comp. networks, the task is done a queuing theory, called 1 to on queuing theory.

- Queue is a very popular data structure basically used in OS, to make use of CPU, to store files in the disk, sharing printers, comp. networks (resources are shareable but not simultaneously).

Basic operations

1. CREATE(Q) – returns an empty queue Q.
2. ISEMPTY(Q) – " a boolean type (true/false)
3. INSERT(E,Q) – inserts element E at rear of the queue Q.
4. REMOVE(Q) – Removes the front element of the queue Q.

* When empty Q is created,
  front = null
  rear = null

Insert(A,Q) → front = A, rear = A
Insert(B,Q) → front = A, rear = B
Insert(C,Q) → front = A, rear = C
⋮
Insert(F,Q) → front = A, rear = F

front → fixed
rear → changes

Remove(Q) → front = B, rear = F
Remove(Q) → front = C, rear = F
⋮

rear → fixed
front → changes

FRONT (CREATE(Q)) = null

REAR (CREATE(Q)) = null

NOEL (CREATE(Q)) = 0 NOEL → no. of elements

ISEMPTY (CREATE(Q)) = true

ISEMPTY (INSERT(E,Q)) = false

REAR (INSERT(E,Q)) = E

if (ISEMPTY(Q))
then FRONT (INSERT(E,Q)) → E

if (ISEMPTY(Q))
then REMOVE (Q) → error

REMOVE (CREATE(Q)) → error

~~REMOVE (INSERT(E,Q)) = E~~

if (ISEMPTY(Q))
then REMOVE(INSERT(E,Q)) → E

REMOVE (INSERT(E,Q)) → FRONT

Implementation of Q

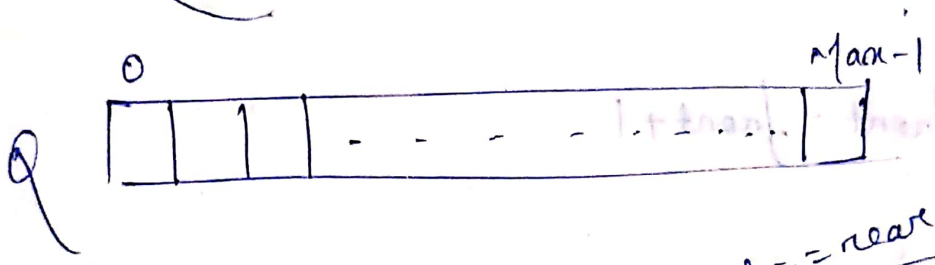
- Queue can be housed in an array subjected to:

1. Elements of the queue must be homogenous

② Upper bound on the no. of elements in the queue must be specified.

③ FIFO rule must be enforced by the programmer to maintain the integrity of the queue.

④ Underflow & overflow cond's must be checked.




```
#define MAX 25
char Q[MAX];
int front, rear;
```

front == rear
in 2 cases:
- Q is empty
- single element in Q

① CREATE(Q)
```
front = -1
rear = -1
```

② ISEMPTY(Q)
```
if (front == -1)
    return (true);
else
    return (false);
```

③ INSERT(E,Q)
```
if (rear == MAX-1)
    print("Queue overflow")
else
{
    if (front == -1)
        front = front + 1
    rear = rear + 1
    Q[rear] = E
}
```

④ REMOVE(Q)

```
if ( front == -1)
    print ("Queue underflow")
else
{
   E = Q[front]
   if ( front == rear)
      front = rear = -1
   else
      front = front + 1
}
```

⑤ Display(Q)

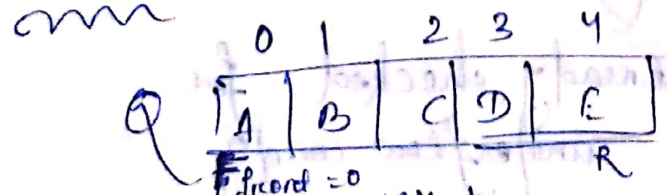
```
if (front == -1)
   print ("Empty queue");
else
{ for (i = front ; i <= rear ; i++)
    print ( Q[i]);
}
```

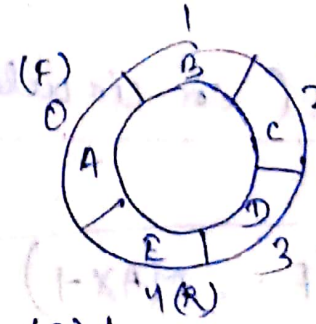
Q. Write a C program for array implementation of a linear queue.

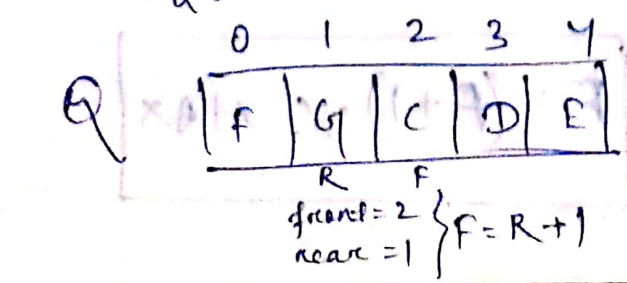
* When elements are removed from the queue the queue is free at the front end. But since the rear = MAX-1, it will show the menage "Q overflow" if insertion is not ponible. To avoid this situation the rear is considered as the front. This type of queue is called circular queue. In this case whenever the user wants to insert, then the next loc^n of the last element is free if the element is inserted.

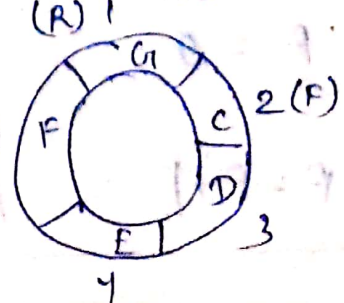
23/3/15

Circular queue









Full Q (in circular Q)

if((F == 0 && R == MAX-1) || (F == R+1))

↳ logical conclusion
front is just ahead of rear

combining the 2 cond's

if (F == (R+1) % MAX) → true cond? for a full Q

Insert (E, Q) (in circular Q)

~~if (R != MAX-1)~~ already checked that Q is not full.

```
if (R == MAX-1)
    R = 0
else
    R = R+1
```

↳ instead we can write

R = (R+1) % MAX

Remove (in circular Q)

already checked for underflow cond?

```
if (F == MAX-1)
    F = 0
else
    F = F+1
```

→ F = (F+1) % MAX

Op^{n}s in circular Q

① Create (Q)

```
F = -1
R = -1
```

② isempty (Q)

```
if (F == -1)
    return (true)
else
    return (false)
```

③ Insert (E, Q)

```
if (F == (R+1) % MAX)
    print ("Queue overflow")
else
{
    R = (R+1) % MAX
    if (F == -1)
        F = 0
    R = (R+1) % MAX
    Q[R] = E
}
```

④ Remove (Q)

```
if (F == -1)
    print ("Queue underflow")
else
{
    E = Q[F]
    if (F == R)
        F = R = -1
    else
        F = (F+1) % MAX
}
```

(5) Display (Q)

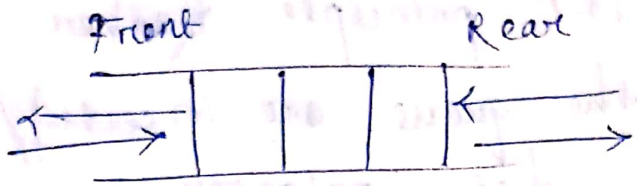
```
if (R >= F)
    for (i = F ; i <= R; i++)
        print (Q[i])
else
{
    for (i = F; i <= MAX-1 ; i++)
        print (Q[i])
    for (i = 0; i <= R ; i++)
        print (Q[i])
}
```

or

```
if ( F == -1)
    print ("Empty Q")
else
{
    for (i = F ; i != R ; i = (i+1) % MAX)
        print (Q[i])
    print (Q[i])
}
```

Q. Write a C program for array implementation of circular queue.

D Queue (Double ended Queue)



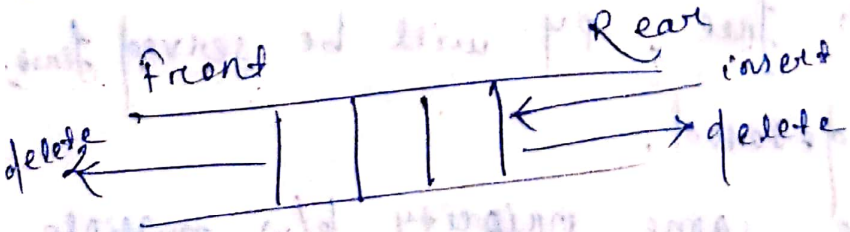
Both the ends are opened for both insertion & deletion.

- With some restrictions D-Queue has 2 variations:

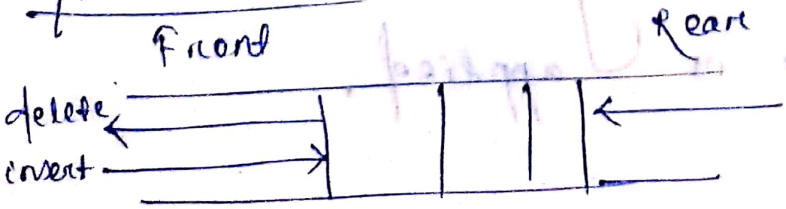
① i/p restricted — DQ → i/p can be done only on rear side

② o/p " — DQ → o/p can be done only on front side

i/p restricted



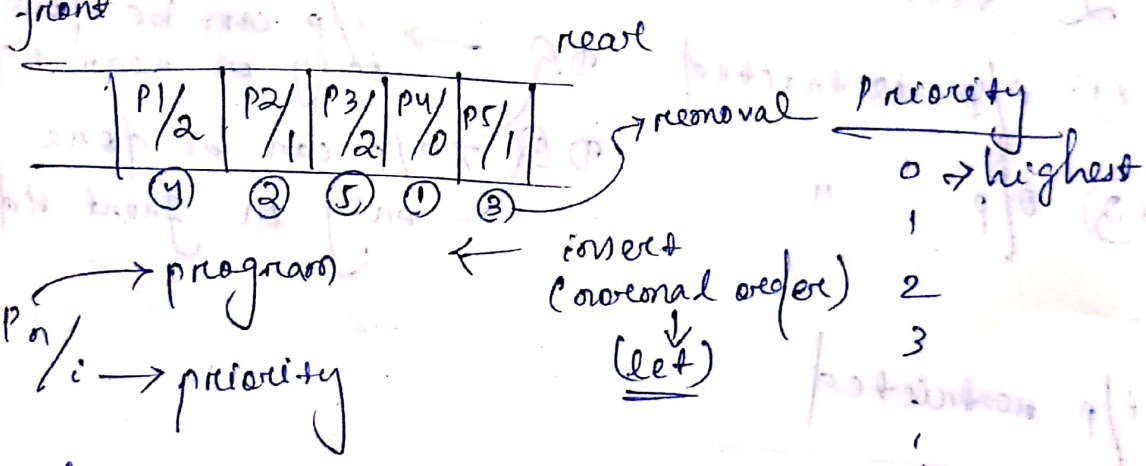
o/p restricted



Q. Write a C prog. for array implemented DQ.

Priority Queue

- It can't follow FIFO principle. Rather the elements of the queue are inserted/deleted according to their priority.
- An element can be inserted in front side
- " " " " deleted from any pos



when CPU is free, P4 will be served first. i.e. P4 is deleted.

If there are same priority b/w multiple elements, then among those elements FIFO principle is applied.

```
struct item
{   int value;
    int priority;
}
```

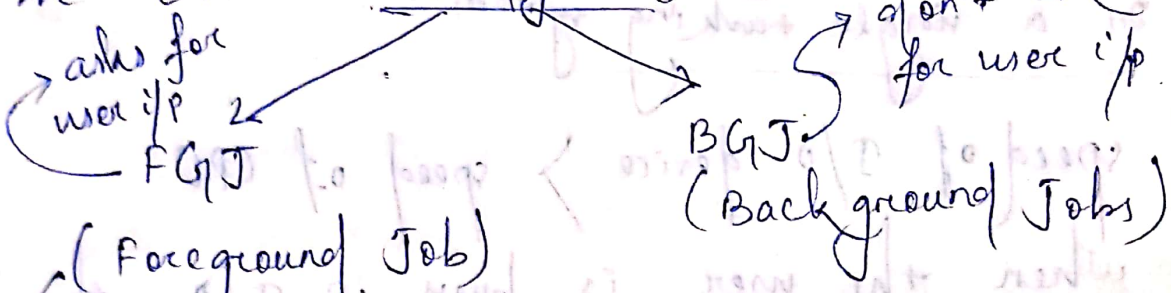
struct item PQ[MAX];
{ Enter the item:
{ Enter " priority:

whenever an element is removed then all the elements on the right to the deleted are shifted to left.

Appl^n's of PQ

The prog.s ready to run / waiting for CPU are entered into a queue → Ready Q.

In Unix Ready Q



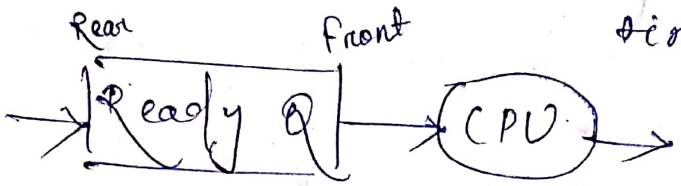
whenever CPU is free, it's allocated to FGJ.

Q. Write a C prog. for implementing a Priority Q.

Appl^{n}s of queue

① CPU time-sharing

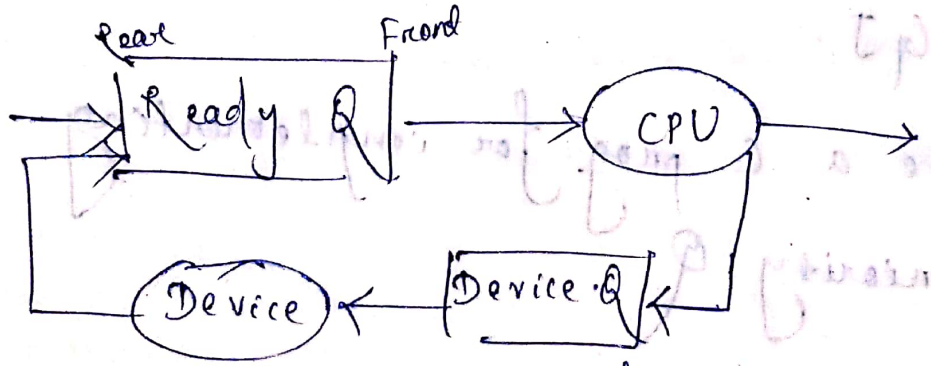
(one CPU, many prog.s
CPU time can be shared, but CPU time is mutually exclusive in time)



In a multitasking system
Whenever a prgm. requests CPU time & CPU is n't free, OS puts the prog. into the ready Q.

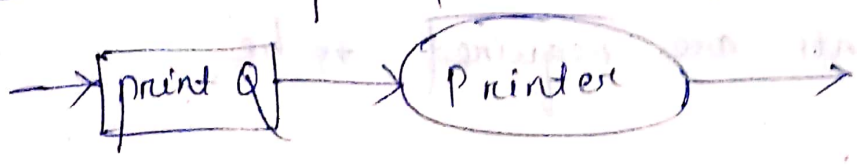
In a single tasking system

speed of I/O device > speed of CPU
when the user is busy in I/O, the CPU sits idle. It's a wastage of CPU time.



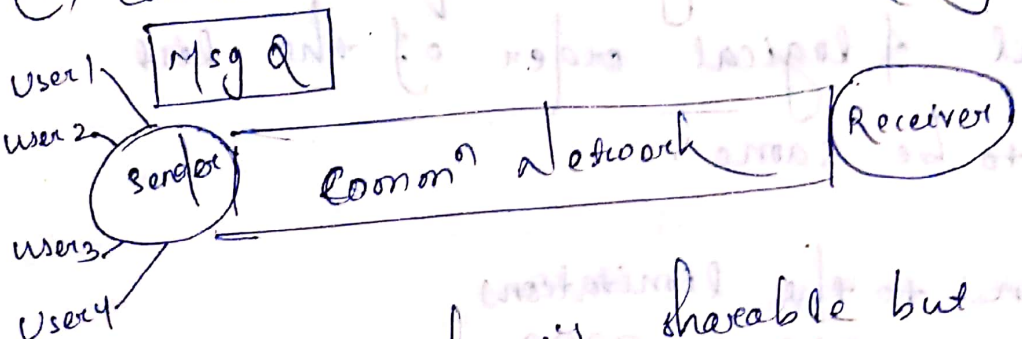
After serving by CPU if any prog. requires device service then it's put into a device Q.

(2) Files spooled onto a printer



Eg
If any file is requesting printer service & printer isn't free then it's inserted into print Q.

(3) Communication network (Menage Q)



Comon network is shareable but one at a time. Otherwise collision occure & all the packets will be lost.
When multiple users are accesing a single comon network, then their menages are put into a Q called menage Q.

25/3/15

Limitations of array implementation

(1) Amount of space allocated is fixed.
(2) Space is allocated regardless of whether the list is full or empty.

```
curr = curr -> next ;
}
}
```

Linked list representation of queue

```
struct qnode
{
    int info;
    struct qnode *next;
};
struct qnode *front, *rear, *curr;
```

(1) create

```
front = NULL
rear  = NULL
```

(2) isempty

```
if (front == NULL)
    return (true) ;
else
    return (false) ;
```

(3) insert

```
curr = (struct qnode *)malloc(sizeof(struct qnode));
curr->next = NULL;
```

```
scanf("%d", curr->info); //or curr->info = E
                                (if in callingpgm
                                 insert(f, &Q))

if (front == NULL)
    front = rear = curr;
else
{
    rear->next = curr;
    rear = curr;
}
```

(4) Remove

```
if (front == NULL)
    printf("\n Q underflow");
else
{
    if (front == rear)
    E = front->info;
    curr = front;
    if (front == rear)
        front = rear = NULL
    else
        front = front->next;
    curr->next = NULL;
    free(curr);
}
```

(5) Display

```
if (front == NULL)
    printf("\n Empty Q")
else
{
    curr = front;
    while (curr != NULL)
    {
        printf (curr -> info);
        curr = curr -> next;
    }
}
```

Appl's of linked list

(1) Management of free space

OS allocates memory using malloc() & frees space using free().
Free space is managed using linked list.
OS maintains the "Avail" pointer. It points to the 1st free space, the next part of the 1st free space points to the next free space & so on.