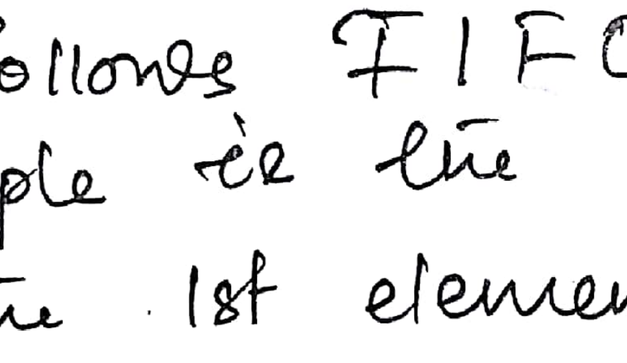
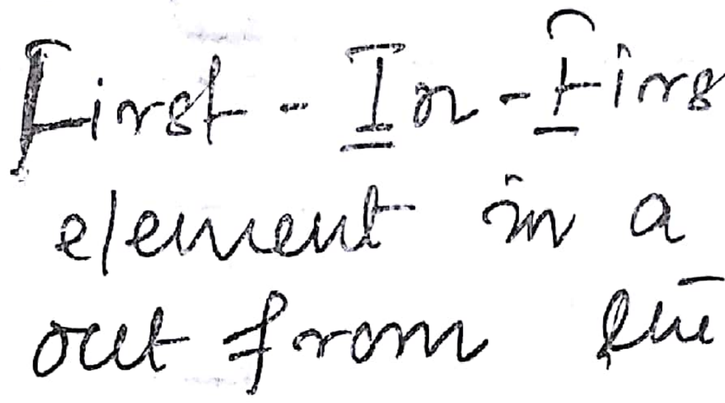
Queue

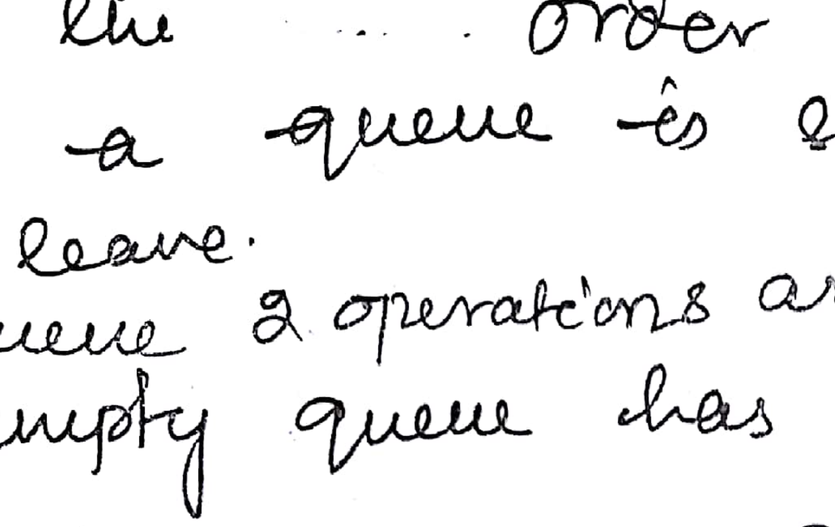
Def^n:

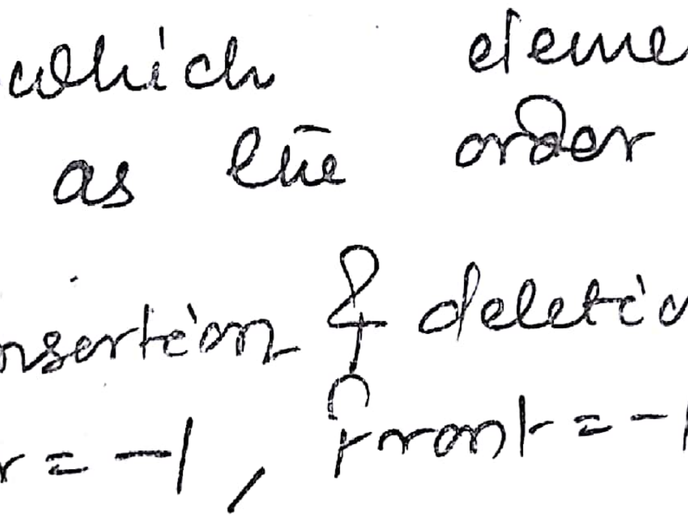
> A queue is a non primitive, linear data structure.
> It is a homogeneous collection of elements in which an element is added at one end called rear end. & the existing element is deleted from the other end called front end.
> It follows FIFO (First-In-First-Out) principle i.e the 1st element in a queue will be the 1st element out from the queue.
> Thus the order in which elements enter into a queue is same as the order in which they leave.
> On queue 2 operations are insertion & deletion.
> An empty queue has Rear = -1, Front = -1

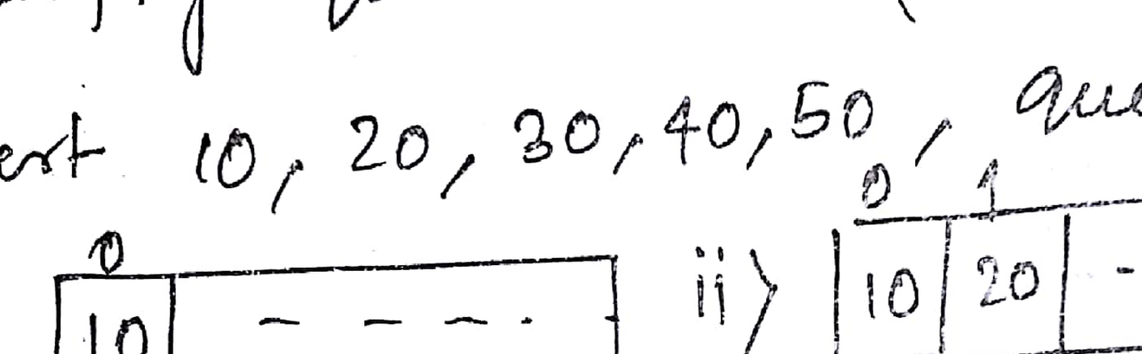
Q) insert 10, 20, 30, 40, 50, queue size = 50

i)


ii)


iii)


iv)


v)


> When ever an element is inserted in the queue the value of rear is incremented by 1.
i.e rear = rear+1
> When ever 1st element is inserted in the queue, at that time both front & rear is incremented by 1.

After that the front will not be incremented or changed during insertion operation.
) When ever an element is removed or deleted from the queue, the value of front is incremented by 1.
) When ever only one element is present in the queue (ie front == rear condition) & we want to do deletion operation then front & rear value is set to -1. ie

Queue implementation:

i) static implementation using array.
ii) Dynamic implementation by using pointer (ie linked list rep^n)

(i) Array Rep^n:

) Here the beginning of the array will become the front of the queue and last loc^n of the array will act as rear of the queue.
) while using array rep^n the total no. of elements in a queue is given by

No of elements present in a queue = rear - front + 1

operation on queue:

i) insertion:

'C' code →

```
#define SIZE 10
void qins(int ele)
{ if (rear == SIZE - 1)
  {
    Pf("overflow");
    qstat();
    exit(0);
```

```
else
    { if(front == -1)
  {rear = 0;
     front=0;
      }
    else
       {
         rear = rear+1;
       }
       q[rear] = ele;
    }

}
```

Algorithm:

qins(q[SIZE], ele)

1) check for queue is already fill or not
   if rear = SIZE-1
2) Write overflow
3) else
   if front = -1
4) set rear=0,
5) set front=0
6) else
   set rear = rear+1
7) set q[rear] = ele
   [end of ~~else~~ if structure]
8) Exit

11) Deletion in a linear queue:-

'C' code:

```
void qdel( )
{
 int ele;
 if( front == -1)
  {
   Pf("queue underflow");
   getch();
   exit(0);
  }
else { ele = q[front];
       if(front == rear)
        {
         front = -1;
         rear = -1;
        }
      else
        front = front+1;
      }
Pf("element deleted is %d", ele);
}
```

Algorithm:

qdel[q[SIZE])

1) Set ele

2) [check for underflow]
   if front=-1

3) write queue underflow

4) else
   set ele = q[front]

5) if front = rear

6) set front=-1

7) set rear=-1

8) else
   set front = front+1

9) write element deleted
   [end of if structure]

10) Exit.

Find the front & rear posn for following operation when size of queue is 5.

i) insert 10 → front=0, rear=0

ii) insert 20 → front=0, rear=1

iii) insert 30 → front=0, rear=2

iv) insert 40 → front=0, Rear=3

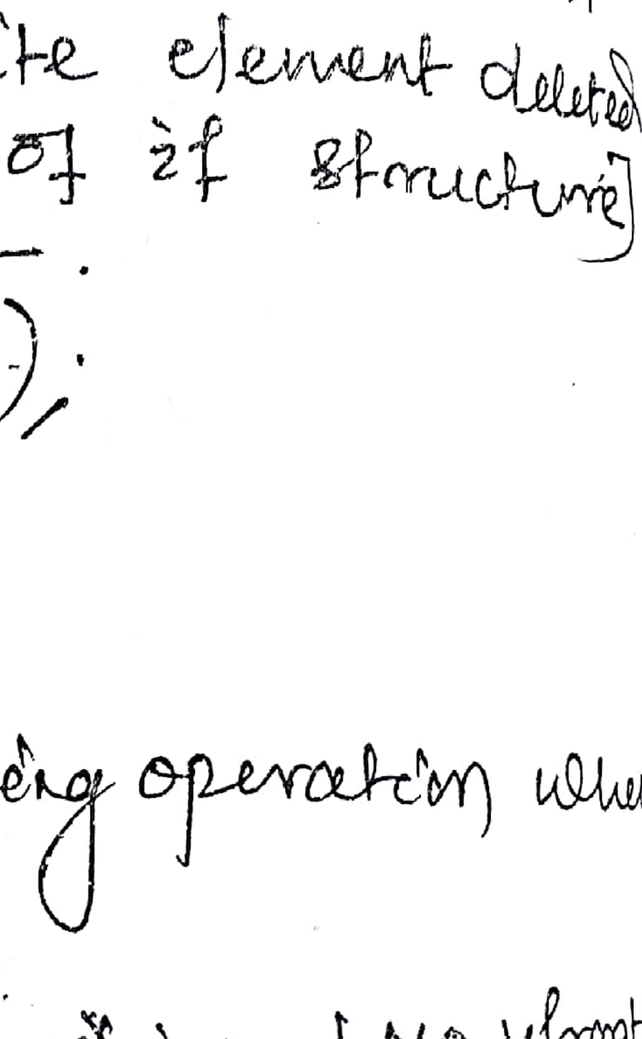
v) insert 50 → front=0, Rear=4

vi) delete → front=1, Rear=4

vii) delete → front=2, Rear=4

viii) delete → front=3, Rear=4

ix) insert 60 → (overflow)



Menu driven program for queue:

```
#define SIZE 10
void qins( int ele);
void display();
void qdel();
int q[SIZE], front=-1, rear=-1;
void main()
{
int c, ele, choice;
clrscr();
while(1)
{
Pf("\n 1. INSERT \n 2. DELETE \n 3. DISPLAY \n
        4. EXIT \n");
Pf("Enter choice");
Sf("%d", &choice);
switch(choice)
{
case 1: printf("Enter the element want to insert");
        scanf("%d", &ele);
        qins(ele);
        break;
case 2:     qdel();
            break;
case 3: display();
        break;
case 4: exit(0);
        break;
default:
        Pf("Invalid choice");
}
}
getch();
}
```

```
void qins(int ele)
{
 if(rear==SIZE-1)
  {pf("overflow");
   getch();
   exit(0);
  }
else
  {
    if(front==-1)
    {
     front=0;
     rear=0;
    }
  else
     rear=rear+1;
  }
  q[rear]=ele;
 }
}
void qdel()
{ int ele;
 if(front==-1)
 {
  pf("underflow");
   getch();
   exit(0);
 }
else { ele=q[front]
 if(front==rear)
  { front=-1;
    rear=-1;
  }
else
  front=front+1;
 pf("element deleted is %d",ele);
}
void display()
{
 int i;
 if(front==-1)
 {pf("No element");
  getch();
  exit(0);
 }
else
 {
 for(i=front;i<=rear;i++)
  pf("%d", q[i]);
 }
}
```

Pasvii

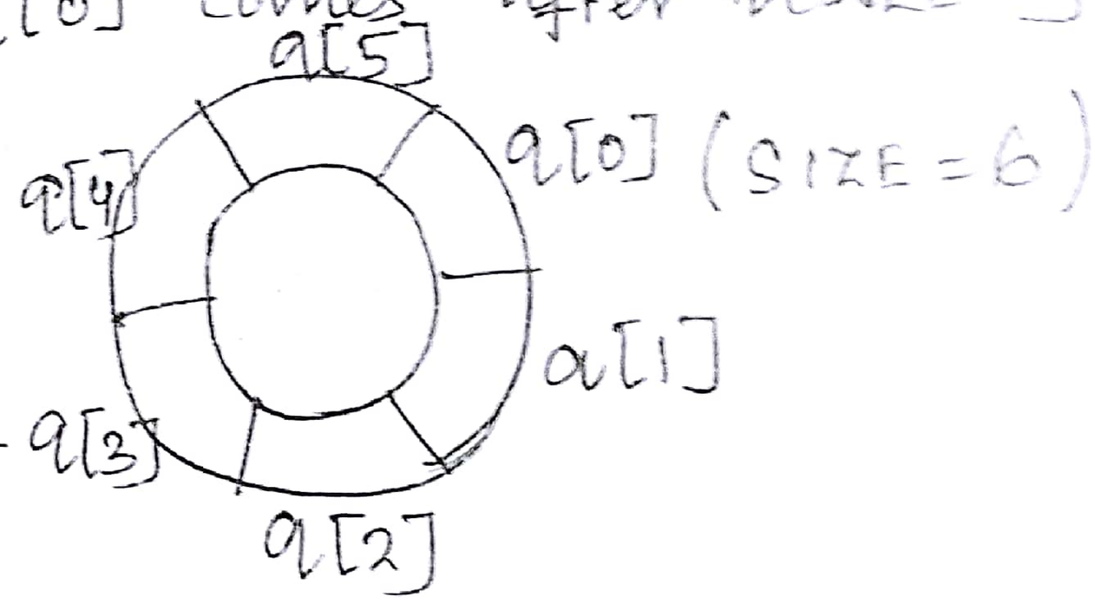
CIRCULAR QUEUE

Disadvantage of linear queue :-

1) Incase of linear queue, if the last pos^n of queue is occupied, then it is not possible to insert any more elements even through some position are vacant towards front position of the queue.

Def^n of circular queue :-

> circular queue is one in which 1st element comes just after the last element ie a[0] comes after a[SIZE-1]



> In case of circular queue queue initialization & testing for overflow condn are same

> Rear is calculated by
(a) the formula rear = (rear+1) % SIZE

> Front is calculated by the formula front = (front+1) % SIZE

> condition for overflow case ie
front == (rear+1) % SIZE.

Operation on a circular queue :-
1) insertion
2) deletion

Q) Write front & Rear pos^n for following operation where queue size is 5.

i) insert 10 → Front=0, Rear=0
ii) insert 20 → Front=0, Rear=1
iii) insert 30 → front=0, Rear=2
iv) insert 40 → front=0, Rear=3
v) insert 50 → front=0, Rear=4
vi) delete → front=1, Rear=4
vii) delete → front=2, Rear=4
viii) delete → front=3, Rear=4
ix) insert 60 → front=3, Rear=0
x) insert 70 → front=3, Rear=1
xi) insert 80 → front=3, Rear=2
xii) insert 90 → overflow

as front == (rear+1) % SIZE
rear
↳ (2+1) % 5
= 3 % 5
= 3

else front is 3 so overflow

(c) code & Algorithm for insertion in circular queue

(c) code

```
void ins(int ele)
{
 if(front==(rear+1)%SIZE)
   {
    Pf("circular queue overflow");
    getch();
    exit(0);
   }
 else
   {if(front==-1)
     {front=0;
      rear=0;
     }
   else
     rear=(rear+1)%SIZE;
   q[rear]=ele;
   }
}
```

Algorithm

ins(q[SIZE], ele)

1) [check for overflow]
   if (front=(rear+1)%SIZE
2) write circular queue overflow
3) else
   if front=-1
4) set front=0
5) set rear=0
6) else
   set rear=(rear+1)%SIZE
   [end of if structure]
7) set q[rear]=ele
   [end of if structure]
8) Exit.

(c) code & Algorithm for deletion in circular queue:

```
void del()
{ int ele;
 if(front==-1)
   {
    Pf("queue underflow");
    getch();
    exit(0);
   }
 else
   { ele=q[front];
    if(front==rear)
     {front=-1;
      rear=-1;
     }
    else
     front=(front+1)%SIZE;
    Pf("%d element is deleted",ele);
   }
}
```

del(q[SIZE])

1) Let ele
2) [check for underflow]
   if front=-1
3) write queue underflow
4) else
   set ele=q[front]
5) if front=rear
6) set front=-1
7) set rear=-1
8) else
   set front=(front+1)%SIZE
   [end of if structure]
9) write element deleted
   [end of if structure]

Circular queue display ⇒

```
void cqdisplay()
{
int i;
if(front==-1)
{
pf("There is no element");
getch();
exit(0);
}
else if(front<=rear)
{
if(front<=
printf("Circular queue is");
for(i=front; i<=rear; i++)
pf("%d", q[i]);
}
else
{ pf("circular queue is");
for(i=front; i<SIZE; i++)
pf("%d", q[i]);
for(i=0; i<=rear; i++)
pf("%d", q[i]);
}
}
```

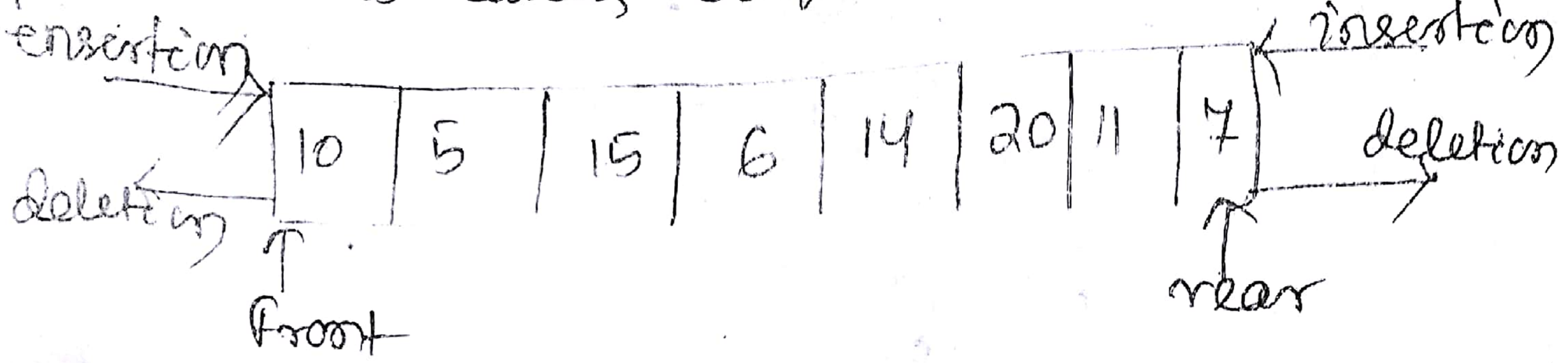
Demerits of Circular queue :-

1) only an element can be inserted at rear end & deleted from front end which is not flexible.
for that reason deque is evolved.

Queue

Double Ended Queue (Deque)

Dqt is a homogeneous list of elements in which insertion & deletion operation are performed from both ends ie we can insert and delete elements from the front end and rear end. so it is called deque (ie double ended queue)



> There are 2 types of deque

i) Inpet - restricted deque

ii) Outpet - restricted deque.

Since both insertion & deletion are performed from either end, it is necesary to design an algorithm to perform the following 4 operation.

1) Insertion of an element from front end of qu

2) " " " rear "

3) Deletion " " front "

4) Deletion " " rear "

i) Input Restricted deque :-

> Dqt allows insertion only at one end ie the rear end but allows deletion at both ends.

> so in this case 2, 3, 4 of above operations are valid.

ii) Output Restricted deque :-

> Dqt allows deletion only at one end ie the front end but allows insertion at both ends.

> so in this case 1, 2, 3 of above operations are valid.

1) Insertion of an element from front end of queue

'C' code

```
void dq-ins-front(int ele)
{
if((rear==front-1) ||
(front==0 && rear==SIZE-1))
{
printf("overflow");
getch();
exit(0)
}
else
{ if(front==-1 && rear==-1)
{
front=0;
rear=0;
}
else if (front==-1)
front=0; rear=0;
else if (front==0)
front=SIZE-1;
else
front=front-1;
q[front]=ele;
}
}
```

Algorithm

dq-ins-front(ele, q[SIZE])

1) [check for overflow ?]
if((rear == front-1) ||
(front=0 && rear=SIZE-1)

2) Write overflow

3) else
if front=-1 && rear=-1

4) set front=0

5) set rear=0

6) else if front=0

7) set front=SIZE-1

8) else
set front=front-1
end of if structure

9) set q[front] = ele
end of if structure

10) EXIT.

2) Insertion of an element from rear end of queue.

'C' code

```
void dq-ins-rear(int ele)
{
if((front==rear+1) ||
(front==0 && rear==SIZE-1))
{
printf("overflow");
getch();
exit(0);
}
else if(front==-1 && rear==-1)
front=0; rear=0;
else if (rear==SIZE-1)
rear=0;
```

Algorithm:

dq-ins-rear(ele, q[SIZE])

step:
1) [check for overflow]
if(front = rear+1) || (front=0 &&
rear = SIZE-1)

2) Write overflow

3) else
if rear = SIZE-1

4) set rear=0

```
        else
            rear=rear+1;
        q[rear]=ele;
    }
}
```

5) else
   Set rear=rear+1
6) Set q[rear]=ele
   [end of if structure]
7) [end of if structure]
8) Exit

3) Deletion of an element from front end of queue

```
void dq_del_front()
{
    int ele;
    if(front==-1)
    {
        printf("Underflow");
        getch();
        exit(0);
    }
    else
    {
        ele=q[front];
        if(front==rear) front=-1; rear=-1;
        else if(front==SIZE-1)
            front=0;
        else
            front=front+1;
        printf("Deleted element is %d", ele);
    }
}
```

dq_del_front()
Step 1: [Check for underflow]
   if front==-1
Step 2: write underflow
Step 3: else
   set ele=q[front]
Step 4: if front=SIZE-1
   set front=0
Step 5: else
   set front=front+1
   [end of if structure]
Step 6: write element deleted
   is ele.
   [end of if structure]
Step 7: Exit.

4) Deletion of an element from rear end of queue:

'c' code:

```
void dq_del_rear()
{
    int ele;
    if(front==-1)
    {
        printf("underflow");
        getch();
        exit(0);
    }
```

dq_del_rear()
Step 1: [check of underflow]
   if front==-1
Step 2: write underflow
Step 3: else
   Set ele=q[rear]

```
else
{
  ele = q[rear];
  if( rear == 0)
    rear = SIZE-1;
  else
    rear = rear - 1;
  printf("\n element deleted is %d", ele);
}
}
```

```
if((front==rear)&&front!=-1)
{front=-1; rear=-1;}
```

4) if rear = 0
   set rear = SIZE-1

5) else
   set rear = rear - 1
   [end of if structure]

6) write element deleted is ele

7) [end of if structure]
   Exit.

PRIORITY QUEUE :-

> A priority queue is a kind of queue in which each element is assigned with a priority and the order in which elements are deleted and processed comes from the following rules.

1) An element with heighest priority is processed first before any element of lowest priority
2) Two or more elements with the same priority are processed according to the order in which they were added to the queue.

> There canbe different criterias for determining the priority. Some of them are :

1) A shortest job is given heigher priority over a longer job.
2) An important job is given the heigher priority over a routine type job.

Ex: CPU scheduling
    Priority scheduling.