Algorithm →

deletion_from_front()

1. if (front == NULL) then
   ~~printf("Underflow");~~
   write "Underflow" & exit.
2. p = front;
3. item = front → info;
4. if (front == rear) then
   front = rear = NULL;
   else
   front = front → next
   front → prev = NULL
5. free (p)
6. Exit

C function →

```
void delete_from_front()
{
 struct node *p;
 if (front == NULL)
 { printf("Underflow");
   exit(0);
 }
 p = front;
 if (front == rear)
    front = rear = NULL;
 else
    { front = front → next
      front → prev = NULL;
    }
 free(p); }
```

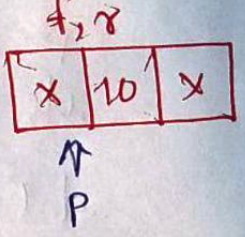
(4) Deletion from rear →

Concept →

Case 1:- Underflow

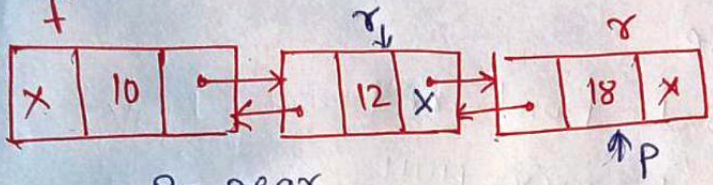
```
if (front == NULL)
  printf("Underflow");
```

Case 2: Only one node




```
P = rear
if (front == rear)
  front = rear = NULL
  free(P);
```

Case 3: more than one node




```
P = rear
rear = rear → prev
rear → next = NULL
  free(P)
```

Algorithm →

Deletion_from_rear( )

1. if (front == NULL) then
   write "~~Underflow~~" "Underflow" & exit
2. P = rear
3. if (front == rear)
   front = rear = NULL
   else
   rear = LINKP[rear]
   LINKN[rear] = NULL
4. free(P)

(iii) C function →

```
void deletion_from_rear()
{
struct node *p;
if (front == NULL)
{ printf("Underflow");
  exit(0);
}
p = rear;
if (front == rear)
  front = rear = NULL;
else
  {
    rear = rear -> prev;
    rear -> next = NULL;
  }
free(p);
}
```

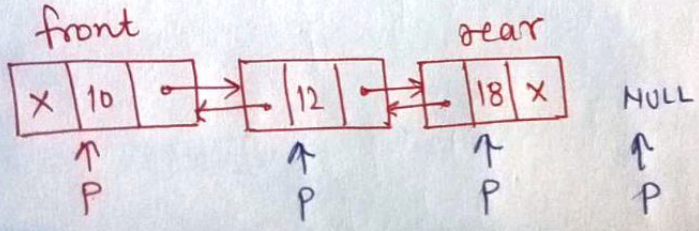
5) Traverse → concept

case 1: Queue is empty

```
if (front == NULL)
  printf("Queue is empty");
```

case 2: Queue is not empty

```
for (p = front; p != NULL; p = p -> next)
    printf("%d", p -> info);
```



Algorithm traverse()

1. If (front == NULL) then
   write "Queue is empty" & exit.
2. P = front
3. repeat step 4 & 5 while (P != NULL)
4. write INFO[P]
5. P = LINK[P]
   [end of loop]
6. Exit.

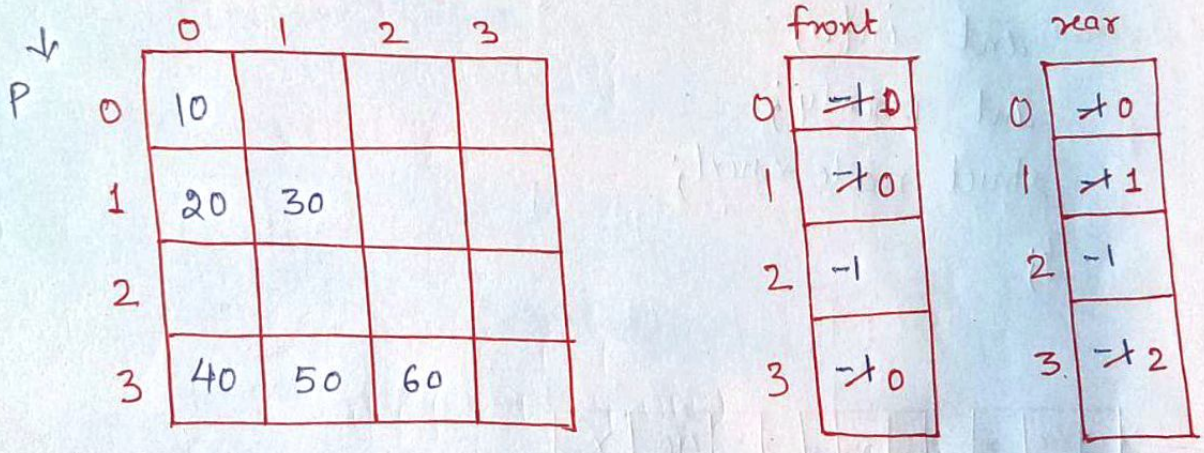
C function →

```
Void traverse()
{
struct node *P;
 if (front == NULL)
 printf("Queue is empty");
 else
 {
   for (P = front; P != NULL; P = P->next)
      printf("%d", P->info);
 }
}
```

Priority Queue → A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1) An element of higher priority is processed before any elements of lower priority.
2) Two elements with the same priority are processed according to the order in which they are ~~deleted~~ added to the queue.

Array representation → (multiple queue representation)



Insert Algo → This algorithm adds an ITEM with priority number P to a priority queue maintained by 2D array Queue.

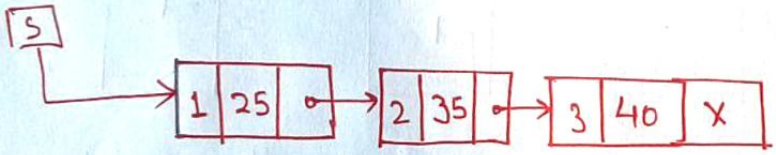
1. Insert ITEM as the rear element in row P of Queue.
2. Exit

DEL Procss Algorithm- This algorithm deletes and processed the first element in a priority Queue maintained by a 2D array Queue.

1) Find the smallest K such that front[K] ≠ -1
   [Find the first non-empty Queue]
2) Delete and procss the front element in row K of Queue.
3. Exit.

Linked List→

```
struct node
{
  int info;
  int priority;
  struct node * next;
};
```



Insert Algo→ This algorithm adds an ITEM(node) with priority Number P to a priority Queue which is maintained in memory as a one way List.

1. Traverse the one way List untill finding a node X whose priority number exceeds P.
2. Insert item in front of node X.
3. if NO such node is found, Insert ITEM(node) as the last node in a List.

Delprocess Algorithm → This algorithm deletes and process the first element in a priority Queue, which appear in a memory as a one way list.

1. Set ITEM = start → info
2. Delete first node from the list
3. Process ITEM
4. Exit.

C program (Array)

```
#include <stdio.h>
#include <stdlib.h>
#define maxsize 30
int f[4]={-1,-1,-1,-1}, r[4]={-1,-1,-1,-1}, pq[4][maxsize], info, p;

void insert()
{
   printf("Enter info");
   scanf("%d", &info);
   printf("Enter priority");
   scanf("%d", &p);
   if(r[p]==maxsize-1)
    {
      printf("Overflow");
       exit(0);
    }
   if(r[p]==-1)
     f[p]=r[p]=0;
  else
     r[p]=r[p]+1;
  pq[p][r[p]]=info;
}
```

```
void deletion()
{
   int item, i=0;
   if(f[i]!=-1)
    {
      item = pq[i][f[i]];
      if(i<r[i])
         f[i] = f[i]+1;        else f[i]=-1;
      }
     else
      i++;
}

void display()
{   int i;
   for(i=0;i<5;i++)
    {
       for(j=f[i];j<=r[i];j++)
          {
             if(f[i]!=-1)
                printf("%d", pq[i][j]);

          }
    }
}
int main()
 {
   int ch, e;
   do
    {
      printf("1. Insert\n");
      printf("2. Deletion\n");
      printf("3. traverse\n");
```

```
printf("\nEnter your choice : ");
scanf("%d", &ch);
switch(ch)
{
    case 1: insert();
            break;
    case 2: deletion();
            break;
    case 3: display();
            break;
}
printf("press -1 to continue: ");
scanf("%d", &e);
} while(e==-1);
return 0;
}
```

Linked List →

```
struct node
{
  int info;
  int P;
struct node *next;
} *front=NULL, *rear=NULL;
```

```
void insert()
{
   struct node *next, *ptr, *q;
   new=(struct node *new) malloc(sizeof(struct node));
    if (new==NULL)
       printf("overflow");
    else
      {
          printf("Enter node info");
          scanf("%od", &new->info);
          printf("Enter node priority");
          scanf("%od", &new->p);
           new->next=NULL;
          if(rear==NULL)
            front=rear=new;
          else
          {
            for(ptr=front; ptr!=NULL&&ptr->p<=new->p;ptr=ptr->next)
                q=ptr;
            if(ptr==front)
             {
               new->next=ptr;
                front=new;
             }
           else if(ptr==NULL)
                q->next=new;
           else
              {
                q->next=new;
               new->next=ptr;
              }
      }
   }
}
```

```
void deletion()
{
    struct node *ptr;
    if(front == NULL)
    {
      printf("Underflow");
       exit(0);
    }
    ptr = front;
    if( front->next == NULL)
        front = NULL;
    else
        front = front->next;
}

void display()
{
    struct node *ptr;
   for(ptr = front; ptr != NULL; ptr = ptr->next)
      printf("%d %d", ptr->p, ptr->info);
}

int main()
{
    int ch, e;
    do
    {
      printf("1. Insert \n");
      printf("2. Deletion\n");
      printf("3. Display\n");
      printf("Enter your choice");
       scanf("%d", &ch);
```

```
switch(ch)
{
    case 1: insert();
            break;
    case 2: deletion();
            break;
    case 3: display();
            break;
}
printf("Press -1 to continue");
scanf("%d", &e);
} while(e==-1);

return(0);
}
```

Recursion → If a function calls itself then this process is known as recursion and the function itself is called recursive function. There must be atleast one condition that ~~satisfies~~ is responsible to exit from recursion. This condition is called base condition.

Principles of recursion →

1. **Base case** – Base case is the terminating condition for recursive function.
2. Each time a function calls itself it must be closer to base case.

3. The initial parameter input value pushed onto the stack.

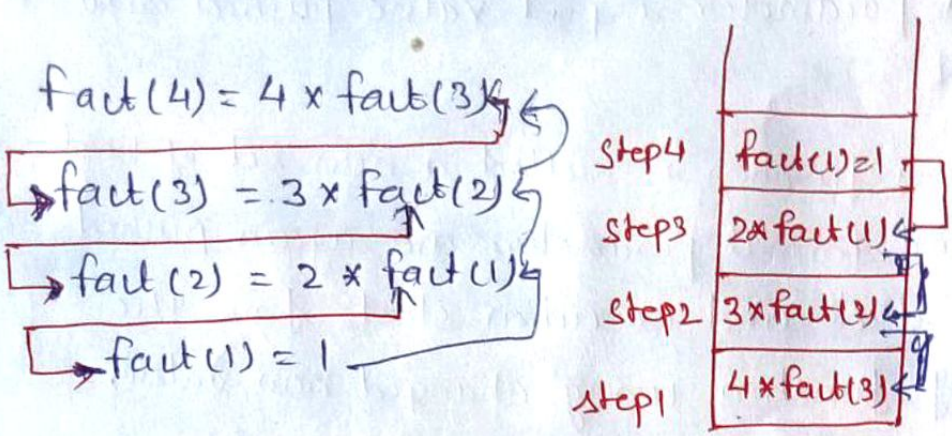
4. Each time a function is called a new set of local variable and formal parameters are again pushed onto the stack and execution starts from the begining of function using changed new value. This process is repeated till a base condition is reached.

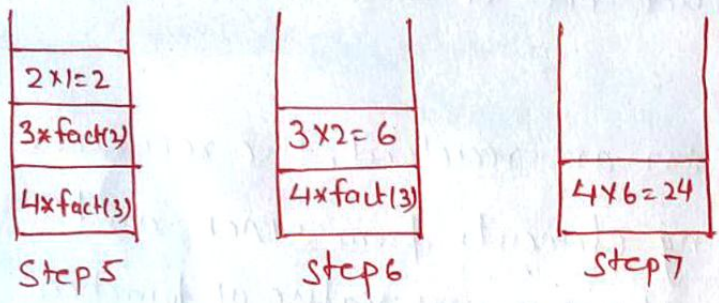
5. Once a base condition are reached, the recursive function calls popping elements from STACK and returns a result to the previous value of function.

6. A sequence of returns ensures that the solution to the original problem is obtained.

Example →

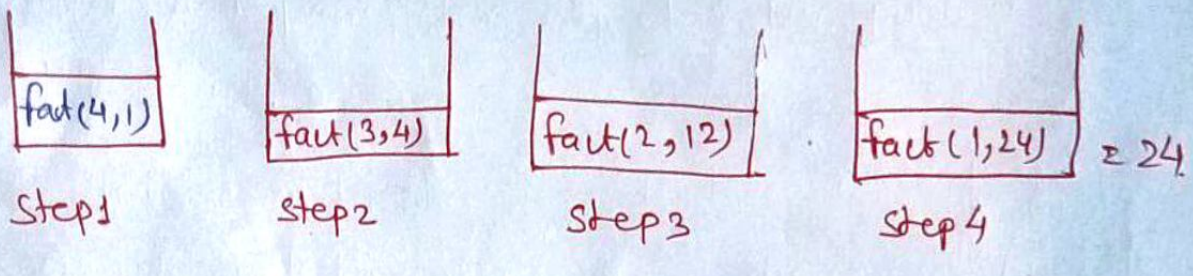
```
void main()
{
  int n, value;
printf("Enter number");
scanf("%d", &n);
  value = fact(n);
printf("factorial = %d", value);
}
int factorial(int n)
{
  if (n==1)
  return(1)
  else
   return(n * factorial(n-1));
}
```





Tail recursion → Tail recursion is defined as a recursive function in which the recursive call is the last statement that is executed by the function.

```
int factorial (int n, int a)
 {
    if (n==1)
    return a;
    else
     factorial (n-1, n*a);
  }
void main()
 {
    int n; fact;
   scanf("%d", &n);
     fact= fatorial (n,1)
  printf("%d", fact);
}
```



Direct & Indirect recursion :→ In direct recursion a function calls itself.

```
f()
{
=
 f();
}
```

Indirect recursion → A function is called indirect recursion when two function call one another mutually.

```
f()
{
=
 g();
=
}
g()
{
=
 f();
}
```

removal of recursion → We can convert recursive function into iteration with the help of following steps:

(i) Converting recursive function to the tail recursion.

(ii) Converting tail recursive function to iteration.