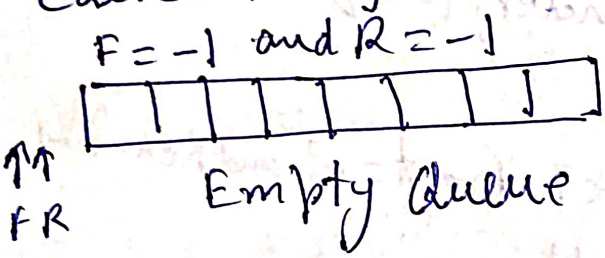
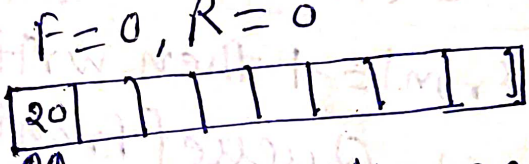
Unit-2

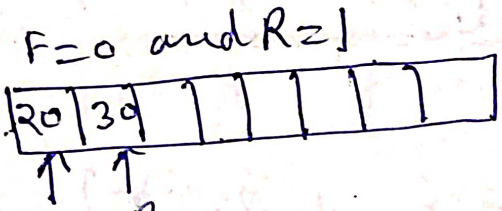
- Queue is logically a first In First Out (FIFO) type of list.
- Queue is a non-primitive linear data structure.
- Queue is a homogeneous collection of elements in which new elements are added at one end called the Rear and the existing elements are deleted from other end called the front.



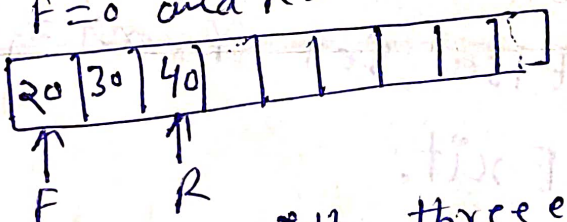
Empty Queue



Queue with one element



Queue with 2-elements



Queue with three elements

~~Del~~

Note:- if F == R then only one element in the Queue.

∴ No of elements in Queue = R - F + 1.

Implemention of Queue:

(i) Using Array

(ii) Dynamic Implementation

Algo for insertion in Queue:

QINSERT (Queue[maxsize], ITEM, Front, Rear)

① If Rear = Maxsize-1 then write overflow & exit.

② If Rear = -1 then set Rear = 0 and Front = 0
Else set Rear = Rear+1.

③ Queue[Rear] = ITEM.

④ Exit.

QDELETE (Queue[maxsize], ITEM, Front, Rear)

① If Front = -1 then write underflow & exit.

② ITEM = Queue[Front].

③ If Front = Rear then set Front = -1 and Rear = -1
Else set Front = Front+1.

④ Exit.

TRAVERSE (Queue[maxsize], Front, Rear, I)

① If Rear = -1 then write Queue is Empty and exit.

② Set I = Front.

③ Repeat steps ④ & ⑤ till I <= Rear

④ Display Queue[I].

⑤ I = I+1

⑥ Exit.

```
#include <stdio.h>
#include <conio.h>
int Queue[50];
int front = -1;
int rear = -1;
int n;
void insert();
void del();
void disp();

void main()
{
  int c;
  char a = 'y';
  clrscr();
  printf("enter size of Queue");
  scanf("%d", &n);
  while (a=='y' || a=='Y')
  {
    printf("enter 1. for Insert in Q");
    printf("\n enter2. for Delete from Q");
    printf("\n enter3. for Display Queue");
    scanf("%d", &c);
    switch()
  {
```

```
case 1:
  insert();
  break;
case 2:
  del();
  break;
case 3:
  disp();
  break;
default:
  printf("enter correct choice");
}
printf("enter y for continue");
fflush(stdin);
scanf("%c",&a);
}
}
void insert()
{ if(Rear == n-1)
  { printf("Overflow");
  else
    { if(Rear == -1)
      { Front = 0;
        Rear = 0;
      }
    else
```

```
Rear = Rear + 1;
printf("enter value");
scanf("%d", &Queue[Rear]);
}
}
void del()
{ int item;
  if (Front == -1)
  printf("Underflow");
  else
  { printf("%d is deleted", Queue[Front]);
    if (Front == Rear)
    { Front = -1;
      Rear = -1;
    }
    else
      Front = Front + 1;
  }
}
void disp()
{ int i;
  if (Rear == -1)
  printf("Queue is empty");
  else { for (i = Front; i <= Rear; i++)
         printf("%d \t", Queue[i]);
       }
}
```

Case 1.

Linked list Implementation of Queue:-

INSERT (Queue, Front, Rear, INFO, NEXT, ITEM, NEW)

① Create a new node & name it NEW.

② If NEW = NULL then write overflow & exit

③ Set NEW -> INFO = ITEM.

④ Set NEW -> NEXT = NULL.

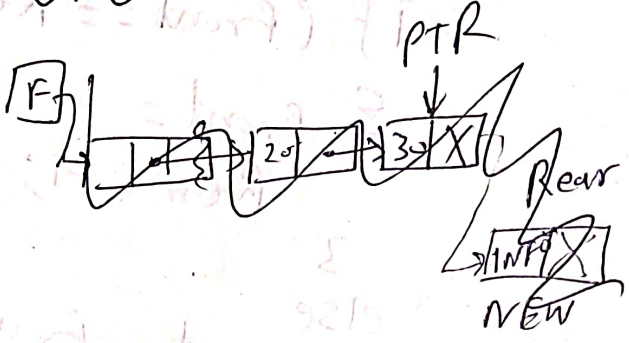
~~⑤ Rear = NEW;~~

⑥ If Front = Null then set Front = NEW & exit.
REAR = NEW & exit

⑦ ~~PTR = Front.~~

⑧ Repeat steps ⑨ while PTR -> NEXT != NULL

⑨ PTR = PTR -> NEXT.



⑩ Rear -> NEXT = NEW

⑪ Rear = NEW

⑫ exit.

```
Void insert()
{ struct node *NEW;
NEW = (struct node *) malloc (sizeof (struct node));
printf ("enter value of node");
scanf ("%d", &NEW->INFO);
NEW -> NEXT = NULL;
Rear = NEW;
if (Front == NULL)
  { Front = NEW; Rear = NEW; }
else
  { PTR = Front;
```

~~while (PTR->NEXT != NULL)~~
~~PTR = PTR->NEXT;~~
~~PTR->NEXT = NEW;~~
~~}~~
~~}~~

Rear->NEXT = NEW;
Rear = NEW;
}
}

DELETE(Queue, Front, Rear, PTR, NEXT, INFO)

① IF Front = NULL then write underflow & exit.
② PTR = Front.
③ Display PTR->INFO is deleted.
④ if Front->NEXT = NULL then
Front = NULL and Rear = NULL. and Exit
⑤ Front = Front->NEXT;
⑥ free(PTR).
⑦ Exit.

```
void delete()
{ struct node *PTR;
  if (Front == NULL)
  printf("Underflow");
  else { printf("%d is deleted", Front->INFO);
         PTR = Front;
         if (Front->NEXT == NULL)
           { Front = NULL;
           } Rear = NULL;
```

```
    else
        front = front->NEXT;
        free(PTR);
    }
}
```

TRAVERSE

(1) If Front = NULL then write Queue is empty & exit.
(2) set PTR = front.
(3) Repeat steps (4) & (5) till PTR ≠ NULL.
(4) Display PTR->INFO.
(5) Set PTR = PTR->NEXT
(6) Exit

```
void traverse()
{
  struct node *PTR;
  PTR = front;
  if (Front == NULL)
    printf("list is empty");
  else {
        while(PTR != NULL)
        { printf("%d \t", PTR->INFO);
          PTR = PTR->NEXT;
        }
      }
}
```

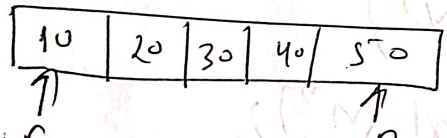
```
#include <stdio.h>
#include <conio.h>
struct node
{
  int INFO;
  struct node *NEXT;
};
struct node *Front, *Rear;
Front = NULL;
Rear = NULL;
void insert();
void delet();
void traverse();
void main()
{ int c;
  char a='y';
  clrscr();
  while(a=='y' || a=='Y')
  { printf("enter 1 for Insert \n");
    printf("enter 2 for Delete \n");
    printf("enter 3 for Display \n");
    scanf("%d", &c);
    switch(c)
   { case 1:
     insert();
     break;
```

```
case 2:
 Delet();
 break;
 case 3:
 disp();
  break;
 default:
   printf("enter correct choice");
}
printf("enter y for continue");
 fflush(stdin);
scanf("%c", &a);
}
}
```

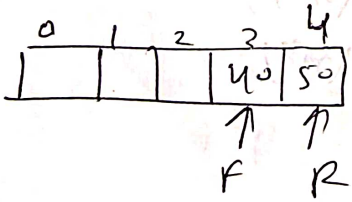
Limitation of simple Queue:-

Overflow condition

if Rear = Maxsize - 1.



Delete Three item then

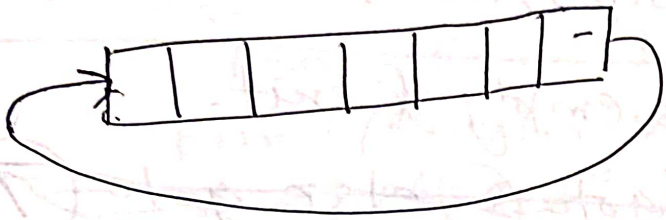


For overflow Rear = Maxsize - 1 = 5 - 1 = 4.

So no item can be inserted. But there is space in Queue for three items.

Circular Queue: Problem of simple Queue can be solved by circular Queue.

- A circular Queue is one in which the insertion of new element is done at the very first location of the Queue if the last location of the Queue is full.



Overflow condition:-

If $F = (R+1) \% Maxsize$ then overflow.

Underflow condition:-

if $F = -1$ or $R = -1$ then underflow

For single element:-

$F = R$ & $F \neq -1$

CQinsert (cQueue[Maxsize], ITEM, Front, Rear)

(1) If Front == (Rear+1) % Maxsize then write overflow & exit.
(2) If Front = -1 then set Front = 0, Rear = 0
ELSE Rear = (Rear+1) % Maxsize.
(3) cQueue[Rear] = ITEM.
(4) Exit.

Case 2:

CQDELETE (CQueue[Maxsize], ITEM, Front, Rear)

① If Front = -1 then write underflow & exit.

② ITEM = CQueue[Front].

③ If Front = = Rear then set Front = -1, Rear = -1

Else Front = (Front + 1) % Maxsize ;

④ Exit.

CQTRAVERSE (CQ[Maxsize], ITEM, Front, Rear)

① If R = -1 then write Q is empty & exit.

② ~~If (F <= Rear) then goto (5) else goto (7)~~

~~Set I = Front and Repeat step~~

③ Set I = Front

④ Repeat step 5 & 6 till I = Rear

⑤ Display Queue[I];

⑥ I = (I + 1) % Maxsize

⑦ ~~If (R > Rear) then goto (8) & else goto (1)~~

⑧ Set I = Front

⑨ ~~Repeat step 10 & 11~~ till I <= n-1

⑩ Display Queue[I]

⑪ I = I + 1

⑫ Exit

⑬ I = 0

⑭ Repeat step (14) & (15) till I <= Rear

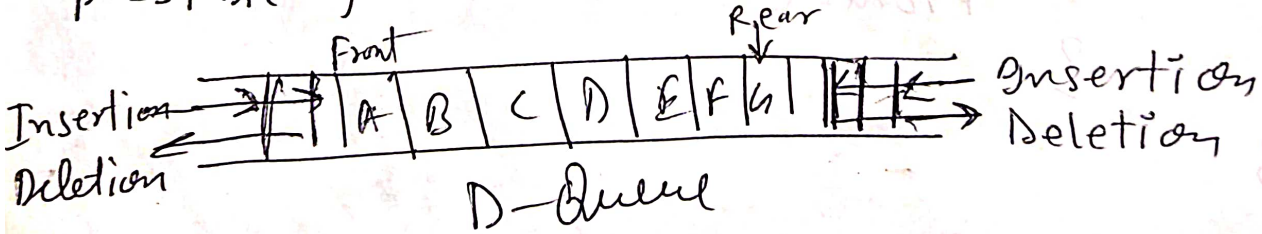
```
void insert()
{ if (Front == (Rear+1) % n)
   printf("Overflow");
 else if (Rear == -1)
      { Front = 0;
        Rear = 0;
      }
    else
      Rear = (Rear+1) % n;
    printf("enter value");
    scanf("%d", &Queue[Rear]);
  }
}
void del()
{ if (front == -1)
   printf("Underflow");
  else {
       printf("%d is deleted", Queue[front]);
       if (front == Rear)
         { front = -1;
           Rear = -1;
         }
       else
         front = (front+1) % n;
       }
}
```

```
void display()
{ int i;
  if (Rear == -1)
  printf("Queue is empty");
else{
  for(I=Front; I<=Rear; I++)
    printf("%d \t", Queue[I]);
```

```
else
for(i=front; i<=rear-1; i++)
printf("%d\t", Queue[i]);
for(i=0; i<rear; i++)
printf("%d\t", Queue[i]);
printf("%d", Queue[i]);
}
}
```

Double ENDED Queue (D-Queue) :-

In a D-Queue, both insertion & deletion operations are performed at either end of the queue. That is, we can insert a element from the rear end or front end. Also deletion is possible from either end.



Types of D-Queue :-

(i) Input - Restricted D-Queue : Elements can be added at only one end but we can delete the element from both ends.

(ii) Output - Restricted D-Queue : Elements can be deleted only from one end but allows insertion at both ends.

Possible operation performed on D-Queue are:

(1) Add an element at the rear end
(2) Add an element at the front end.
(3) Delete an element from the front end.
(4) Delete an element from the rear end.
(5) Display elements of Queue.

Insert Front

(1) Input DATA to be inserted.
(2) If (Front == 0 ~~|| (Rear == n-1) || (Front == Rear+1)~~)
(3) then write you can't enter from front & exit.
(4) If Front = -1 then Front = 0 & Rear = 0 Else Front = Front - 1;
(5) Queue [Front] = DATA.
(6) Exit

INSERTREAR :-

(1) Input DATA to be inserted.
(2) If (Rear == n-1) then write you can't enter from rear.
(3) If (Rear == -1) then front = 0 and Rear = 0. Else rear = rear + 1.

④ Queue[Rear] = DATA.

⑤ Exit.

DELETEFront (Queue[n], Front, Rear, DATA)

① If Front = -1 then write Underflow & exit.

② DATA = Queue[Front]

③ If Front == Rear then

Front = -1 and Rear = -1

Else Front = Front + 1

④ ~~Enter DATA~~ exit.

DELETE Rear (Queue[n], Front, Rear, DATA)

① If Rear = -1 then write underflow & exit.

② DATA = Queue[Rear].

③ If (Front == Rear) then

Front = -1 and Rear = -1

Else

Rear = Rear - 1.

④ Exit.

TRAVERSE (Queue[n], Front, Rear, DATA)

① If Rear = -1 then write Q is empty & exit.

② set I = Front.

③ Repeat steps ④ & ⑤ till I <= Rear

④ Display Queue[I].

⑤ I = I + 1

⑥ Exit

Circular Queue using linked list:

INSERT (Queue, Front, Rear, INFO, NEXT, ITEM, NEW).

1. Create a new node and name it NEW.
2. If NEW = NULL then write overflow & exit.
3. set NEW -> INFO = ITEM.
4. If Front = NULL then set
   Front = NEW.
   Rear = NEW.
   & Rear -> NEXT = ~~NEW~~ front. & exit.
5. Rear -> NEXT = NEW.
6. Rear = NEW.
7. Rear -> NEXT = Front.
8. Exit.

```
void insert ()
{ struct node *NEW;
NEW=(struct node *) malloc( sizeof(struct node));
printf("enter value of node");
scanf("%d", &NEW->INFO);
 if (Front == NULL)
   { Front = NEW;
     Rear = NEW;
     Rear->NEXT = Front;
   }
else { Rear -> NEXT = NEW;
       Rear = NEW;
       Rear -> NEXT = Front;
}  }
```

DELETE(Queue, Front, Rear, PTR, NEXT, INFO)

① If Front = NULL then write underflow & exit
② PTR = Front;
③ Display (PTR->INFO) is deleted.
④ If (Front == Rear) then set
Front = NULL & Rear = NULL, free(PTR) & exit.
⑤ Front = Front -> NEXT.
⑥ Rear -> NEXT = Front.
⑦ Free(PTR).
⑧ Exit.

```
void del()
{
  struct node *PTR;
  if (Front == NULL)
    printf ("Underflow");
  else
    {printf ("%d is deleted", Front->INFO);
     PTR = Front;
     if (Front == Rear)
     {
       Front = NULL;
       Rear = NULL;
     }
     else{
       Front = Front->NEXT;
       Rear->NEXT = Front;
     }
    free(PTR);
  }
}
```

TRAVERSE

① IF Front = NULL then write Queue is empty & exit.

② Set PTR = Front.

~~③ If Front = Rear then Disply PTR->INFO & exit.~~

~~④ Repeat steps ⑤ & ⑥ till PTR.~~

~~③ Printf Display PTR->INFO.~~

~~④ PTR = PTR->NEXT.~~

⑤ Repeat steps ④ & ⑤ till ~~PTR != Front.~~ PTR->NEXT != Front,

④ Display PTR->INFO

⑤ PTR = PTR->NEXT

⑥ Display PTR->info

⑦ exit.

```
void traverse()
{ struct node *PTR;
  if (Front == NULL)
   printf("Queue is empty");
  else
      { PTR = Front;
        printf("%d\t", PTR->INFO);
        PTR = PTR->NEXT;
        while (PTR->next != Front)
        { printf("%d\t", PTR->INFO);
          PTR = PTR->NEXT;
        }
        printf("%d", PTR->INFO);
  }
}
```

D-Queue using Array:-

```
void insertFront()
{ if (Front==0)
  printf("you can't enter from front");
 else{ if (Front==-1)
       { Front=0;
         Rear=0;
       }
     else
       Front=Front-1;
     printf("enter value");
     scanf("%d",&Queue[Front]);
   }
}

void insertRear()
{ if (Rear==n-1)
  printf("you can't enter from Rear");
 else{ if (Rear==-1)
       { Front=0;
         Rear=0;
       }
     else
     Rear=Rear+1;
     printf("enter value");
     scanf("%d",&Queue[Rear]);
   }
}
```

```
void delfront ()
{ if (Front == -1)
   printf ("Underflow");
  else { printf ("%d is deleted", Queue[Front]);
         if (Front == Rear)
         { Front = -1;
           Rear = -1;
         }
         else
           Front = Front+1;
       }
}
void delRear()
{ if (Rear == -1)
   printf ("Underflow");
  else { if (Front == Rear)
         { Front = -1;
           Rear = -1;
         }
         else
         Rear = Rear-1;
       }
}
void traverse()
{ int i;
  if (Rear == -1)
  printf ("Queue is empty");
  else { for(i = Front; i <= Rear; i++)
         printf ("%d\t", Queue[i]);
       }
}
```

printf ("%d is deleted", Queue[Rear]);

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 were added to the queue.

Representation of the priority Queue:-

(1) One-way linked list

(2) ~~Multiple~~ Array-Representation [multiple Queue Representation]

One-way linked list:

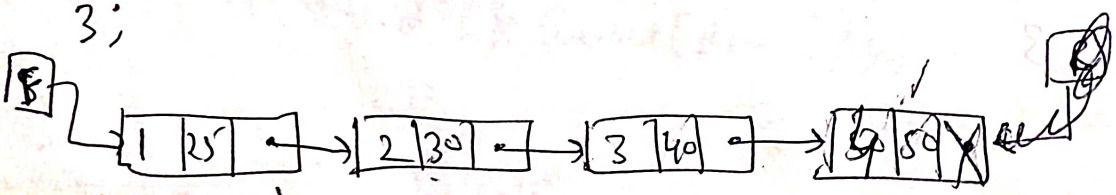
In this each node have three fields -

(1) An information field. INFO

(2) A priority number PRN

(3) A link to next node.

```
struct node
{
    int INFO;
    int PRN;
    struct node *NEXT;
};
```



DELPROCESS Algo :- This algo deletes & process the first element in a priority queue which appears in memory as a one way list.

① Set ITEM = START->INFO.
② Delete First node from the list.
③ Process ITEM
④ Exit.

INSERT Algo :- This algo adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(a) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
(b) If no such node is found, insert ITEM as the last element of the list.

Array Representation :- we use separate queue for each level of priority. Each queue will appear in its own circular array and must have its own pair of pointers, Front & Rear. A two-dimensional array is used for this purpose.-

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1

	1	2	3	4	5	6
1		AA				
2	BB	XX	YY			
3						
4	ZZ				DD	EE

Insert Algo:- This algo adds an ITEM with priority p number m to a priority queue maintained by a two-dimensional array Queue.

(1) Insert ITEM as the rear element in row m of Queue.
(2) Exit

DELPROCESS Algo - This algo deletes and processes the first element in a priority queue maintained by a two-Dimensional array Queue.

(1) Find the smallest K such that k[Front] ≠ NULL. [Find the first Non-empty Queue]
(2) Delete and process the front element in row K of Queue.
(3) Exit

Implementation of DE-Queue using Linked list:

```
struct node
{ int INFO;
  struct node *PREV, *NEXT;
};
struct node *f = NULL;
struct node *r = NULL;

void addR()
{ struct node *NEW;
NEW=(struct node *)malloc(sizeof(struct node));
```

```
printf("enter value of node");
scanf("%d", &NEW->INFO);
NEW->NEXT = NULL;
if (r==NULL)
{ f = NEW;
  r = NEW;
  NEW->PREV=NULL;
}
else
{ r->NEXT = NEW;
  NEW->PREV=r;
  r = NEW
}
}
void delf()
{ struct node *PTR;
if (f==NULL)
printf("underflow");
else
{ printf("%d is deleted", f->INFO);
  PTR=f;
  if (f==r)
  { f=NULL;
    r=NULL;
  }
  else
  { f=f->NEXT;
    f->PREV=NULL;
  }
  free(PTR);
}
}
```

```
void delr()
{ struct node *PTR;
  if(r==NULL)
  printf("Under flow");
  else
    { printf("%d is deleted", r->INFO);
      PTR=r;
      if(f==r)
        { f=NULL;
          r=NULL;
        }
      else
        { r=r->PREV;
          r->NEXT=NULL
        }
      free(PTR);
    }
}
void addf()
  { struct node *NEW;
NEW=(struct node *)malloc(sizeof(struct node));
 printf("enter value of node");
 scanf("%d", &NEW->INFO);
 NEW->PREV=NULL;
 if(f==NULL)
   { f=NEW;
     r=NEW;
     NEW->NEXT=NULL;
   }
 else
   { f->PREV=NEW;
     NEW->NEXT=f;
     f=NEW;
   }
}
```

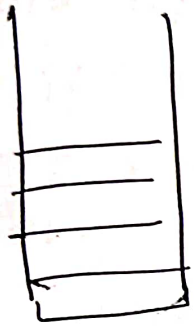
Application of Queue:-

1. Round Robin technique for processor scheduling is implemented using Queue.
2. When the jobs are submitted to a NW printer, job sent to a printer are placed in Queue.
3. BFS algo uses Queue.
4. Every real-life line is a Queue. Exp:- Railway station line, ticket counter at cinema Hall.

```
void disp()
{ struct node *PTR;
  PTR=f;
  while(PTR!=NULL)
  { printf("%d", PTR->INFO);
    PTR=PTR->NEXT;
  }
}
```

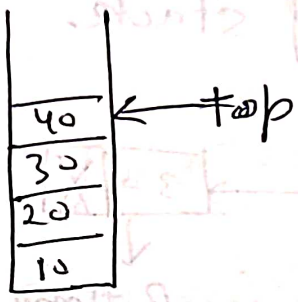
Stack:

- Stack is logically a Last In First Out LIFO type of List.
- Stack is a non-primitive linear data structure.
- Stack is a homogeneous collection of elements in which insertion and deletion is done from top of the Stack.



Empty stack

top = -1



Stack Abstract data type:- Unit no-1 Page 7-8.

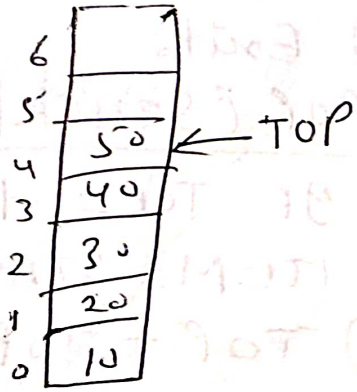
Representation of Stack:-

(i) Array Representation of stack: The stack is maintained by a linear array STACK, a pointer variable TOP which contains the location of top element.

Under flow condition **top = -1**

Over flow condition **top = n-1**.

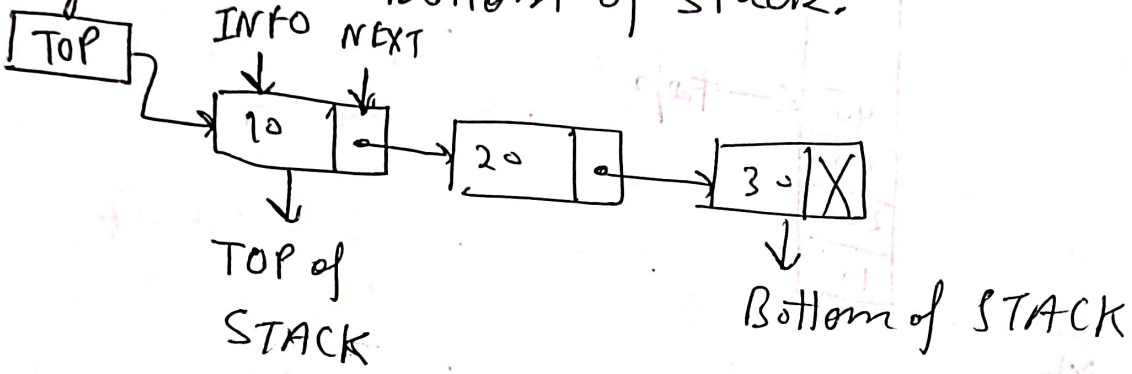
where n is size of STACK.



Linked list Representation of STACK:

The linked list representation of a stack is implemented using a singly linked list. The node have two field - | INFO | NEXT |

(i) INFO field holds the elements of STACK
(ii) NEXT field holds the address of next node.
The TOP pointer tells the top element of STACK and null pointer of the last node in the list signals the bottom of stack.



Implementation of STACK using Array:

PUSH (STACK[MAXSIZE], TOP, ITEM)

(1) If TOP = MAXSIZE-1 then write overflow & exit
(2) Read ITEM.
(3) Set TOP = TOP+1.
(4) Set STACK[TOP] = ITEM
(5) Exit.

POP (STACK[MAXSIZE], TOP, ITEM)

(1) If TOP = -1 then write underflow & exit.
(2) ITEM = STACK[TOP]. ~~is deleted.~~
(3) TOP = TOP-1
(4) ~~End~~. ITEM is deleted. (5) exit.

DISPLAY(STACK(MAXSIZE), TOP, ITEM)

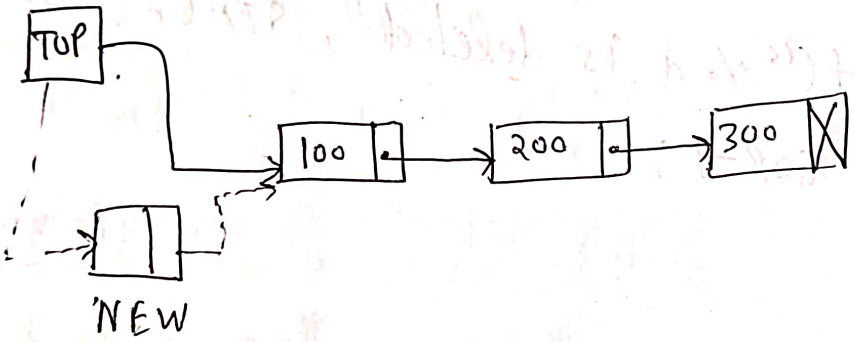
① If TOP=-1 then write STACK is empty.
② Set I = TOP.
③ Repeat steps ④ & ⑤ till I >=0.
④ Display STACK[I].
⑤ ~~TOP=TOP-1~~ I = I-1
⑥ Exit.

```
void Pop()
{ if (TOP==-1)
   printf("Underflow");
  else
   { printf("%d is deleted", STACK[TOP]);
     TOP=TOP-1;
   }
}
void Push()
{ if (TOP==n-1)
  printf("Overflow");
 else
    { TOP = TOP+1;
      printf("enter value");
      scanf("%d", &STACK[TOP]);
    }
}
```

```
void disp()
{ int i;
 if(TOP==-1)
 printf("STACK is empty");
 else
    { for(i=TOP; i>=0; i--)
       printf("%d\t", STACK[i]);
    }
}
```

Implementation of STACK using linked list:-

Push:-



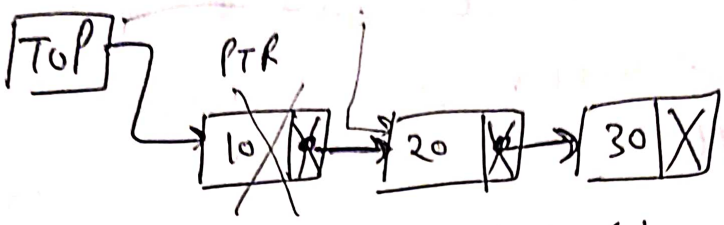
PUSH (STACK, TOP, INFO, NEXT, ITEM, NEW) :-

1. Create a new node and name it NEW.
2. If NEW = NULL then write Overflow & exit.
3. Set NEW -> INFO = ITEM.
4. NEW -> NEXT = TOP
5. TOP = NEW
6. Exit

```
void Push()
{ struct node *NEW;
NEW=(struct node *)malloc(sizeof(struct node));
```

```
printf("enter value of node");
Scanf("%d", &NEW->INFO);
NEW->NEXT = TOP;
TOP = NEW;
}
```

POP :-



POP (STACK, TOP, INFO, NEXT, PTR)

(1) If TOP = NULL then write Underflow & exit.
(2) PTR = TOP.
(3) Display (PTR->INFO) is deleted.
(4) TOP = TOP->NEXT.
(5) free (PTR).
(6) Exit

```
void pop()
{ struct node *PTR;
  if( TOP==NULL)
  printf("Underflow");
else { printf("%d is deleted", TOP->INFO);
       PTR=TOP;
       TOP= TOP->NEXT;
       free(PTR);
     }
}
```

DISPLAY (STACK, PTR, INFO, NEXT, TOP)

① If TOP = NULL then write STACK is empty.
② Set PTR = TOP.
③ Repeat steps ④ & ⑤ till PTR != NULL
④ Display PTR -> INFO.
⑤ PTR = PTR -> NEXT.
⑥ Exit.

```
void display()
{
  struct node *PTR;
  PTR = TOP;
 if (TOP == NULL)
 printf("STACK is empty");
else
    {
      While (PTR != NULL)
      {printf("%d\t", PTR->INFO);
       PTR = PTR->NEXT;
      }
    }
}
```

Application of STACK:

① Undo in Text Editor
② Web browser History.
③ conersion of infix Expression to Postfix & Prefix form.
④ Evaluation of Postfix & Prefix form
⑤ Checking the validity of an Arithmetic Expression.
⑥ Used in Recursion.
⑦ Used in Back Tracking Algo.

Notation for Arithmetic Expression:

There are basically three type of notations.

(i) Infix Notation
(ii) Prefix Notation
(iii) Postfix Notation

Infix Notation: In this notation, operator is placed between two operands.

For example -

A+B, C-D, E*F

Prefix Notation:- In this Notation, operator is placed before the operands.

Exp:-

+AB, -CD, *EF

Postfix Notation: In this Notation, operator is placed after the operands.

AB+, CD-, EF-

Operator precedence:

Exponential operator (^), Highest Precedence
Multiplication (*), Division (/). Next Precedence
Subtraction, Addition, Lowest Precedence.

The operators with same priority are evaluated from left to right.

Exp:- $2^3 + 5 * 4 ^ 2 - 18/9$

$= 8 + 5 * 16 - 18/9$

$= 8 + 80 - 2$

$= 86$ ans

Transforming Infix expression (Q) into Postfix (P):

(1) Push "(" onto STACK and add ")" to the end of Q.

(2) Scan Q from left to right & repeat steps 3 to 6 for each element of Q until the STACK is empty.

(3) If an operand is encountered, add it to P.

(4) If a left Parenthesis is encountered, push it onto STACK.

(5) If an operator (X) is encountered then

(a) Repeatedly pop from STACK and add to P each operator which has the same or higher precedence than (X).

(b) add (X) to stack.

(6) If a right parenthesis is encountered then

(a) Repeatedly pop from STACK and add to P each operator untill a left parenthesis is encountered.

(b) Remove left parenthesis [Don't add to P]

(7) Exit.

Exp:- A + (B * C - (D / E ↑ F) * G) * H. (Q)

Convert it Postfix.

ans:- A + (B * C - (D / E ↑ F) * G) * H

Scanned Symbol	STACK	P
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB

Scanned Symbol	STACK	P
C	(+(*	ABC
*	(+(*	ABC
(	(+(*-(	ABC
D	(+(*-(	
—	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
↑	(+(-(/↑	ABC*DE
F	(+(-(/↑	ABC*DEF
)	(+(-	ABC*DEF↑/
*	(+(-*	ABC*DEF↑/
G	(+(-*	ABC*DEF↑/G
)	(+	ABC*DEF↑/G*-
*	(+*	ABC*DEF↑/G*-
H	(+*	ABC*DEF↑/G*-H
Empty		ABC*DEF↑/G*-H*+

Ans

Transforming Infix expression (Q) into Prefix (P):-

(a) Push) onto stack and add (at beg of Q

(b) Scan Q from Right to Left & Repeat steps 2 to 5 until the Stack is empty.

(2) If it is operand, add it to P.

(3) If it is closing parenthesis), push it onto STACK.

(4) If it is an operator (X) then -

(i) If STACK is empty, push (X) on STACK

(ii) If the Top of STACK is closing parenthesis, push (X) onto STACK

(iii) If it has same or higher priority than the top of STACK, push (X) on STACK.

(iv) Else Pop the operator from the STACK & add it to P, repeated steps which has higher precedence than (X). (b) add (X) to stack

(5) If it is a opening parenthesis, pop operator from STACK and add them to P until a closing parenthesis is encountered

(6) ~~If there is no O/P in Q then unstack the remaining operators and add them to P.~~

(7) Reverse the P.

Exp:- 2*3/(2-1)+5*(4-1) convert it into Prefix.

Scanned Symbol	STACK)	P
)	))	
1	))	1
-	))-	1
4	))-	14
(	)	14-
*	)*	14-
5	)*	14-5
+	)+	14-5*
)	)+)	14-5*1
1	)+)	14-5*1
-	)+)-	14-5*1
2	)+)-	14-5*12
(	)+	14-5*12-
/	)+/	14-5*12-
3	)+/	14-5*12-3
*	+/*	14-5*12-3
2	+/*	14-5*12-32
Empty		14-5*12-32*+/+

Reverse P = +/*-21*5-41

Ans

Reverse P = +/*23-21*5-41

Ans 2

Evaluation of Postfix expression (P):-

① Add a right parenthesis ")" at the end of P.
② Scan P from left to right and repeat step ③ & ④ for each element of P untill the ")" sentinel is encountered.
③ If an operand is encountered, put it on STACK.
④ If an operator ⊗ is encountered then
(a) Remove the two top elements of stack, where A is the top element and B is the next top element.
(b) Evaluate B ⊗ A.
(c) Place the result of (b) back on stack.
⑤ Set Value equal to the top element of stack.
⑥ Exit.

Exp:- 5, 6, 2, +, *, 12, 4, /, − Evaluate it.
5, 6, 2, +, *, 12, 4, /, −, ")"

Scan Symbol	STACK
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	37
")"	37

value = stack[top]
= 37

Checking validity of an expression containing nested parenthesis:

Take a boolean variable valid which will be true if the expression is valid and will be false if the expression is invalid. Initially valid is true.

(1) Scan the symbols of expression from left to right for each symbol. Repeat steps 2 & 3

(2) If the symbol is a left parenthesis then push it onto STACK

(3) If the symbol is right parenthesis then
if the stack is empty then valid = false.
else pop an element from STACK
if popped parenthesis does not match the parenthesis being scanned then valid = false.

(4) After scanning all the symbols if the stack is not empty then make valid = false.

(5) Check the value of valid if it true then write expression is valid else invalid.

(6) Exit

Exp:- [(A+B)-{c+D}]-[F+G]

Scan symbol	STACK	valid = True
[	[	
(	[(	
A	[(	
+	[(	
B	[(	
)	[	
-	[	
{	[{	
c	[{	
+	[{	
D	[{	
}	[	
-	Empty	
[	[	
F	[	
+	[	
G	[	
]	Empty	

Ans:- Valid = True so Expression is valid.

Recursion: The function which call itself again and again is known as Recursive function.
The function will call itself as long as the base condition is not satisfied.

Principles of Recursion:-

① Base case: Base case is the terminating condition for recursive function.

② If condition: If condition in the recursive program defines the terminating condition.

③ Each time a function call itself it must be closer to base case in some sense to a solution.

④ The recursive function calls will not be executed immediately.

⑤ The initial parameter input value pushed on to the stack.

⑥ Each time a function is called a new set of local variable and formal parameters are again pushed on to the stack and execution starts from the beginning of the function using changed new value. This process is repeated till a base condition is reached.

⑦ Once a base condition or reached the recursive function calls poping elements from

STACK, and returns a result to the previous value of function.

(8) A sequence of returns ensures that the solution to the original problem is obtained.

Exp:- Recursive program for factorial.

```
void main()
{ int n, value;
  printf("enter number");
  scanf("%d", &n);
  if (n<0)
    printf("No factorial of negative number");
  else if (n==0)
    printf("Factorial of zero is 1");
  else
    { value = fact(n);
      printf("factorial of %d = %d", n, value);
    }
}
factorial (int m)
{ int f;
  if (m == 0)
    return (1);
  else f = m * fact(m-1);
  return (f);
}
```

Factorial · 4 = 24.

fact(4) = 4 X fact(3)

→ fact(3) = 3 X fact(2)

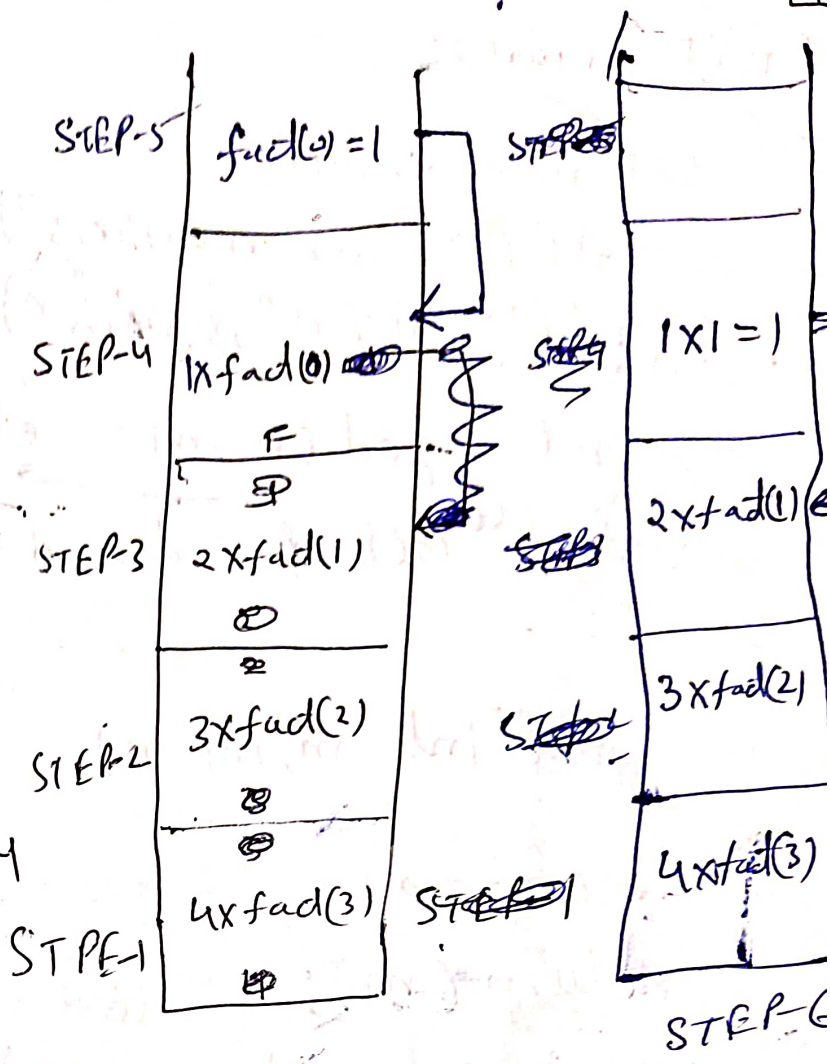
→ fact(2) = 2 X fact(1)

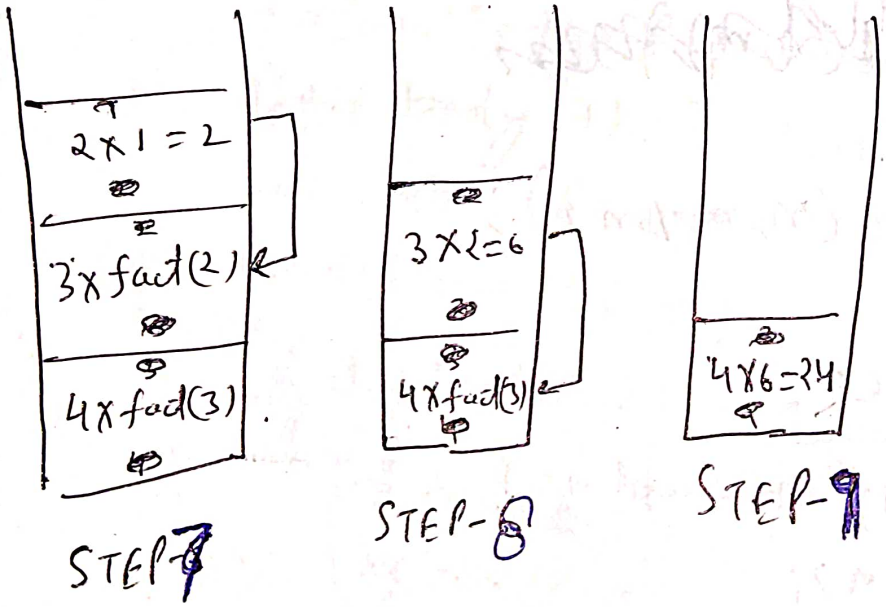
→ fact(1) = 1

fact(2) = 2 X fact(1) = 2

fact(3) = 3 X fact(2) = 6

fact(4) = 4 X fact(3) = 24





Greatest common Divisor of Two numbers:

$$GCD(m,n) = \begin{cases} GCD(n,m) & \text{if } (n>m) \\ m & \text{if } n=0 \\ GCD(n, m\%n) & \text{if } m>n \end{cases}$$

Program for GCD:—

```
void main()
{
  int m,n, result;
  printf("enter numbers");
  scanf("%d %d", &m, &n);
  if (m>n)
  result = gcd(m,n); else result = gcd(n,m);
  printf("GCD of %d and %d = %d", m, n, result);
}
int gcd(int m, int n)
{
  if (n==0)
    return(m);
  else if(n>m)
    return(gcd(n,m))
  else
      return(gcd(n, m%n));
}
```

Fibonacci Sequence:

$$F_n = \begin{cases} n & \text{for } n=0 \text{ and } n=1 \\ F_{n-1}+F_{n-2} & \text{for } n>1 \end{cases}$$

Program for Fibonacci :—

```
void main()
{
  int n, result;
  printf("Enter the term no. which U want to
                                  calculate");
```

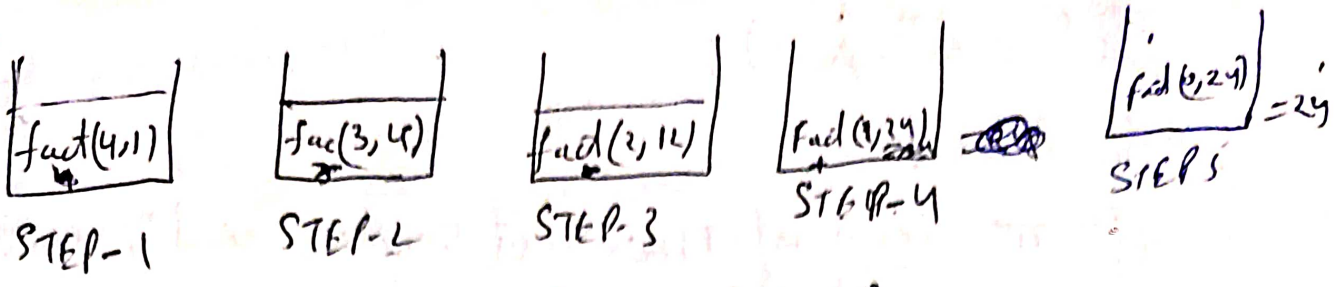
```
scanf("%d", &n);
result = fib(n);
printf("nth term of Fibonacci Series = %d", result);
}
int fib(int n)
{
  if(n==0 || n==1)
  return(n);
  else
  return(fib(n-1) + fib(n-2));
}
```

- we use STACK to maintain all the functions but here it will not append new function in STACK but it will overwrite the value of previous function with current one. So function call time & stack implementation time will be reduced & it will give better performance.

Tail Recursion:- The process in which function calls itself in function body is called as recursion. But when this called function is the last executed statement in the function body then it is called tail recursion. Hence we take the return value as one parameter of function itself.

```
void main()
{ int n, value;
printf("enter value");
scanf("%d", &n);
if(n<0) printf("No factorial of -ve no");
else if(n==0)
  printf("Factorial of zero is 1");
else
  { value = factorial(n,1);
  printf("Factorial of %d = %d", n, value);
}
```

```
int factorial(int n, int fact)
{ if(n==0)
  return(fact);
  else
  factorial(n-1, n*fact);
}
```



Direct & Indirect Recursion:

In Direct Recursion a function call itself.

Exp:-

```
abc()
{
  =
  abc();
  =
}
```

~~In Indirect~~

Indirect Recursion:-

A function is called Indirect Recursion when two function call one another mutually.

Exp:-

```
abc()
{
  =
  xyz();
  =
}
xyz()
{
  =
  abc();
  =
}
```

Removal of Recursion:

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

(i) converting recursive function to the tail recursive.

(ii) converting Tail recursive function to iteration. Exp:- Factorial of number.

Simulating Recursion:- simulating

Recursion means to simulate the recursive mechanism by using non-recursive technique. To create a correct solution in non-recursive language the conversion from recursive technique to non-recursive technique must be correct.

In C-language, a recursive solution is more expensive than a non-recursive solution in time as well as in space. So reduce the expense of design and then simulate to the non-recursive solution to use in actual practice.

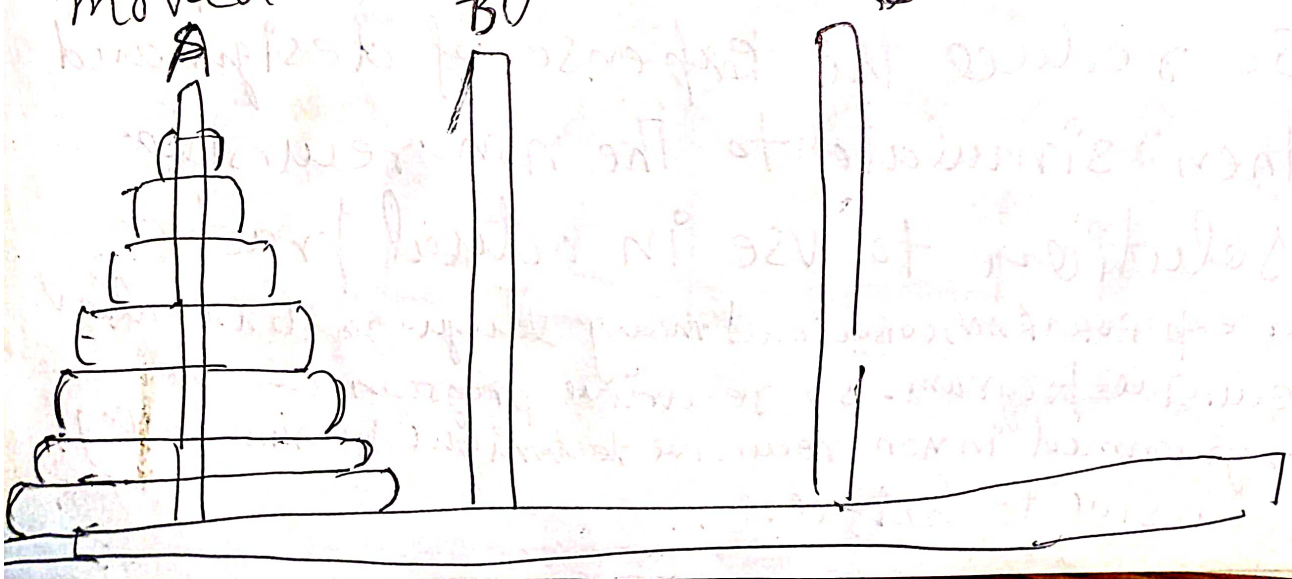
For exp FORTRAN, COBOL and many languages do not allow recursive program. So recursive program can be programmed in non-recursive technique by simulating recursive technique.

Tower of Hanoi problem:

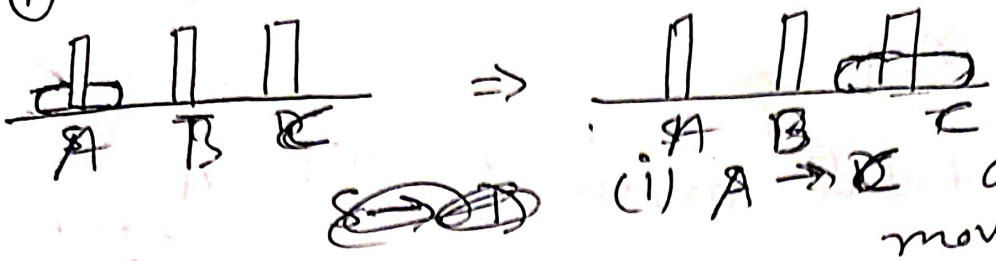
In this problem, there are n disks of different sizes and there are three pegs A, B and C. All the n disks are placed on a peg A in such a way that a larger disk is always below a smaller disk. The other two pegs are initially empty. The aim is to move the n disks to the second peg C using the third peg B as a temporary storage.

The rules for the movement of disks are as follows:

- Only one disk could be moved at a time.
- A larger disk must never be stacked above a smaller one.
- Only the top disk on any peg may be moved to any other peg.

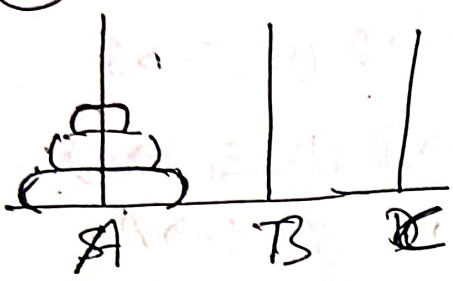


① if n=1. then move top disk from A to C

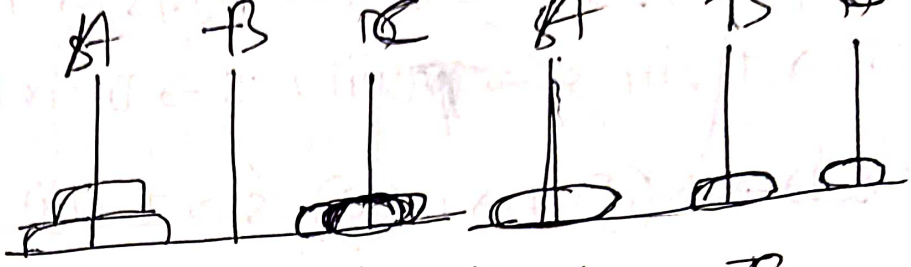


(i) A → C only one step movement needed.

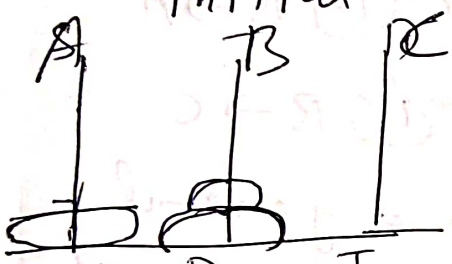
② if n=3



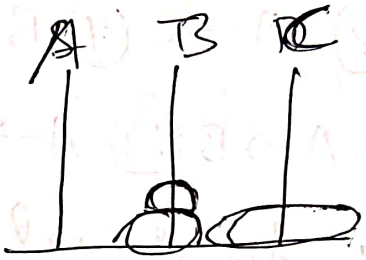
initial



(i) A → C (ii) A → B

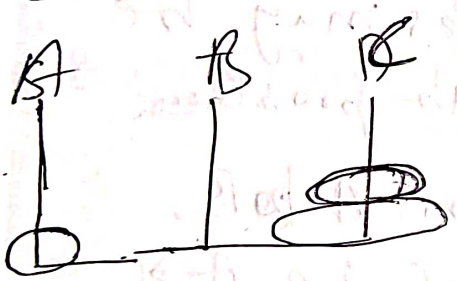


(iii) C → B



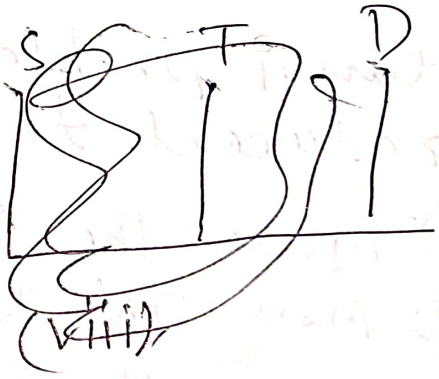
(iv) A → C

(v) B → A



(vi) B → C

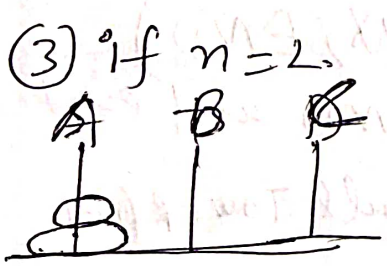
(vii) A → C



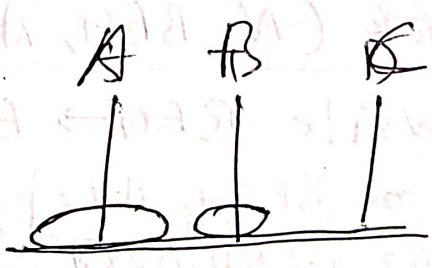
(viii)

So Total 7 movement needed.

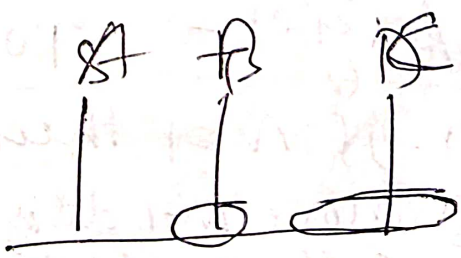
③ if n=2.



Initial



(i) A → B



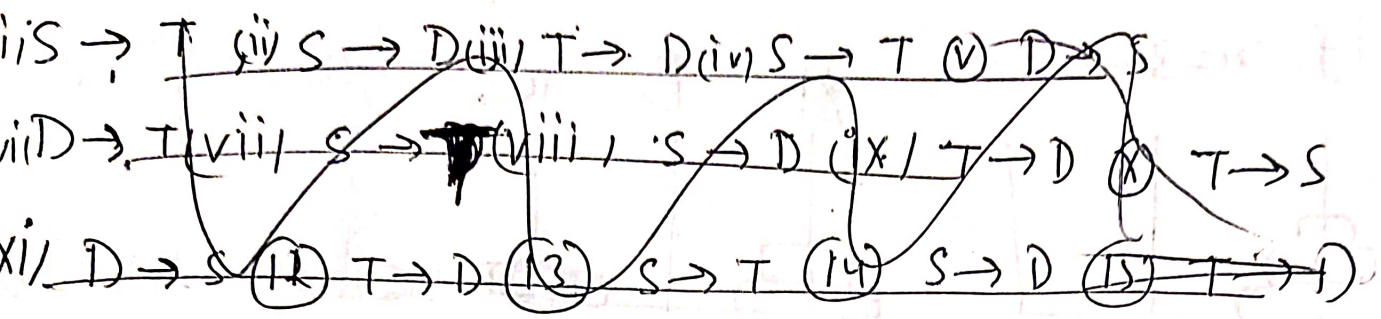
(ii) A → C

C → D (iii) B → C

So Total 3 movement needed.

So if n disk then total movement needed $= 2^n - 1$ ∞

for $n = 4$ total movement $= 2^4 - 1 = 15$



① A → B ② A → C ③ B → C ④ A → B ⑤ C → A

⑥ C → B ⑦ A → B ⑧ A → C ⑨ B → C ⑩ B → A

⑪ C → A ⑫ B → C ⑬ A → B ⑭ A → C ⑮ B → C

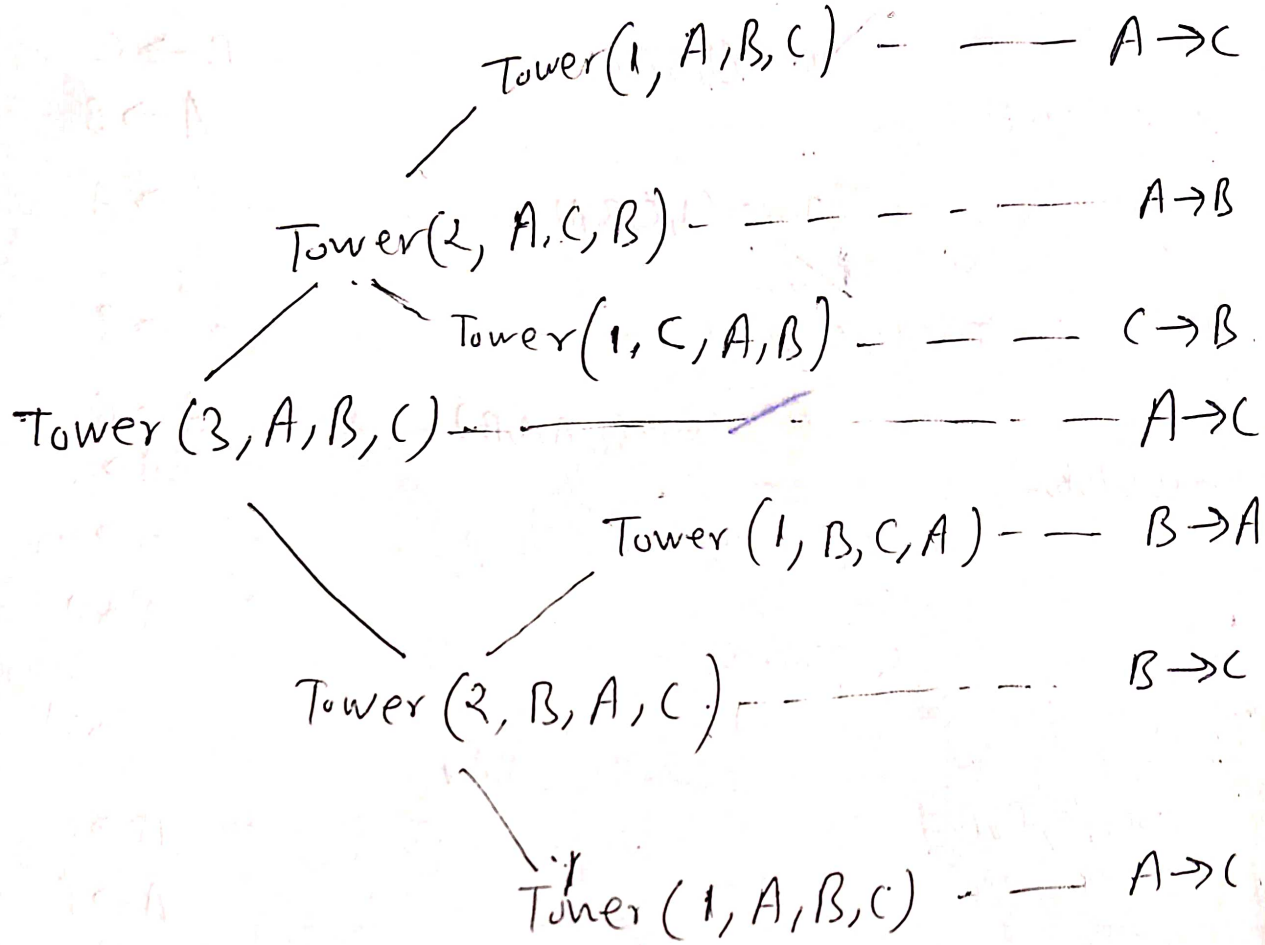
We use recursion to develop a general solution. The Solution to the Tower of Hanoi problem for $n > 1$ disks may be reduced to the following sub-problem.

① Move the Top $n-1$ disk from A to B.
② Move The Top disk from A to C i.e. A → C.
③ Move the Top $n-1$ disks from B to C.

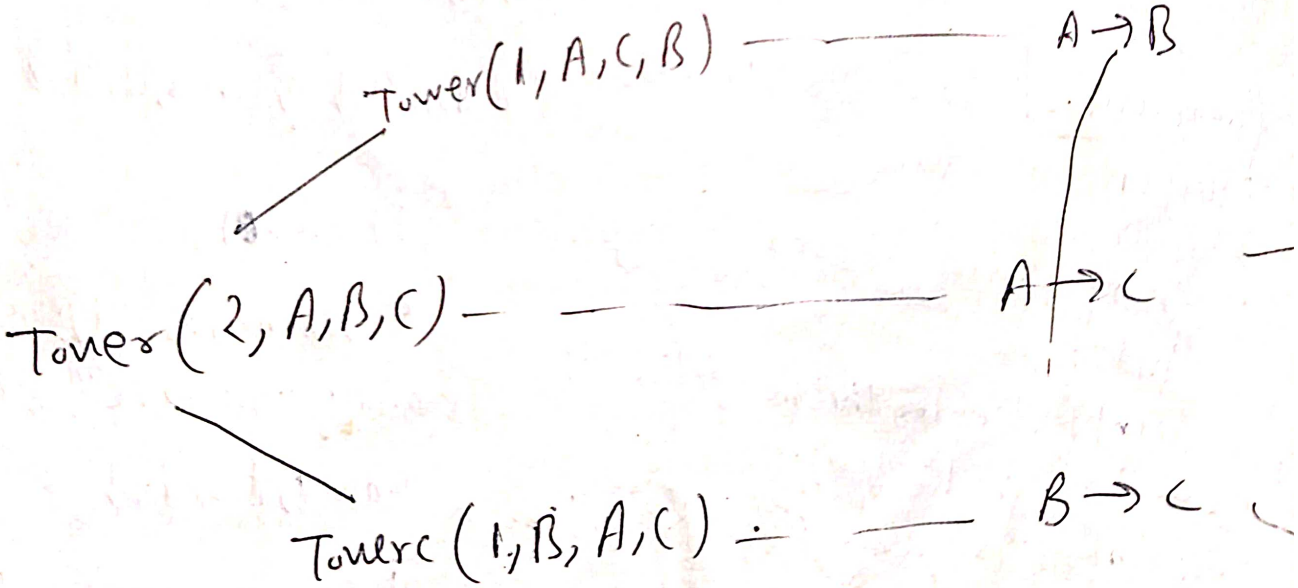
Algo:- TOWER (N, BEG, AUX, END)

① If $N = 1$ then write BEG → END and exit
② [Move N-1 disk from BEG to AUX] Call TOWER(N-1
call TOWER(N-1, BEG, END, AUX).
③ write BEG → END.
④ [Move N-1 Disk from AUX to END]
call TOWER(N-1, AUX, BEG, END)
⑤ Exit

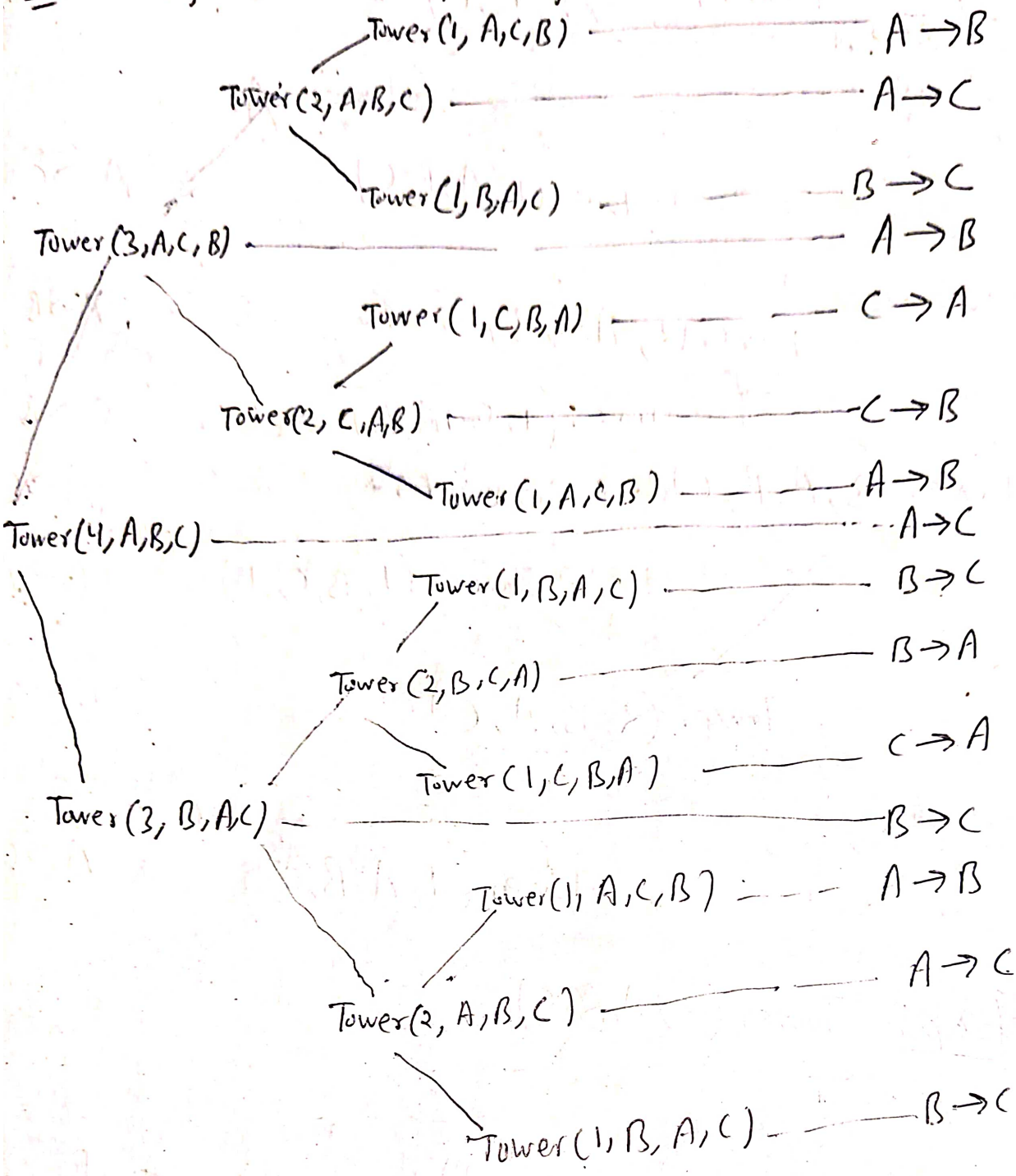
Exp:- Tower (3, A, B, C)



Exp:- Tower (2, A, B, C)



Q) n = 4, Solve the Tower of hanoi.



Program of Tower of Hanoi :-

```
main()
{
    char s='S', t='T', d='D';
    int n;
    printf("enter the no. of disk");
    scanf("%d", &n);
    printf("sequence is:\n");
    toh(n, s, t, d);
}
```

```
toh(int n, char s, char t, char d)
{
    if (n>0)
    {
        toh(n-1, s, d, t);
        printf("%c -> %c", s, d);
        toh(n-1, t, s, d);
    }
    else
        return(0);
}
```