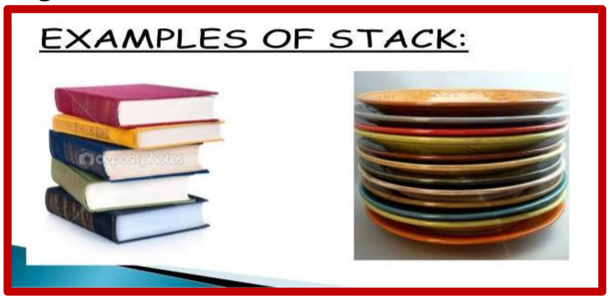
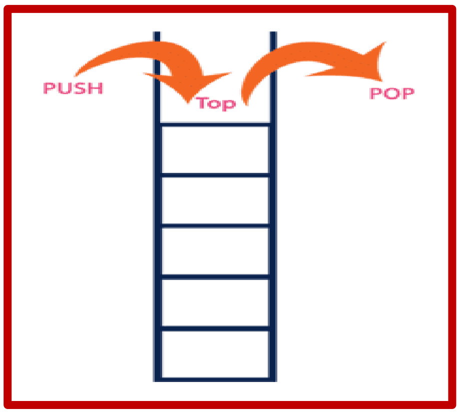
UNIT-II
STACK

Stack Definition:

- Stack is a linear data structure and it is an ordered collection of items.
- All the items of stack are inserted and removed from the same end and that end is known as TOP of the stack.
- Random access of items are not possible in stack.
- Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack.
- Stack follow a particular order in which the operations (insertion/deletion of elements) are performed. The order is LIFO(Last In First Out) or FILO(First In Last Out).which means the item which is inserted first will be removed out at last.
- For Eg: stack of books, stack of dishes,stack of cookies etc.



- Stack Representation:



Stack Operations:

Mainly the following 2 basic operations are performed in the stack:

1. **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
2. **Pop:** Removes an item from the stack. If the stack is empty, then it is said to be an Underflow condition.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

i. **peek()** − get the top data element of the stack, without removing it.

ii. **isFull()** − check if stack is full.

iii. **isEmpty()** − check if stack is empty.

1.Push():

Inserting an element into the stack is called Push Operation. Only one item is inserted at a time and item has to be inserted only at top of the stack.

Example 1 : Insert 4 items 30, 20, 25 and 10 one after the other in the stack with a STACK SIZE 4.

Solution:

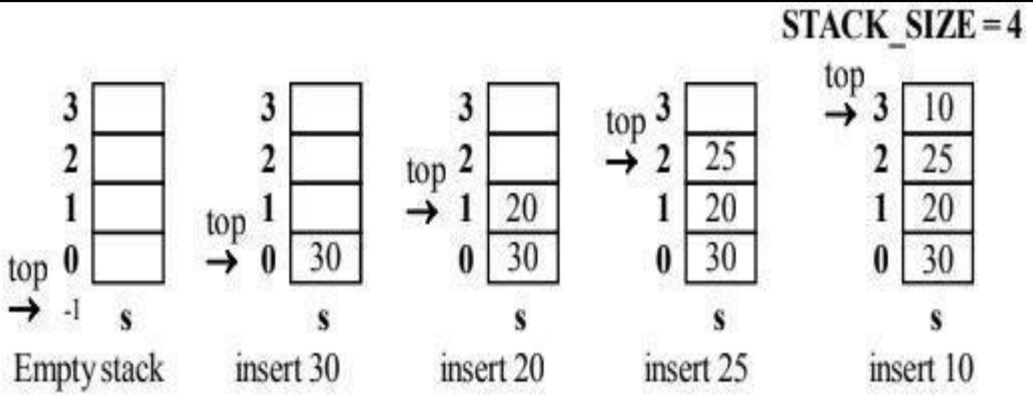


Figure 4 Sequence of insertion operations

When items are being inserted, we may get stack overflow. Now, let us see "What is stack overflow?"

Stack overflow :

- When elements are being inserted, there is a possibility of stack being full.
  Once the stack is full, it is not possible to insert any element. Trying to insert an element, even when the stack is full results in overflow of stack.
- For example, consider the stack shown above with STACK_SIZE 4. We can insert at the most four elements. After inserting 30, 20, 25 and 10 there is no space to insert any item. Then we say that stack is full. This condition is called overflow of stack.

Algorithm for push operation():

```
void push()
 {
  if(top==size-1)
 {
        printf("Could not insert data, Stack is full.\n");
  }
 else
{
    top = top + 1;
    stack[top] = data;
  }
}
```

2. **pop():**
   Deleting an element from the stack is called pop operation. Only one item can be deleted at a time and item has to be deleted only from top of the stack.

   Example : Performing pop operation when stack already contains 30, 20, 25, and 10.

Solution:

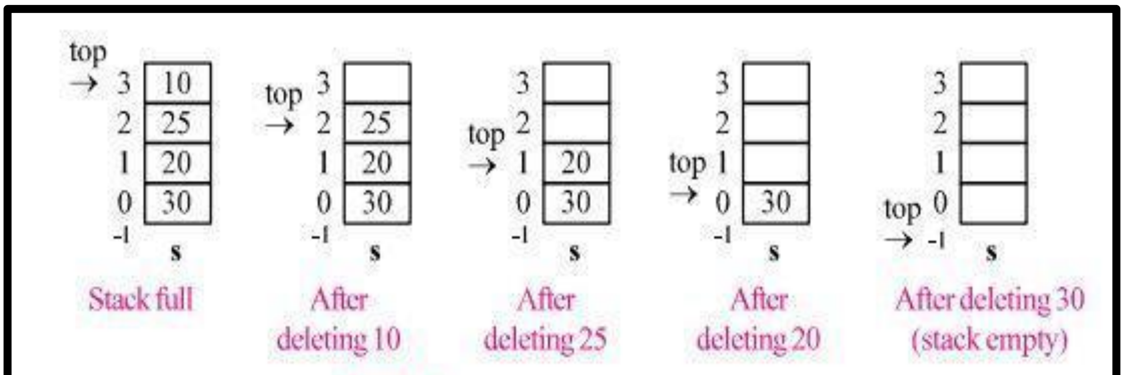


Figure 6 Sequence of POP operations

When items are being deleted, we may get stack underflow. Now, let us see "What is stack under- flow?"

<u>**stack under- flow:**</u>

- When elements are being deleted, there is a possibility of stack being empty. When stack is empty, it is not possible to delete any item. Trying to delete an element from an empty stack results in stack underflow.
- For example, in the above figure 6, after deleting 10, 25, 20 and 30 there are no elements in the stack and stack is empty. Deleting an element from the stack results in stack underflow.

Algorithm for pop operation():

```
void pop()
{
if(top==-1)
 {
printf("Could not retrieve data, Stack is empty.\n");
}
else
{
data = stack[top];
top = top - 1;
}}
```

<u>**Algorithm for isEmpty():**</u>

```
bool isempty()
{
  if(top == -1)
    return true;
  else
    return false;
}
```

<u>**Algorithm for isFull():**</u>

```
bool isempty()
{
  if(top == size-1)
    return true;
  else
    return false;
}
```

Algorithm for peek():

```
int peek()
{
  return stack[top];
}
```

Stack Implementation:

Stack can be implemented in 2 ways:

- Using Array
- Using Linked List

1. **Using Array:** When stack is implemented using array then TOP contains index number and using that index number only element is pushed in and popped out from the correct location.

Program to implement stack using Array:

```
#include<iostream>
using namespace std;
#define size 5
int s[size],top=-1;
void push()
{

if(top==size-1)
{
cout<<"stack full can not insert element\n";
}
else
{
        top++;
cout<<"Enter element to be pushed\n";

cin>>s[top];
cout<<"Element inserted\n";
}
}
//==================================================
void pop()
{
if(top==-1)
{
cout<<"stack empty can not delete\n";
}
else
{
top--;
cout<<"Element removed\n";
}
}
//===================================================
void display()
{
```

```
int i;
if(top==-1)
{
cout<<"stack empty \n";
}
for(i=top;i>=0;i--)
{
cout<<s[i]<<"\n";
}
}
int main()
{
int ch;
while(1)
{
cout<<"\n1. for push()\n2.for pop()\n3.for display()\n4.for exit()\n";
cin>>ch;
if(ch==1)
push();
else if(ch==2)
pop();
else if(ch==3)
display();
else
exit(0);
}
return 0;
}
```

Output:

```
1. for push()
2.for pop()
3.for display()
4.for exit()
1
Enter element to be pushed
12
Element inserted

1. for push()
2.for pop()
3.for display()
4.for exit()
1
Enter element to be pushed
13
Element inserted

1. for push()
2.for pop()
3.for display()
```

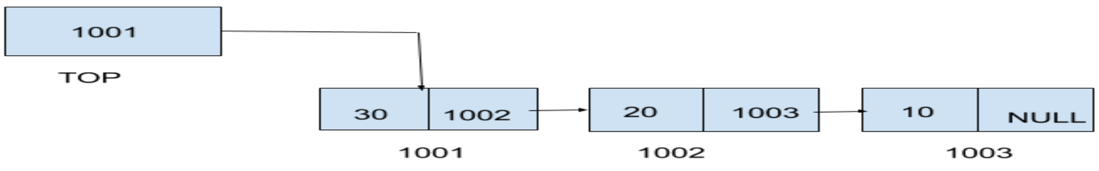
```
4.for exit()
1
Enter element to be pushed
14
Element inserted

1. for push()
2.for pop()
3.for display()
4.for exit()
3
| 14 |
|____|
| 13 |
|____|
| 12 |
|____|

1. for push()
2.for pop()
3.for display()
4.for exit()
4
```

2. **Using Linked List:** When Stack is implemented using linked list then TOP Contains address of head node and using that address push and pop is possible.

Representation of stack using linked:



<u>Adding a node to the stack (Push operation)</u>

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : O(1)

Program to implement stack using Linked List:

```
#include<iostream>
using namespace std;
struct node
{
        int data;
        node *link;
};
node *top=NULL;
void push()
{

        node *n=new node;
        cout<<"Enter data:";
        cin>>n->data;
        n->link=NULL;
        if(top==NULL)
        {
                top=n;
                cout<<"\n data inserted into stack \n";
        }
        else
        {
                n->link=top;
                top=n;
                cout<<"\n data inserted into stack \n ";
        }
}
void pop()
{
        if(top==NULL)
        {
                cout<<"\n stack empty \n";
        }
        else
        {
     node *t;

          t=top;
          top=top->link;
          delete t;
          cout<<"\n node deleted\n";
        }
}
void display()
{
        if(top==NULL)
        {
                cout<<"\n stack empty \n";
        }
        else
        {       node *r;
           r=top;
                while(r!=NULL)
                {
                        cout<<r->data<<"->";
                        r=r->link;
                }
                cout<<"NULL\n";
                }
}
int main()
{
        int ch;
        while(1)
        {
        cout<<"\n1.push\n2.pop\n3.display\n4.Exit\n";
        cin>>ch;
        if(ch==1)
        push();
        else if(ch==2)
        pop();
        else if(ch==3)
        display();
        else
        exit(0);
        }
}
```

Applications of stack:

1. Conversion of expressions
2. Evaluation of expressions
3. String reverse(Palindrome)

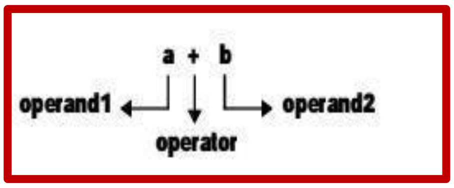
Notation:

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

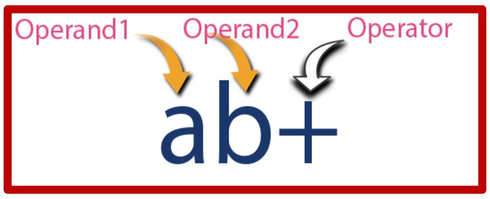
a) **Infix Notation:**

- In an expression, if an operator is in between two operands, the expression is called an infix expression. The infix expression can be parenthesized or un-parenthesized. For example,
- a + b is an un-parenthesized infix expression
- (a + b) is a parenthesized infix expression



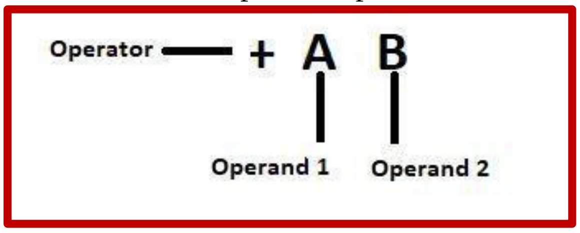
b) **Postfix Notation:**

- In an expression, If an operator comes after the operands, the expression is called postfix expression.
- A B + its an postfix expression
- 100 10 / its an postfix expression



c) **Prefix Notation:**

- In an expression, If an operator comes before the operands, the expression is called a prefix expression.
- + A B its an prefix expression
- - 20 10 it's an prefix expression



Precedence and Associativity of operator:

Precedence of operator:

- The precedence of operators determines which operator is executed first if there is more than one operator in an expression.
- Precedence means priority. Each and every operator has some priority and according to the priority only expressions are executed.

Associativity of Operator:

- Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **to** **R**ight or **R**ight **to** **L**eft.

Operator	Precedence/Priority	Associativity
( ), [ ]	1 (Highest)	L -> R
^(exponent/power)	2	L -> R
*	3	L -> R
/		L -> R
%		L -> R
+	4	L -> R
-		L -> R
=	5	R-> L

Solve few expressions:

Q.1) 4*3+2-5 -> Here,* has highest priority compare to + and - , so * will execute first

Solution:

4 * 3 + 2 -5

= 12 + 2 – 5 ->here + and – both have same priority so here associativity rule will apply

= 14 – 5

= 9

Note: As + and - both have the same priority so precedence rule will not be applicable..in such case associativity rule is used and the expression will evaluate from left to right.

Q.2) (4+2) / 3 * 4 -> ( ) has highest priority ,so ( ) will execute first

Solution:

(4+2) / 3 * 4

= 6 / 3 * 4

= 2 * 4

= 8

Q.3) (10+2) * 4 + 3 + (8 – 4)

Solution:

(10+2) * 4 + 3 + (8 – 4)

= 12 * 4 + 3 + (8 – 4)

= 12 * 4 + 3 + 4

= 48 + 3+ 4

= 55

Q.4) 10 – 2 + 3 * 4 ^ 2

Solution:

10 – 2 + 3 * 4 ^ 2

= 10 – 2 + 3 * (4*4)

= 10 – 2 + 3 * 16

= 10 – 2 + 48

= 8 + 48

= 56

Note: ^ is an exponent/power operator. 4 ^ 2 means $4^{2.}$ To find the 4^2, you need to multiply 4 twice.

Convert infix expression into Postfix expression using stack:

Algorithm:

1. Scan the Infix Expression from left to right and repeat Step 2 to 5 for each element of infix expression until the Stack is empty.
2. If an operand is encountered, add it to the postfix expression.
3. If a left parenthesis is encountered, push it onto Stack.
4. If an operator is encountered ,then:
    - 4.1 Repeatedly pop from Stack and add to postfix expression each operator (on the top of Stack) which has the same precedence as or higher precedence than incoming operator (scanned operator).
    - 4.2 Else-Add operator to Stack.

5 If a right parenthesis is encountered ,then:

5.1 Repeatedly pop from Stack and add to postfix expression each operator (on the top of Stack) until a left parenthesis is encountered.
5.2 Remove the left Parenthesis.

6 END.

Rule:

- If scanned operator (incoming operator) precedence is higher than the operator in the stack then push the incoming operator in the stack.
- If scanned operator (incoming operator) precedence is lower or same than the operator in the stack then pop the operator of the stack and again compare the incoming operator with stack operator.

Q.1) 4 * 3 + 2 - 5 -> infix expression

Solution:

Input/scan	**Stack**	**Postfix expression**	**Description**
4	empty	4	4 is an operand so print it
*	*	4	* is an operator, so compare * with top of stack, but as stack is empty so put incoming operator * into the stack.
3	*	4 3	3 is an operand so print it
+	+	4 3 *	+ is an operator, so compare + with top of stack i.e. * , + is having lower priority than *,so pop * and push the incoming operator + in the stack.
2	+	4 3 * 2	2 is an operand so print it
-	-	4 3 * 2 +	- is an operator,so compare - with top of stack, + and – both are having equal priority so pop top of stack i.e. + and push the incoming operator(-) in stack .
5	-	4 3 * 2 + 5	5 is an operand so print it
	Stack is empty	4 3 * 2 + 5-	As nothing is there for scan,so pop out all the operator from top of the stack one by one.
Postfix Expression : 4 3 * 2 + 5 -			

Q.2) 10 – 2 + 3 * 4 ^ 2

Solution:

Input/scan	**Stack**	**Postfix expression**	**Description**
10		10	10 is operand so print it

-	-	10	- is an operator, so compare - with top of stack, but as stack is empty so put incoming operator - into the stack.
2	-	10 2	2 is operand so print it
+	+	10 2 -	+ is an operator,so compare + with top of stack, + and – both are having equal priority so pop top of stack i.e. - and push the incoming operator(+) in stack .
3	+	10 2 - 3	3 is operand so print it
*	+ *	10 2 - 3	* is an operator, so compare * with top of stack i.e. + , * is having higher priority than +,so push incoming operator(*) in stack.
4	+ *	10 2 – 3 4	4 is operand so print it
^	+ * ^	10 2 – 3 4	^ is an operator, so compare ^ with top of stack i.e. *, ^ is having higher priority than *,so push incoming operator(^) in stack.
2	+ * ^	10 2 – 3 4 2	2 is operand so print it
	Stack is empty	10 2 – 3 4 2 ^ * +	As nothing is there to scan,so pop out all the operator from top of the stack one by one.
Postfix Expression : 10 2 - 3 4 2 ^ * +			

Q.3) (((8 + 1) - (7 - 4)) / (11 - 9))
Solution:

Input/scan	**Stack**	**Postfix expression**
(	(	
(	((	
(	(((	
8	(((	8
+	(((+	8
1	(((+	8 1
)	((	8 1 +
-	((-	8 1 +
(	((-(	8 1 +

7	((-(	8 1 + 7
-	((-(-	8 1 + 7
4	((-(-	8 1 + 7 4
)	((-	8 1 + 7 4- -
)	(	8 1 + 7 4- -
/	(/	8 1 + 7 4- -
(	(/(	8 1 + 7 4 - -
11	(/(	8 1 + 7 4- - 11
-	(/(-	8 1 + 7 4- - 11
9	(/(-	8 1 + 7 4 - 11 9
)	(/	8 1 + 7 4- - 11 9 -
)	Stack is empty	8 1 + 7 4- - 11 9 - /
Postfix Expression : 8 1 + 7 4 - - 11 9 - /		

Note:

- If left parenthesis '(' comes then directly put it in the stack, and if right parenthesis ')' comes then remove all the operator from stack until you get left parenthesis ')'.
- Parenthesis is never mentioned in postfix and prefix expression.
- If stack contains an left parenthesis '(' then incoming operators will never be compared with '(', incoming operator will be directly pushed into stack.

Q.4) K + L- M * N +(O^P) * W/ U/V*T+Q

Solution:

Input/scan	**Stack**	**Postfix expression**
K		K
+	+	K
L	+	K L
-	-	KL+
M	-	KL+ M
*	- *	KL+ M
N	- *	KL+ MN
+	+	KL+ MN*-
(	+ (	KL+ MN*-

O	+ (	KL+ MN*-O
^	+ (^	KL+MN*-O
P	+ (^	KL+ MN*-OP
)	+	KL+ MN*-OP^
*	+ *	KL+ MN*-OP^
W	+ *	KL+ MN*-OP^W
/	+ /	KL+MN*-OP^W*
U	+ /	KL+ MN*-OP^W*U
/	+ /	KL+ MN*-OP^W*U/
V	+ /	KL+ MN*-OP^W*U/V
*	+ *	KL+ MN*-OP^W*U/V/
T	+ *	KL+ MN*-OP^W*U/V/T
+	+	KL+ MN*-OP^W*U/V/T*+
Q	+	KL+ MN*-OP^W*U/V/T*+Q
	STACK EMPTY	KL+ MN*-OP^W*U/V/T*+Q+
Postfix Expression : K L+ M N * - O P ^ W * U/V/T *+Q+		

Convert infix expression into Prefix expression using stack:

<u>Algorithm:</u>

1. Reverse the infix expression. While reversing left parenthesis '(' will become right parenthesis ')' and right parenthesis ')' will become left parenthesis '('.
2. Find out the postfix expression of reversed expression.
3. Reverse the obtained postfix expression.
   Reversed postfix expression is the prefix expression

Q.1) 4 * 3 + 2 - 5 ->infix expression

<u>Solution:</u>

Reverse the above infix expression

5 - 2 + 3 * 4

Find out postfix expression of reversed expression:

Input/scan	**Stack**	**Prefix expression**
5		5
-	-	5
2	-	5 2
+	+	5 2 -

3	+	5 2 - 3
*	+ *	5 2 - 3
4	+ *	5 2 – 3 4
	Stack empty	5 2 – 3 4 * +

- 5 2 – 3 4 * + is the postfix expression now again reverse it to get prefix expression.
- **+ * 4 3 - 2 5** is the prefix expression of 4 * 3 + 2 - 5

Q.2) K + L- M * N +(O^P) * W/ U/V*T+Q
Solution:
Reverse the above infix expression:
Q+T*V/U/W*(P^O)+N*M-L+K

Input/scan	**Stack**	**Prefix expression**
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QT V
/	+/	QTV*
U	+/	QTV*U
/	+/	QTV*U/
W	+/	QTV*U/W
*	+*	QTV*U/W/
(	+* (	QTV*U/W/
P	+* (	QTV*U/W/P
^	+* (^	QTV*U/W/P
O	+* (^	QTV*U/W/PO
)	+*	QTV*U/W/PO^
+	+	QTV*U/W/PO^*+
N	+	QTV*U/W/PO^*+N
*	+*	QTV*U/W/PO^*+N
M	+*	QTV*U/W/PO^*+NM
-	-	QTV*U/W/PO^*+NM*+

L	-	QTV*U/W/PO^*+NM*+L
+	+	QTV*U/W/PO^*+NM*+L-
K	+	QTV*U/W/PO^*+NM*+L-K
		QTV*U/W/PO^*+NM*+L-K+

- **Q T V * U / W / P O ^ * + N M * + L – K +** is the postfix expression now again reverse it to get prefix expression.
- **+ K – L + * M N + * ^ O P / W / U * V T Q** is the prefix expression .

Q.3) ((A + (B-C) * D) ^ E + F)
<u>Solution:</u>
Reverse the above infix expression:
(F+E^(D*(C-B)+A))

Input/scan	**Stack**	**Prefix expression**
(	(	
F	(	F
+	(+	F
E	(+	FE
^	(+^	FE
(	(+^(	FE
D	(+^(	FED
*	(+^(*	FED
(	(+^(*(	FED
C	(+^(*(	FEDC
-	(+^(*(-	FEDC
B	(+^(*(-	FEDCB
)	(+^(*	FEDCB-
+	(+^(+	FEDCB-*
A	(+^(+	FEDCB-*A
)	(+^	FEDCB-*A+
)	STACK EMPTY	FEDCB-*A+^+

- **F E D C B - * A + ^ +** is the postfix expression now again reverse it to get prefix expression.
- **+ ^ + A * - B C D E F** is the prefix expression .

Q.4) (((8 + 1) - (7 - 4)) / (11 - 9))
Reverse the above infix expression:
((9-11)/ ((4-7)-(1+8)))

Input/scan	**Stack**	**Prefix expression**
(	(	
(	((	
9	((	9
-	((-	9
11	((-	9 11
)	(	9 11 -
/	(/	
(	(/(	
(	(/((	
4		9 11 - 4
-	(/((-	9 11 – 4
7	(/((-	9 11 – 4 7
)	(/(	9 11 – 4 7 -
-	(/(-	9 11 – 4 7 -
(	(/(- (	
1		9 11 – 4 7 - 1
+	(/(- (+	
8		9 11 – 4 7 – 1 8
)	(/(-	9 11 – 4 7 – 1 8+
)	(/	9 11 – 4 7 – 1 8+ -
)	Stack empty	9 11 – 4 7 – 1 8+ - /
Reverse (prefix Expression):		

- **9 11 – 4 7 – 1 8 + - /** is the postfix expression now again reverse it to get prefix expression.
- **/ - + 8 1 - 7 4 – 11 9** is the prefix expression .

Convert postfix into infix:
Algorithm:
1. Scan the input from left to right.
2. If the scanned is an operand then push it in the stack

3. Else..if the scanned input is operator the pop top 2 operands from stack. Put the operator between these operands and form a string. Encapsulate the resulted string with parenthesis. Push the resulted string back to stack.

Note: Each string will be considered as one operand.

Q.1) 10 2 – 3 4 2 ^ * +

Scanned input	**Stack**	**Description**
10	10	
2	10 2	
-	(10-2)	- is an operator so removed top 2 operands i.e. 10 and 2 and put the operator between them i.e. 10 - 2 and now encapsulate(cover) it with parenthesis i.e (10-2) and put it back in the stack. Now (10-2) is a single operand like A,2,B,10 etc.
3	(10-2) 3	
4	(10-2) 3 4	
2	(10-2) 3 4 2	
^	(10-2) 3 (4^ 2)	^ is an operator so remove top 2 operands i.e. 4 and 2 and put ^ between them and encapsulate with parenthesis i.e. (4^2) and now (4^2) is a single operand.
*	(10-2) (3 *(4^ 2))	* is an operator so remove top 2 operands i.e. 3 and (4^2) and put * between them i.e. 3*4^2 and now encapsulate with parenthesis i.e. (3*(4^2)) and now (3*(4^2)) is a single operand.
+	((10-2)+ (3 *(4^ 2)))	+ is an operator so remove top 2 operands i.e. (10-2) and (3*(4^2)) put + between them i.e (10-2) + (3*(4^2)) and now encapsulate with parenthesis i.e. ((10-2) + (3*(4^2))) and. now ((10-2) + (3*(4^2))) is a single operand.
Infix Expression is: ((10 -2)+ (3 *(4^ 2)))		

Q.2) AB – DE * +

Scanned input	**Stack**	**Description**
A	A	
B	A B	
-	(A-B)	
D	(A-B) D	
E	(A-B) D E	
*	(A-B) (D*E)	
+	((A-B)+ (D*E))	
AB – DE * + = ((A-B)+ (D*E))		

Convert prefix into infix

Algorithm:

1. Scan the input from right to left.
2. If the scanned input is an operand then push it in the stack
3. Else..if the scanned input is operator the pop top 2 operands from stack. Put the operator between these operands and form a string. Encapsulate the resulted string with parenthesis. Push the resulted string back to stack.

Note:

- **Each string will be considered as one operand.**
- **Element at the top of the stack will be considered as 1st operand and previous one will be 2nd operand…This is only in case of prefix.**

Q.1) / - + 8 1 - 7 4 –11 9

Scanned input	**Stack**	**Description**
9	9	
11	9 11	
-	(11-9)	'-' is an operator so pop 2 operands and place – in between them and cover them with parenthesis. So 11 is 1st operand and 9 is 2nd operand. So the string will be (11-9) and (11-9) is a single operand .
4	(11-9) 4	
7	(11-9) 4 7	
-	(11-9) (7-4)	
1	(11-9) (7-4) 1	
8	(11-9) (7-4) 1 8	

+	(11-9) (7-4) (8+1)	
-	(11-9) ((8+1) - (7-4))	
/	(((8+1) - (7-4)) /(11-9))	
Infix Expression is: (((8+1) - (7-4)) /(11-9))		

Q.2) – * AB * ED
Solution:

Scanned input	**Stack**	**Description**
D	D	
E	D E	
*	(E*D)	
B	(E*D) B	
A	(E*D) B A	
*	(E*D) (A * B)	
-	((A * B) - (E*D))	
Infix Expression is: ((A * B) - (E*D))		

<u>Convert postfix expression into Prefix expression using stack:</u>
<u>Solution</u>
step 1: convert postfix to infix
step 2: convert infix to prefix

Q.1) Convert postfix expression 4 3 * 2 + 5 - to prefix
<u>Solution:</u>
step 1: convert postfix to infix

Input/scan	**stack**
4	4
3	3
*	(4*3)
2	(4*3) 2
+	((4*3)+2)

5	((4*3)+2) 5
-	**(((4*3)+2)- 5)**
(((4*3)+2)- 5) is infix expression	

step 2: convert infix to prefix expression
convert infix expression (((4*3)+2)- 5) to prefix now.
note: scan from right to left

input/scan	**stack**	**prefix expression**
(	**(**	
5	**(**	**5**
-	**(-**	
(	**(- (**	
2	**(- (+**	**5 2**
+	**(- (+**	
(	**(- (+ (**	
3	**(- (+ (**	**5 2 3**
*****	**(- (+ (***	
4	**(- (+ (***	**5 2 3 4**
)	**(- (+**	**5 2 3 4***
)	**(-**	**5 2 3 4* +**
)	**stack empty**	**5 2 3 4* + -**
5 2 3 4 * + - is postfix expression.Reverse it to get prefix expression so, **- + * 4 3 2 5** is prefix expression		

4 3 * 2 + 5 - = - + * 4 3 2 5

<u>Evaluation of Postfix Expression:</u>
<u>Algorithm:</u>

1. Read the expression from left to right,
2. If the scanned input is an operand then push the operand in the stack
3. Else..If the scanned input is an operator then pop the two operands from the stack and evaluate (calculate) it.
4. Push back the result of the evaluation in the stack. Repeat it till the end of the expression.

Q.1) 4 3 * 2 + 5 –
Solution

Scanned input	stack	Description
4	**4**	4 is operand so put it in stack
3	**4 3**	3 is operand so put it in stack
*	**12**	'*' is an operator so pop 2 operand from stack and perform the operation. So consider 4 as operand1 and 3 as operand2 now evaluate 4*3=12.so again put 12 in stack.
2	**12 2**	2 is operand so put it in stack
+	**14**	+ is an operator,so pop 2 operand from stack and perform the operation. So consider 12 as operand1 and 2 as operand2 now evaluate 12+2=14.so again put 14 in stack
5	**14 5**	5 is operand so put it in stack
-	**9**	- is an operator,so pop 2 operand from stack and perform the operation. So consider 14 as operand1 and 5 as operand2 now evaluate 14-5=9.so again put 9 in stack
4 3 * 2 + 5 – = 9		

Q.2) 5 6 2 + * 12 4 / -
Solution:

Scanned input	stack	Description
5	5	
6	5 6	
2	5 6 2	
+	5 8	Op1 is 6 and op2 is 2,so 6+2=8
*	40	Op1 is 5 and op2 is 8,so 5*8=40
12	40 12	
4	40 12 4	
/	40 3	Op1=12 and op2=4,so 12/4=3
-	37	Op1=40 and op2=3,so 40-3=37
5 6 2 + * 12 4 / - = 37		

Evaluation of prefix expression:

Algorithm:

1. Read the expression from right to left,
2. If the scanned input is an operand then push the operand in the stack
3. Else..If the scanned input is an operator then pop the two operands from the stack and evaluate (calculate) it.
4. Push back the result of the evaluation in the stack. Repeat it till the end of the expression.

Note:

- Scanning of prefix expression always happens from right to left
- Top of stack is operand 1 and previous one is operand 2

Q.1) + * 4 3 - 2 5

Solution:

Scanned input	stack	Description
5	5	
2	5 2	
-	-3	Op1 is 2 and op2 is 5,so 2-5=-3
3	-3 3	
4	-3 3 4	
*	-3 12	Op1 is 4 and op2 is 3,so 4*3=12
+	9	Op1 is 12 and op2 is -3,so 12+(-3)= 9

Q.2) - * 5 + 6 2 / 12 4

Solution:

Scanned input	stack	Description
4	4	
12	4 12	
/	3	12 / 4=3..put 3 in the stack
2	3 2	
6	3 2 6	
+	3 8	6+2=8..put 8 in the stack
5	3 8 5	
*	3 40	5*8=40..put 40 in the stack
-	37	40-3=37..put 37 in the stack
- * 5 + 6 2 / 12 4 **= 37**		

Q.3) / - + 8 1 - 7 4 –11 9
Solution:

Scanned input	stack	Description
9	9	
11	9 11	
-	2	
4	2 4	
7	2 4 7	
-	2 3	7-4=3..put 3 in the stack
1	2 3 1	
8	2 3 1 8	
+	2 3 9	8+1=9.. put 9 in the stack
-	2 6	9-3=6..put 6 in the stack
/	3	6/2=3…put 3 in the stack
/ - + 8 1 - 7 4 –11 9 =3		

Recursion:

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.
- The recursion continues until some condition is met to prevent it.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example:

```
void data()
 {
   data(); /* function calls itself */
}
int main()
 {
   data();
}
```

Properties:

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

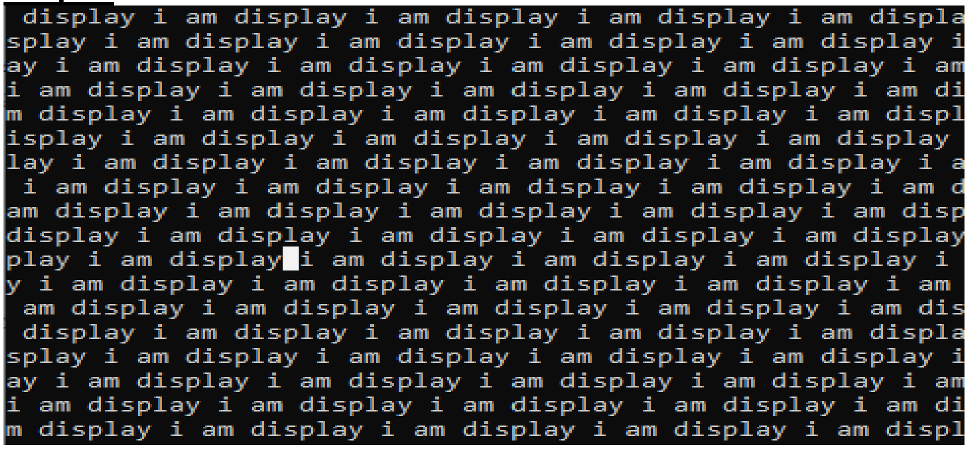
- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

1. Program for recursion without any base criteria (condition):

Program:

```
#include<iostream>
using namespace std;
void display();
int main()
{
        display();
}
void display()
{
        cout<<"i am display ";
        display();
}
```

Output:



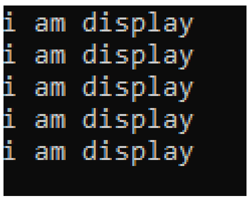
```
 display i am display i am display i am display i am displa
splay i am display i am display i am display i am display i
ay i am display i am display i am display i am display i am
i am display i am display i am display i am display i am di
m display i am display i am display i am display i am displ
isplay i am display i am display i am display i am display
lay i am display i am display i am display i am display i a
 i am display i am display i am display i am display i am d
am display i am display i am display i am display i am disp
display i am display i am display i am display i am display
play i am display i am display i am display i am display i
y i am display i am display i am display i am display i am
 am display i am display i am display i am display i am dis
 display i am display i am display i am display i am displa
splay i am display i am display i am display i am display i
ay i am display i am display i am display i am display i am
i am display i am display i am display i am display i am di
m display i am display i am display i am display i am displ
```

- In the above program we calling display() function inside the body of display() function. This is called recursion.
- But we have not used any condition in above program so in this the display() function will be called infinite times. That’s why base condition is necessary in recursion.

2. Program for recursion with base criteria (condition):

```
#include<iostream>
using namespace std;
void display();
int main()
{
        display();
}
void display()
{
        static int i=1;
        cout<<"i am display\n";
        i++;
        if(i<=5)
        {
                display();
        }
}
```

Output:

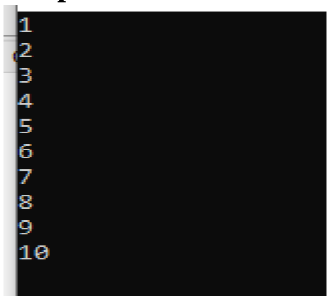

```
i am display
i am display
i am display
i am display
i am display
```

- In the above program display() function is called inside display() function but with a base condition i.e. i<=5. It means as long as the value of i is 5 or less than 5 display() function will be called .
- But as the value of i becomes greater than 5, condition will get false and display() function will no more get called and the program will also not go in infinite loop.

3. Program to print 1 to 10 using recursion.

```
#include<iostream>
using namespace std;
void display();
int main()
{
        display();
}
void display()
{
        static int i=1;
        cout<<i<<endl;
        i++;
        if(i<=10)
        {
                display();
        }
}
```

Output:


```
1
2
3
4
5
6
7
8
9
10
```

4. Program to display factorial of a number using recursion.

Logic:

Suppose we want factorial of 4 using recursion then logic will be

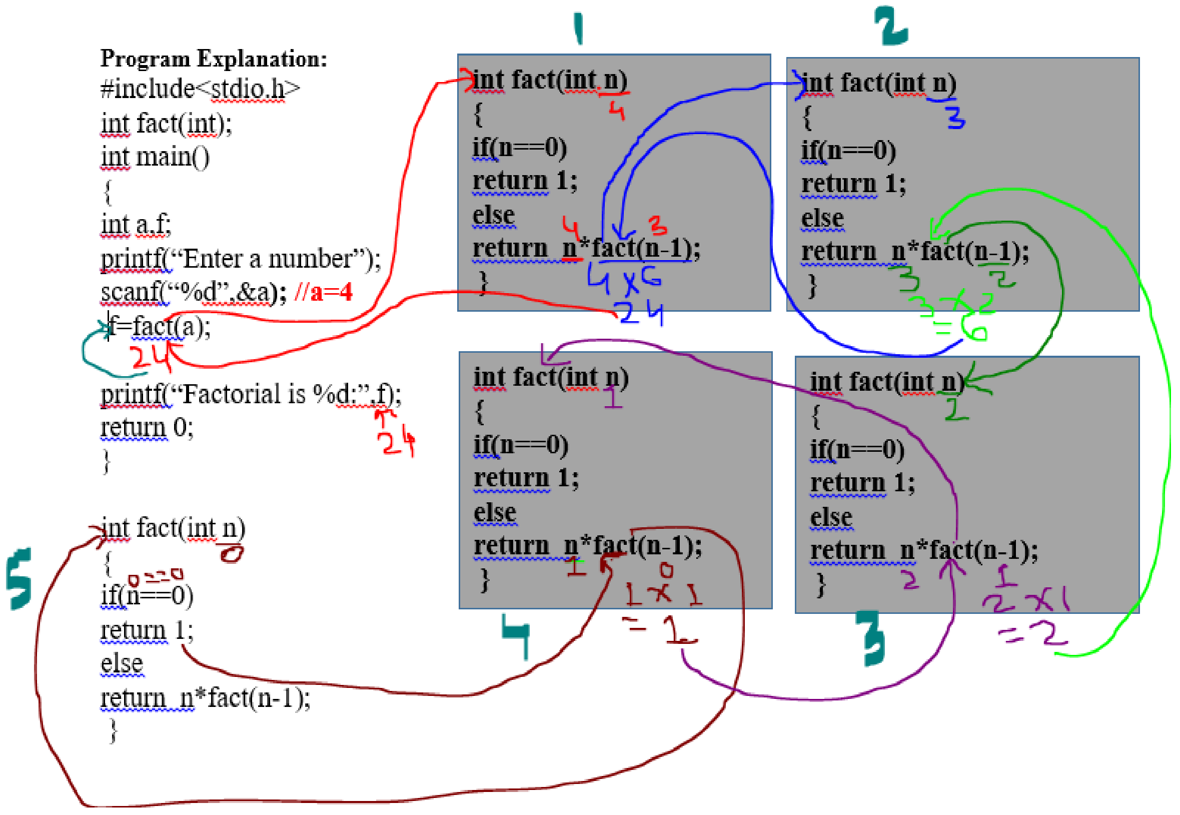
n* fact(n-1)

4*fact(3)
3*fact(2)
2*fact(1)
1*fact(0)

```
#include<iostream>
using namespace  std;
int fact(int);
int main()
{
int a,f;
cout<<"Enter a number:";
cin>>a;
f=fact(a);
cout<<"factorial is "<<f;
return 0;
}
int fact(int n)
{
if(n==0)
return 1;
else
return  n*fact(n-1);
}
```

Output:

```
Enter a number: 4
Factorial is 24
```



Program Explanation:

```
#include<stdio.h>
int fact(int);
int main()
{
int a,f;
printf("Enter a number");
scanf("%d",&a); //a=4
f=fact(a);
printf("Factorial is %d:",f);
return 0;
}

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

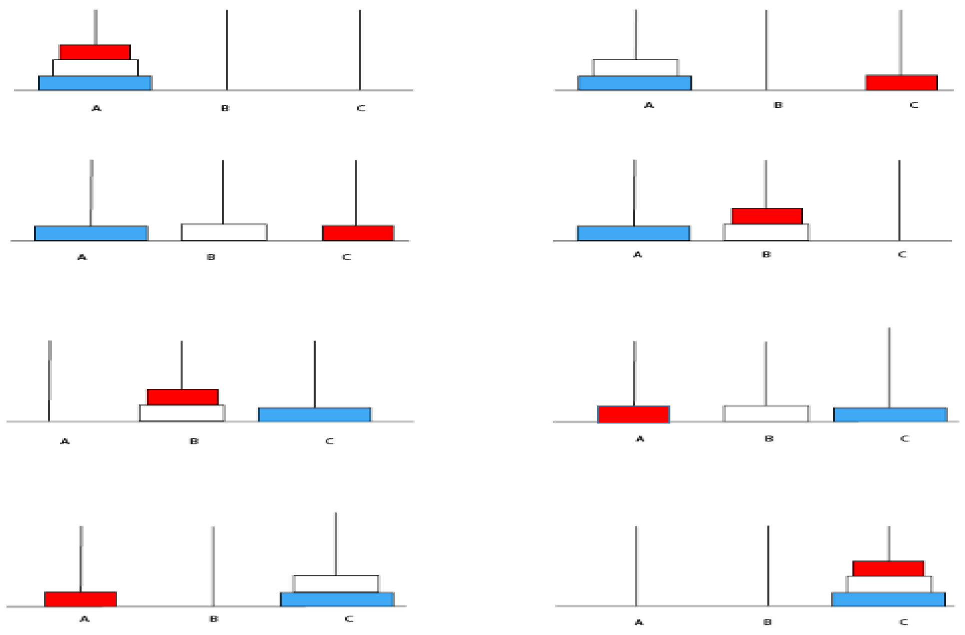
```
int fact(int n)
{
if(n==0)
return 1;
else
return  n*fact(n-1);
}
```

Tower of Hanoi:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

4) Total **2^n-1** moves is required to shift the rod from source to destination where n is number of disks.For example if 3 disks are there then **2^3-1** i.e. **7** moves will be required.

Let us have 3 disks stacked on a peg (rod)



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk

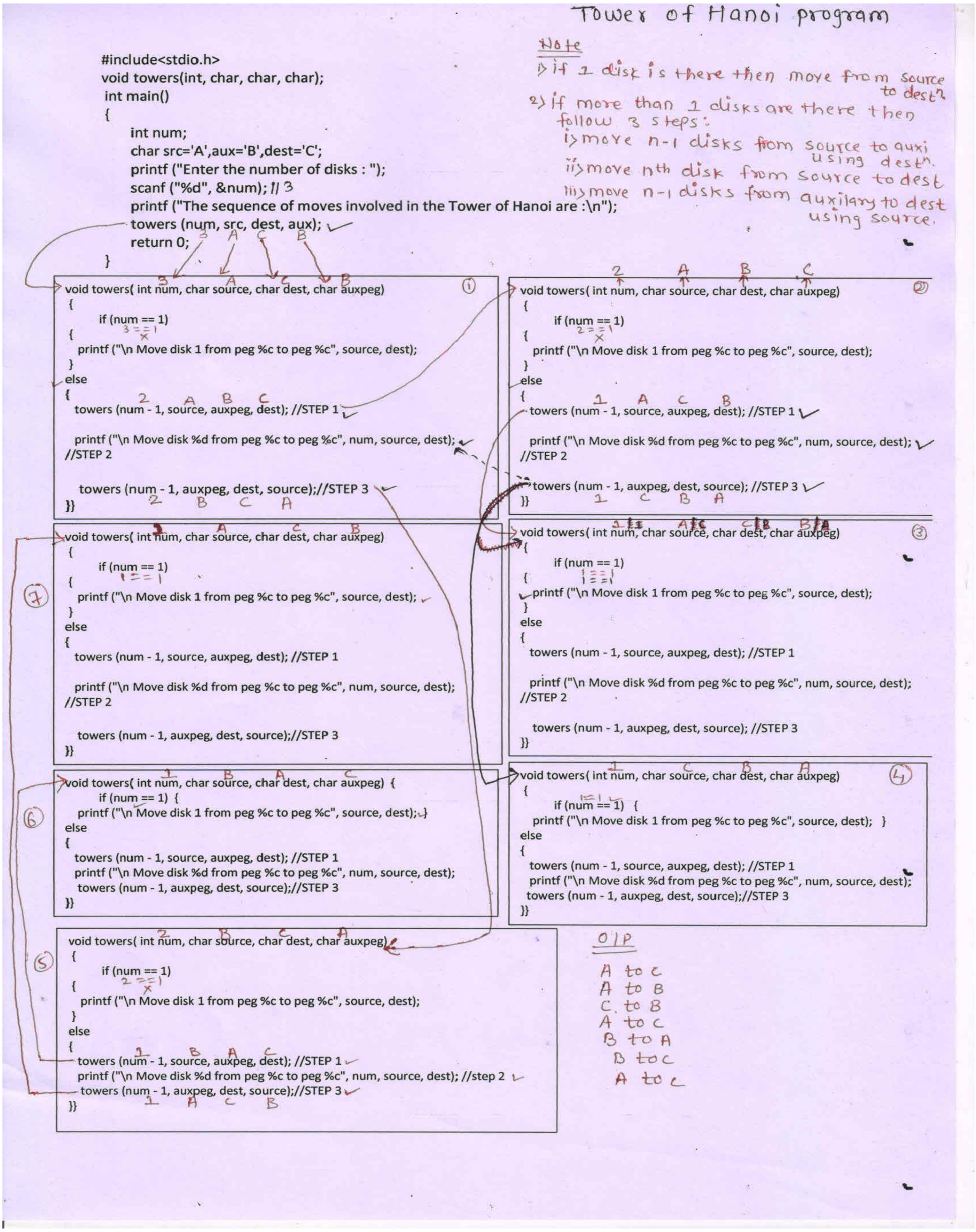
- If there is only 1 disk in the rod then move the disk from source to destination.
- But if there is more than 1 rod then following steps need to be followed:
  a) move n-1 disks from source to auxiliary rod using destination rod
  b) move nth i.e. last disk from source to destination
  c) move n-1 disks from auxiliary rod to destination using source rod

Program of Tower of Hanoi:

```
#include<iostream>
using namespace std;
void towers(int, char, char, char);
 int main()
 {
       int num;
       char src='A', aux='B', dest='C';
        cout<<"Enter the number of disks : ";
        cin>>num;
        cout<<"The sequence of moves involved in the Tower of Hanoi are :\n";
        towers (num, src,dest,aux);
        return 0;
}
 void towers( int num, char source, char dest, char auxpeg)
 {
   if (num == 1)
  {
       cout<<"\n Move disk 1 from peg "<<source<<" to peg "<< dest;
   }
 else
{
   towers (num - 1, source, auxpeg, dest); //STEP 1
    cout<<"\n Move disk "<< num<<" from peg " <<source<<" to peg "<< dest;  //STEP 2
    towers (num - 1, auxpeg, dest, source); //STEP 3
 }
 }
```

Output:

```
Enter the number of disks: 3
The sequence of moves involved in the Tower of Hanoi is
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```



Tower of Hanoi program

Note
1) if 1 disk is there then move from source to destn
2) if more than 1 disks are there then follow 3 steps:
i) move n-1 disks from source to auxi using destn.
ii) move nth disk from source to dest
iii) move n-1 disks from auxilary to dest using source.

```
#include<stdio.h>
void towers(int, char, char, char);
 int main()
 {
     int num;
     char src='A',aux='B',dest='C';
     printf ("Enter the number of disks : ");
     scanf ("%d", &num); // 3
     printf ("The sequence of moves involved in the Tower of Hanoi are :\n");
     towers (num, src, dest, aux); ✓
                3    A    C    B
     return 0;
 }
```

①
```
            3            A            C            B
void towers( int num, char source, char dest, char auxpeg)
 {
     if (num == 1)
         3==1
           x
 {
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);
 }
else
{
              2       A        B        C
   towers (num - 1, source, auxpeg, dest); //STEP 1 ✓

   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest); ✓
//STEP 2

   towers (num - 1, auxpeg, dest, source);//STEP 3 ✓
             2       B      C      A
}}
```

②
```
            2            A            B            C
void towers( int num, char source, char dest, char auxpeg)
 {
     if (num == 1)
         2==1
           x
 {
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);
 }
else
{
              1       A        C        B
   towers (num - 1, source, auxpeg, dest); //STEP 1 ✓

   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest); ✓
//STEP 2

   towers (num - 1, auxpeg, dest, source); //STEP 3 ✓
             1       C      B      A
}}
```

⑦
```
            1            A            C            B
void towers( int num, char source, char dest, char auxpeg)
 {
     if (num == 1)
         1==1
 {
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest); ✓
 }
else
{
   towers (num - 1, source, auxpeg, dest); //STEP 1

   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest);
//STEP 2

   towers (num - 1, auxpeg, dest, source);//STEP 3
}}
```

③
```
            1            A/C          C/B          B/A
void towers( int num, char source, char dest, char auxpeg)
{
     if (num == 1)
         1==1
 {       1==1
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);
 }
else
{
   towers (num - 1, source, auxpeg, dest); //STEP 1

   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest);
//STEP 2

   towers (num - 1, auxpeg, dest, source); //STEP 3
}}
```

⑥
```
            1            B            A            C
void towers( int num, char source, char dest, char auxpeg) {
     if (num == 1) {
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);}
else
{
   towers (num - 1, source, auxpeg, dest); //STEP 1
   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest);
   towers (num - 1, auxpeg, dest, source);//STEP 3
}}
```

④
```
            1            C            B            A
void towers( int num, char source, char dest, char auxpeg)
 {
     if (num == 1)  {
         1==1
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);  }
else
{
   towers (num - 1, source, auxpeg, dest); //STEP 1
   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest);
  towers (num - 1, auxpeg, dest, source);//STEP 3
}}
```

⑤
```
            2            B            C            A
void towers( int num, char source, char dest, char auxpeg) ✓
 {
     if (num == 1)
         2==1
           x
 {
   printf ("\n Move disk 1 from peg %c to peg %c", source, dest);
 }
else
{
              1       B        A        C
   towers (num - 1, source, auxpeg, dest); //STEP 1 ✓
   printf ("\n Move disk %d from peg %c to peg %c", num, source, dest); //step 2 ✓
   towers (num - 1, auxpeg, dest, source);//STEP 3 ✓
             1       A      C      B
}}
```

O/P

```
A to C
A to B
C to B
A to C
B to A
B to C
A to C
```

****