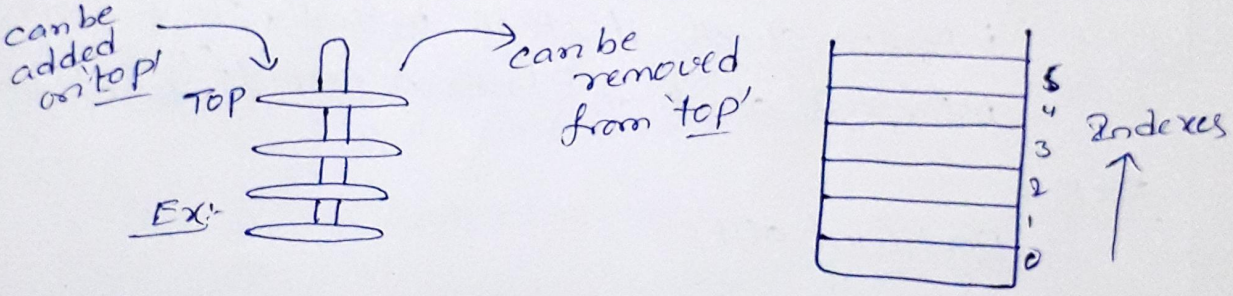
STACKS

Introduction to stack:

→ Stack is a linear data structure. (Abstract data type)
→ It is a collection of homogeneous data elements
→ Two operations can be done on stack
  (i) Insertion
  (ii) Deletion.

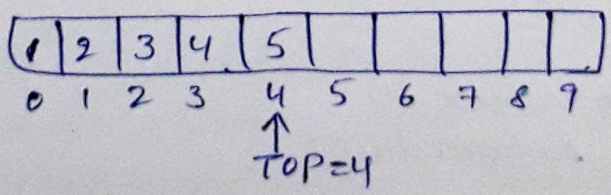
In Stack elements can be added or removed only from one end, which is called the "TOP". Hence, a stack is called LIFO (Last-In-First-out) data structure, as the element that was inserted last is the first one to be taken out.

Examples of stack are:- CD Rack, pile of plates etc.



A stack can be implemented by Array and Linked list. stack can be either a fixed size or can be dynamic size. Array implementation of stack using arrays which makes it a fixed size stack implementation.

Array Implementation of stacks:-



Top → used to store the address of the topmost element
Max → stores the maximum no of elements that the stack can hold.

If Top = -1, stack is empty

Top = Max-1, stack is full.

The operations on stack are:

Push → pushing (storing) an element on the stack

Pop → Removing (deleting) an element from the stack.

To do push and POP we need to check the status of stack. Those are

Peek → get the top data element of the stack, without removing it.

isFull → Check if stack is full (then stack is overflow)

isEmpty → check if stack is empty (then stack is underflow)

Peek :-

```
peek(int a[])
{
  if (top==-1)
    {
      printf("\n stack is Empty");
      return -1;
    }
  else
      return (a[top]);
}
```

isFull :-

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

isEmpty:-

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

PUSH operation:-

The process of putting a new element onto stack is known as PUSH. Steps to be followed are -

1. Check if stack is full.
2. If stack is full, print overflow and exit
3. If stack is not full, increment top to point next empty space.
4. Add data element to the stack location, where top is pointing.

Above Algorithm in form of function is:-

```
int push (int a[], int val)
{
    if (top==max-1)
    {
        printf("\n stack is overflow");
    }
    else
    {
        top++;
        a[top]= val;
    }
```

POP operation:-

Accessing the element while removing it from stack, is known as pop operation. Here, top will be decremented to a[] lower position to point to next value. Steps are

1. Check if stack is Empty
2. If stack is Empty, display underflow and exit
3. If stack is not empty, access the data element at which top is pointing

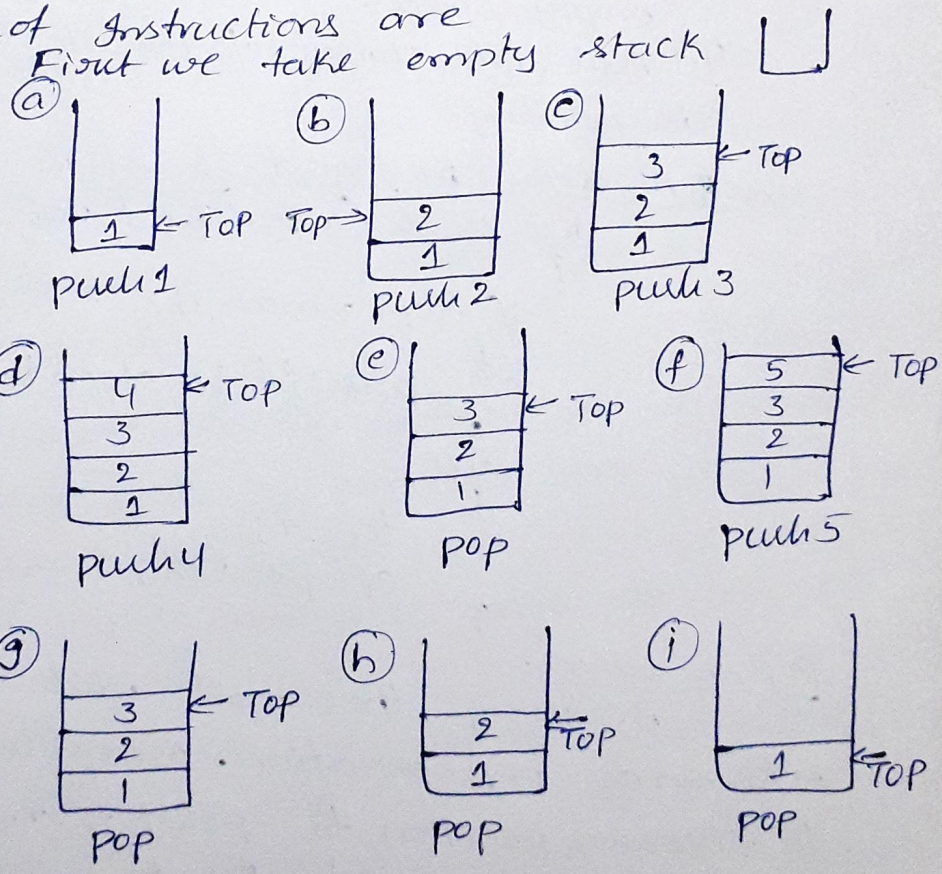
4. Decrease the value of top by 1.
5. return value.

Function :-

```
int pop (int a[])
{
    if (top==-1)
    {
        printf("\n stack underflow");
        return -1;
    }
    else
    {
        val=a[top];
        top--;
        return val;
    }
}
```

Ex :- sequence of instructions are
First we take empty stack

ⓐ Push 1
ⓑ push 2
ⓒ push 3
ⓓ push 4
ⓔ pop
ⓕ push 5
ⓖ pop
ⓗ pop
ⓘ pop



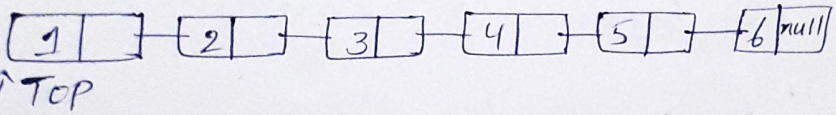
Linked list implementation of stack:-

The process of creating stack using array is easy, but the drawback is that the array must be declared to have some fixed size.

If the (array) stack size cannot be determined in advance, then the other alternative is linked list representation.

In linked stack every node has two parts - one stores data and another stores the address of the next node. The start pointer of linked list is used as TOP.

Here, if TOP = null, stack is empty



Operations on linked stack are:-

① Push:- The new element is added at the topmost position of the stack.

Steps are:-

1. Allocate memory for new node as Newnode
2. save data in Newnode data position
3. Check, top is null.
4. If top is null, set Newnode address position is null.
5. If top is not null, set Newnode address position is TOP
6. Then set Newnode as TOP.

Function for linked stack push operation is

```
struct stack
push (struct stack *top, int val)
{
    struct stack *ptr;
```

```
ptr=(struct stack*)malloc(sizeof(struct stack));
ptr->data=val;
if(top==null)
  {
     ptr->next=null;
     top=ptr;
  }
else
  {
     ptr->next=top;
     top=ptr;
  }
}
```

② **POP:-** Pop operation is used to delete topmost element from a stack. Before deleting the value, we must check top = null, because if top is null stack is empty.

steps are :-

1. Check top is null
2. print underflow and exit
3. set ptr that points to the top
4. Set Top to the next node in sequence.
5. The data occupied in ptr will be deleted.

Function:-

```
Struct stack *pop(struct stack *top)
{
   struct stack *ptr;
   ptr=top;
   if(top==null)
         printf("\n stack underflow");
   else
   {
      top=top->next;
      printf("\n The value being deleted %d", ptr->
                                                data);
      free(ptr);
   }
   return top;
}
```

APPLICATIONS OF STACKS:-

1. Reversing a list
2. Parenthesis checker
3. Conversion of infix expression into a postfix expression
4. Evaluation of postfix expression
5. Conversion of infix expression into a prefix expression
6. Evaluation of prefix expression.

(1) Reversing a list:-

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. once all numbers have been read, the numbers can be popped one at a time the stored in the array starting from the first index.

code:

```
#include <stdio.h>
#include<conio.h>
int stack[10];
int top=-1;
int pop();
void push(int);
int main()
{
    int val,n,i;
    arr[10];
    clrscr();
    printf("enter no of elements:");
    scanf("%d",&n);
    printf("enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    for(i=0;i<n;i++)
        push(arr[i]);
    for(i=0;i<n;i++)
    {
        val=pop();
        arr[i]=val;
    }
    printf("\n The reversed
            array is:");
    for(i=0;i<n;i++)
    {
        printf("%d",arr[i]);
    }
    getch();
    return 0;
}
void push(int val)
{
    stack[++top]=val;
}
int pop()
{
    return(stack[top--]);
}
```

(2) Parentheses checker:-

Stacks can be used to check the validity of parentheses in any arithmatic / algebraic expression.

Valid → Every open bracket there is a corresponding closing bracket Ex:- {A+(B-C)}

Invalid → Any open bracket doesn't contain relative closing bracket Ex:- {A+[B-C)}

Code:-

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int stack[10];
int top=-1;
void push(char);
char pop();
void main()
{
  char exp[10], temp;
  int i, flag=1;
  clrscr();
  printf("Enter the expression");
  gets(exp);
  for(i=0; i<strlen(exp); i++)
  {
    if(exp[i]=='('||exp[i]=='{'||exp[i]=='[')
    push(exp[i]);
    if(exp[i]==')'||exp[i]=='}'||exp[i]==']')
      if(top==-1)
      flag=0;
      else
      {
        temp=pop();
        if(exp[i]==')'&&(temp=='{'||temp=='['))
```

```
            flag=0;
        if (exp[i]=='}' && (temp=='(' || temp=='['))
            flag=0;
        if (exp[i]==']' && (temp=='{' || temp=='('))
            flag=0;
    }  // else loop
} // for loop
if (top>=0)
    flag=0;
if (flag==1)
    printf("Valid Expression");
else
    printf("Invalid Expression");
} // main
void push(char c)
{
    if (top=(Max-1))
        printf("stack overflow");
    else
    {
        top=top+1;
        stack[top]=c;
    }
}
char pop()
{
    if (top==-1)
        printf("stack underflow");
    else
        return(stack[top--]);
}
```

① (A+[B*C)) → It results Invalid Expression.

② (A+[B*C]) → It results valid Expression

③ Conversion of an Infix Expression into a postfix expression.

Before this, we have to know what is an expression, infix, postfix and prefix expressions.

Expression :- An Expression is a combination of operators and the operands having different types of notations. They are

(1) Infix Expression :-

In Infix notation an operator is placed in between operands. Ex :- a*b

(2) Postfix Expression :-

In postfix notation an operator is placed after the operands. Ex :- ab*

(3) Prefix Expression :-

In Prefix notation an operator is placed before the operands Ex :- *ab

Ex :- Infix	Prefix	Postfix
(A-B)*(C+D)	*-AB+CD	AB-CD+*
A+B*C	+A*BC	ABC*+
A+(B*C)	~~ABC*+~~ +A*BC	ABC*+
(A+B)*C	*+ABC	AB+C*

we have to follow the operator precedence before converting the expressions. Precedence order is

① Brackets ( ), ^ ② *, / ③ +, - ④ >, <, !=, <=, >=, = ⑤ && ⑥ ||.

Whenever we convert Infix to postfix expression, we have to follow some rules.

→ Scan Expression left to right
→ Print operands as they arrive
→ If operator arrive & stack is empty, push that operator on to the stack.
→ If incoming operator has higher precedence than the TOP of the stack, push it on stack.
→ If incoming operator has lower precedence than the TOP of the stack, then POP and print TOP. Then test the incoming operator with the new TOP of stack.
→ If incoming operator has equal precedence with TOP, use Associativity Rules.
→ For, Associativity of Left to Right, POP and print TOP, then Push the incoming operator.
→ For, Associativity of Right to Left, Push incoming operator on stack.
→ At the end of Expression, POP & print all operators from the stack.
→ If the incoming symbol is '(' push it on stack
→ If the incoming symbol is ')' pop the stack and print operators till '(' is found.
→ If stack 'TOP' is '(', push operator on stack.

By following all these rules, we convert the following two expressions.

① A+B*C-D/E
② (A+B*(C-D))/E

① A + B * C - D / E

→

Character	Stack	Postfix Expression
A	-	A
+	+	A
B	+	AB
*	+ *	AB
C	+ *	ABC
-	-	ABC*+
D	-	ABC*+D
/	-/	ABC*+D
E	-/	ABC*+DE
-	-	ABC*+DE/-

② (A+B*(C-D))/E

Character	Stack	postfix Expression
(	(	-
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
(	(+*(	AB
C	(+*(	ABC
-	(+*(-	ABC
D	(+*(-	ABCD
)	(+*	ABCD-
)	-	ABCD-*+
/	/	ABCD-*+
E	/	ABCD-*+E

O/P:- ABCD-*+E/-

Algorithm for conversion of Infix to postfix Notation:

1. Start
2. Start reading the infix expression from left to right
3. Repeat step 4 to 7 for each element until the stack is Empty
4. If we scan a operand we output it, print it
5. Else,

   if the scanned operator is greater in precedence than the operator in the stack (or) if the stack is empty (or) the stack contains a '(', push it.

   else

   Pop all the operators having greater or equal precedence that that of scanned operator. After doing that push the scanned operator to the stack. In case there is a parenthesis while popping then stop and push the scanned operator in the stack.
6. If a '(' is encountered, push it onto stack.
7. If a ')' is encountered, repeatedly pop from stack and output it until a '(' is encountered.
8. The output is printed in postfix notation
9. Stop.

(4) Evaluation of postfix expression:-

In evaluation of postfix expression, every character in the postfix expression is scanned from left to right. If an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Algorithm :-

1. Start
2. Start reading expression from left to right
3. Repeat 4,5 until stack is empty
4. If an operand is encountered, push it on to stack.

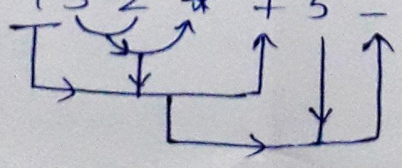
   If an operator is encountered, then
   - Pop the top two elements from the stack as A & B
   - Evaluate B operator A, where A is the topmost element and B is the element below A.
   - Push the result of evaluation on the stack.

   [end of if]
5. Set the result equal to the topmost element of the stack
6. Exit.

Ex:- 2 3 1 * + 9 - = -4

3 6 2 + * 12 4 / - = 37

4 3 2 * + 5 - = 5



⇒ Result.

Postfix expression :- 4 3 2 * + 5 -

operand	(Stack) operation	Stack (result)
4	push	4
3	push	3
2	push	2
*	pop(2,3) and 3*2=6 then push 6	4,6
+	pop(6,4) and 4+6=10 then push 10.	10.
5	push	10,5
-	pop(5,10) and 10-5=5 then push 5	5

⑤ Conversion of Infix to prefix expression:-

Whenever we convert Infix to prefix expression, we have to follow some rules.

→ Reverse infix expression & swap '(' to ')' & ')' to '('.
→ Scan expression from left to right.
→ If operand, print as they arrive
→ If operator, if stack is empty push
→ If operator coming higher precedence than the top, push
→ If operator of equal precedence with top && operator is '^', pop and print top of stack. Then test with new TOP.
→ If operator of equal precedence with top, push on stack
→ If operator have lower precedence than the top, then pop and print the TOP of stack. Then test with new TOP.

Infix
↓
Reverse
↓
Infix to postfix (some changes)
↓
output
↓
Reverse
↓
prefix exp.

→ At the end of expression, pop & print all operators from stack
→ If symbol is '(' push
→ If symbol is ')', pop the stack & print until '(' is found.
→ At the end reverse o/p string again.

Ex: ① Infix → a+b*c-d

Reverse → d-c*b+a

Character	Stack	output
d	Empty	d
-	-	d
c	-	dc
*	-*	dc
b	-*	dcb
+	-+	dcb*
a	-+	dcb*a
-		dcb*a+-
	Reverse	-+a*bcd

② Infix → (A*B)-(C/D)+E

Reverse → E+(D/C)-(B*A)

Character	stack	output
E	Empty	E
+	+	E
(	+(	E
D	+(	ED
/	+(/	ED
C	+(/	EDC
)	+	EDC/
-	+-	EDC/
(	+-(	EDC/
B	+-(	EDC/B
*	+-(*	EDC/B
A	+-(*	EDC/BA
)	+-	EDC/BA*
-	Empty	EDC/BA*-+

→ = +--*AB/CDE

③ (A+B)/C *D-E ← Infix

E-D*C/(B+A) ← Reverse

Character	stack	output
E	Empty	E
-	-	E
D	-	ED
*	-*	ED
C	-*	EDC
/	-*/	EDC
(	-*/(	EDC
B	-*/(	EDCB
+	-*/(+	EDCB
A	-*/(+	EDCBA
)	-*/(+	EDCBA
	Empty	EDCBA+/*-

⇓ Reverse

-*/+ABCDE

⑥ Evaluation of prefix expression :-

In evaluation of prefix expression, every character in prefix expression is scanned from Right to left (or Reverse the expression read from left to right). If an operand is occured, then push into stack. If operator occured, then the top two values are popped from the stack and the operator is applied on those values. The result is pushed on to the stack.

Algorithm :-

1. Start
2. Start reading expression from Right to left.
3. Repeat steps 4, 5 until stack is empty.
4. If an operand occurs, push it on to stack.

If an operator is encountered, then

- Pop the top two elements from the stack as A & B
- Evaluate B operator A. Where A is the topmost element and B is the element below A.
- Push the result of evaluation of the stack.

[end of if]

5. Set the result equal to the topmost element of the stack
6. Exit

Right to left ←

Ex①: + − 2 7 * 8 / 4 12

Scanned element	Stack (operation)	Stack (result)
12	Push	12
4	Push	4
/	Pop(4,12) and 12/4 = 3 push 3	3
8	push	3, 8
*	Pop(8,3) and 8*3 = 24 push 24	24
7	push	7
2	push	2
−	Pop(2,7) and (7−2)=5	24, 5
+	push 5	
	Pop(5,24) and 24+5=29 push 29	29

Ex 2:- − + 4 * 3 2 5

element ←	operation	result
5	push	5
2	push	2
3	push	3
*	Pop(3,2) and 2*3=6 push	5, 6
4	push	5, 6, 4
+	Pop(4,6) and 6+4=10 push	5, 10
−	Pop(10,5) and 10−5=5 push	5