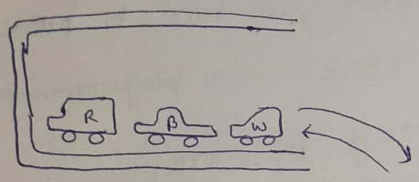
STACK

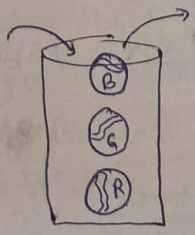
It works on LIFO :- Last-in First out. Actually Stack is a Collection of Elements and those Elements are inserted/deleted on this discipline.

Eg :-



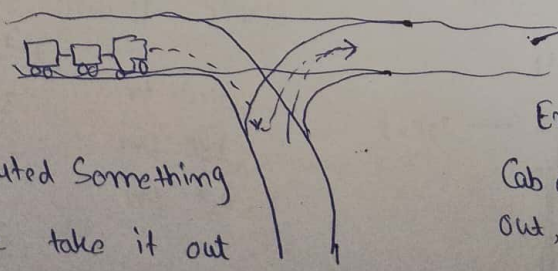
White Car insertated last and white car has to come out first, this is Last in First out.

Eg :-



→ First ball :- Blue → it will insert at last but will come first.

Eg :-



Engine Entered first and last Cab entered last and if Comes out, than Cab Comes out first.

Note :- if we inserted Something in a stack and we take it out again than it will reverse its direction by using Stack.

★ Now, Recursions, we know Recursions using Stack, they are working as a loop but they uses stack. and we can convert recursive function into iterative but when we convert it into iterative version, some times we have to use stack.

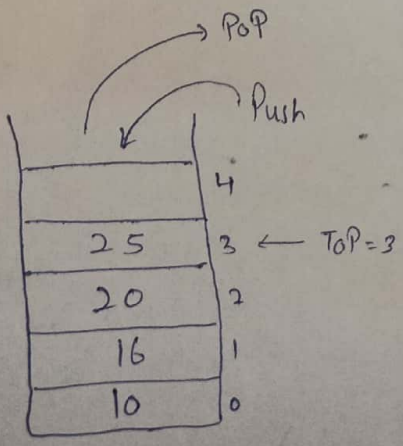
★ Recursions uses stack, we already know - ... but that is a System stack, programmer doesn't have to do anything but when we are converting it into iterative version, we have to provide a stack which is known as programmer stack. So programmer should create a stack and implement it in a program.

★ ADT STACK :- ABSTRACT DATA TYPE STACK

Data :- 1) Space for storing Elements.
2) Top Pointer.

Operations :-

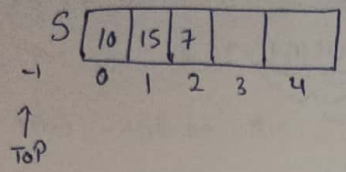
1) Push (x)
2) Pop ()
3) Peek (index)
4) Stack Top ()
5) is Empty ()
6) is Full ()

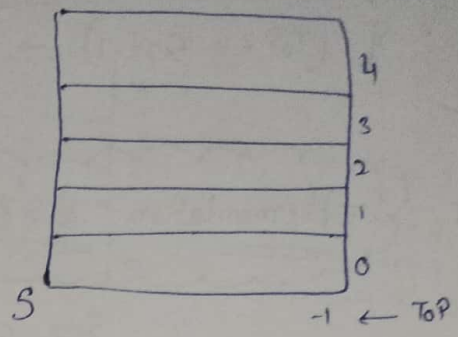


We can store values in Array & Linked List, So we use them to implement a stack.

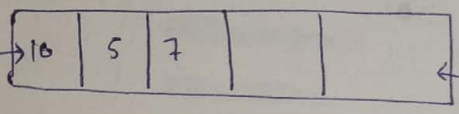
STACK USING ARRAY :-

Size = 5





* Things Required for Implementing a Stack :-
  - An Array
  - Variable for Size
  - TOP Pointer

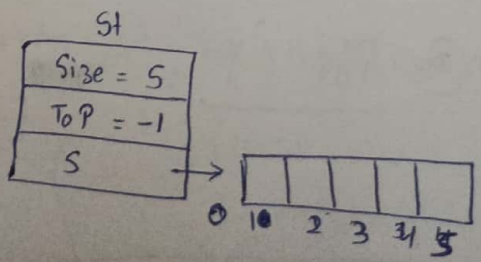


We can insert from this side also, but we will not insert from this side because if we insert from this side, we have to shift all the elements of an array which requires O(n) Time Complexity.

We will insert from this side because by inserting from this side, it requires O(1) Time Complexity and No shifting of Elements required.

- **Structure of Stack** :-

```
Struct Stack
{
    int Size;
    int TOP;
    int *S;
};
```



- **Initializing** :-

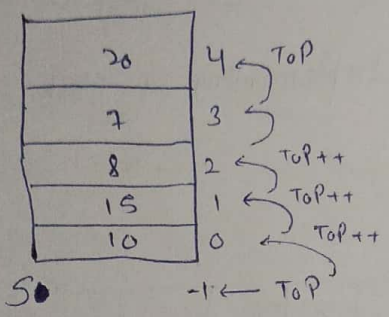
```
int main()
{
    Struct Stack St;
    printf("Enter Size of Stack:\n");
    scanf("%d", &St.Size);
    St.S = new int[St.Size]; // in C++
    St.TOP = -1; // Pointing outside the Stack if Stack is empty.
```

- if (ToP == -1) → Stack Empty Condition
- if (ToP == Size-1) → Stack Full Condition

★ Implemantation of Stack Using Array :-

① Push ():-

Size = 5

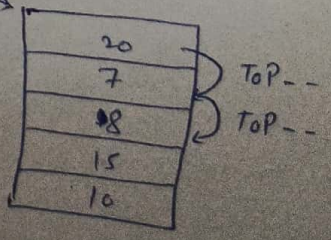


Time Complexity :- O(1)

```
Void Push (Stack *St, int x)
  {
    if (St → ToP == St → Size-1) // is stack full or not
        printf ("Stack overflow");
    else
      {
        St → ToP++;
        St → S [St → ToP] = x;
      }
  }
```

② Pop () :-

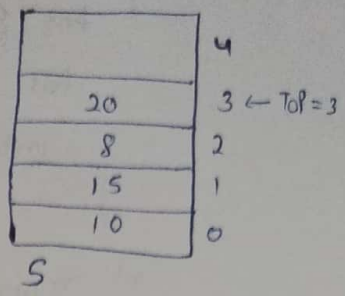
```
int Pop (stack * St, int x)
  {
    int x=-1;
    if (St → ToP == -1)
        printf ("Stack underflow");
    else
      {
        x = St → S [St → ToP];
        St → ToP--;
      }
    return x;
  }
```



Time Complexity :- O(1)

Peek() → knowing a value at Given position.

it means if we want to get the 1st position, than it will be a top, we don't have to see the index, Just see Top position, that will be the 1st position another so on.



3 - 1 + 1 = 3
3 - 2 + 1 = 2
3 - 3 + 1 = 1
3 - 4 + 1 = 0

Pos	Index = Top - Pos + 1
1	3 = 3 - 1 + 1
2	2 = 3 - 2 + 1
3	1 = 3 - 3 + 1
4	0 = 3 - 4 + 1

Time Complexity :- O(1)

```
int Peek (stack St , int Pos)
{
    int x = -1
    if ( Top - Pos + 1 < 0)
        printf (" Invalid Position");
    else
        x = St. S [St. (Top - Pos + 1)];
    Return x;
}
```

④
```
int Stack Top ( stack st)
{ if ( St. Top == -1)
      Return -1;
  else
  Return St. S [St. Top];
}
```

⑤
```
int is Empty (stack st)
{ if (St. Top == -1)
      Return 1;
  else
      Return 0;
}
```

⑥
```
int is Full (stack st)
{ if (St. Top == St. Size-1)
      Return 1;
  else
      Return 0;
}
```

Code :-

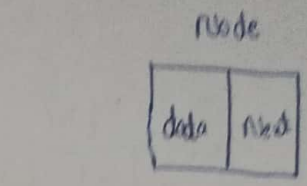
```
Struct Stack
{ int size;
  int top;
  int *s;
};
int main()
{
  struct Stack st;
  void create (struct Stack * st);
  void Display ( struct Stack st);
  void Push ( struct Stack *st, int x);
  int PoP ( struct Stack *st);
  int Peek ( struct Stack st, int index);
  Create (&st);
  Push (&st, 10);
  Push (&st, 20);
  Push (&st, 30);
  Push (&st, 40);
  printf ("%d", Peek (st, 3));
  printf ("\n");
  Display (st);
  printf ("After Popping : %d", PoP (&st));
}
Void create (struct Stack * st)
{
  printf (" Enter Stack size : \n");
  scanf ("%d", &st -> size);
  st -> top = -1;
  st -> s = (int *) malloc ( st -> size * sizeof (int));
}
```

```
Void Display (struct Stack st)
  {
    int i;
    for (i = st.top ; i>=0 ; i--)
    {
      printf ("%d", st.s[i]);
    }
    printf ("\n");
  }
Void Push (struct stack *st , int x)
  {
    if ( st -> top == st -> size - 1)
          printf ("Stack overflown\n");
      else
        {
          st -> top++;
          st -> s[st -> top] = x;
        }
  }
int PoP (struct Stack *st)
    {
      int x = -1;
      if (st -> top == -1)
            printf ("Stack underflown\n");
        else
        {
          x = st -> s [st -> top --];
        }
        return x;
    }
```

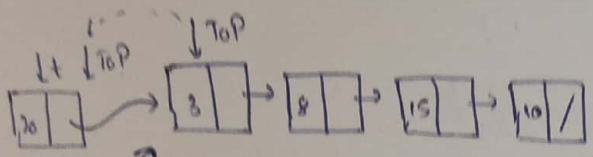
```
int Peek (struct stack st, int index)
{
    int x = -1;
    if (st.top - index + 1 < 0)
        printf ("Invalid index");
    else
    {
        x = st.s [st.top - index + 1];
    }
    return x;
}
int isEmpty (struct stack st)
{
    if (st.top == -1)
        return 1;
    return 0;
}
int isFull (struct stack st)
{
    if (st.top == st.size - 1)
        return 1;
    return 0;
}
int stacktop (struct stack st)
{
    if (! isEmpty (st))
    {
        return st.s [st.top];
    }
    return -1;
}
```

Stack Using Linked list :-

It also follows LIFO.




```
Struct Node
{
    int data;
    Struct Node * next;
}
```



We can insert / delete from this end and it will take time which is O(1).

We can also insert / delete from this end but it will take time which is O(n) bcz for this side, we have to traverse through linked list.

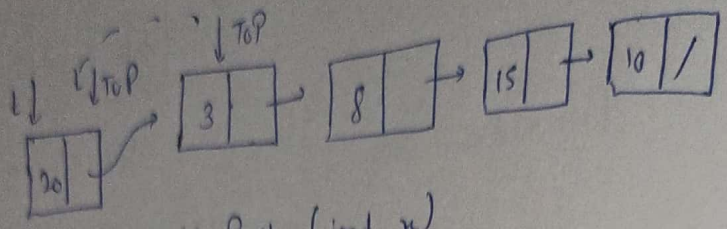
- STACK is Empty

```
if (TOP == NULL)
```

- STACK is FULL

```
Node *t = new Node;
if (t == NULL)
```

☆ Implementation of Stack Using Linked List :-

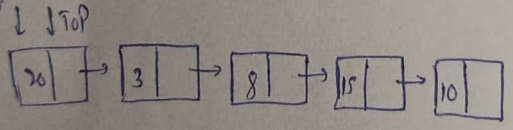


(1)

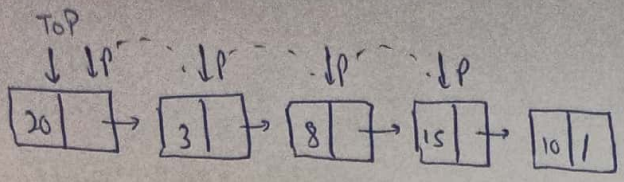
```
Void Push (int x)
{
    Node *t = new Node;
    if (t == NULL)
        printf ("Stack overflow");
    else
    {
        t → data = x;
        t → next = Top;
        Top = t;
    }
}
```

If we insert the very first value, then it will also work.

(2)




```
int PoP ( )
{
    Node *P;
    int x = -1;
    if (TOP == NULL)
        printf ("Stack is Empty");
    else
    {
        P = TOP;
        TOP = TOP → next;
        x = P → data;
        free (P);
    }
    return x;
}
```




```
int Peek (int Pos)
{
    int i;
    Node *P = ToP;
    for (i=0 ; P!=NULL && i< Pos-1 ; i++)
    {
        P = P→ next;
    }
    if (P!=NULL)
        Return P→ data;
    else
        Return -1;
}
```

(4)

```
int Stack ToP ( )
{
    if (ToP)
        Return ToP→ data;
    Return -1;
}
```

(5)

```
int isEmpty ( )
{
    Return ToP ? 0 : 1;
}
```

(6)

```
int isFull ( )
{
    Node *t = new Node;
    int R = t ? 1 : 0;  → if t != NULL, then R=1 else R=0
    free (t);
    Return R;
}
```

Code :-

```
Struct Node
{
   int data;
   struct Node * next;
} * top = NULL;

int main()
{
   void Display();
   void Push (int x);
   int PoP();
   int Peek (int index);
   Push (10);
   Push (20);
   Push (30);
   Display();
   printf ("%d", PoP());
   printf ("%d", Peek (2));
}
Void Push (int x)
{
   struct Node *t;
   t = (struct Node *) malloc (sizeof (struct Node));
   if (t == NULL)
      printf ("Stack is Full \n");
   else
   {
      t -> data = x;
      t -> next = top;
      top = t;
   }
}
```

PoP()

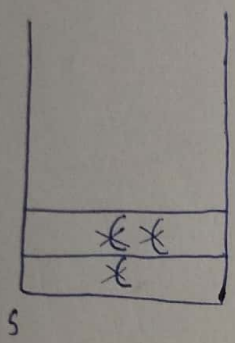
```
struct Node
int x=-1;
if (top
else
```

```
int pop()
{
    struct Node *t;
    int x=-1;
    if (top==NULL)
        printf("Stack is Empty\n");
    else
    {
        t= top;
        top= top->next;
        x= t->data;
        free(t);
    }
    return x;
}
void Display()
{
    struct Node *p;
    p= top;
    while(p!=NULL)
    {
        printf("%d ", p->data);
        p= p->next;
    }
    printf("\n");
}
int Peek(int index)
{
    int i;
    struct Node *p= top;
    for(i=0; p!=NULL && i<index-1; i++)
        p= p->next;
    if(p!=NULL)
        return p->data;
    else
        return -1;
}
```

★ Parenthesis Matching

:- We have check whether the parenthesis are balanced or not, that is for every opening bracket, there must be a closing bracket.

$((a+b) * (c-d))$



→ if stack is Empty, then parenthesis are balanced

$(((a+b) * (c-d))$ → parenthesis are not balanced

Note :- If we are Scanning through an Expression by pushing opening brackets and popping out closing brackets, In this procedure

(i) If at the end the stack is empty, then they are perfectly matching.

(ii) If at the end, there will be an opening bracket remaining then it will not be matching.

(iii) If we have a closing bracket but the stack is Empty, then it is not matching.

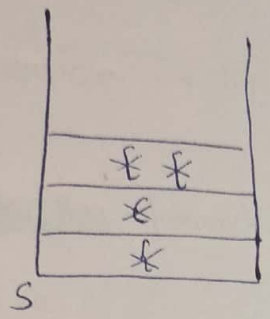
ode :-

```
Struct Node
{ char data;
  Struct Node * next;
}* top = NULL;

int main ()
{
  char * exp = "((a+b) * (c+d))";
   int balanced (char * exp);
   int m = balanced (exp);
   printf ("%d", m);
}
Void Push (char x)
  {
     Struct Node *t;
     t = (struct Node *) malloc (sizeof (struct Node));
      if (t == NULL)
          printf ("Stack is full\n");
      else
       { t → data = x;
         t → next = top;
          top = t;
       }
  }
```

```
char PoP()
{
  struct node *t;
  if (top == NULL)
      printf ("Stack is Empty \n");
  else
    {
      t = top;
      top = top -> next;
      free (t);
    }
}

int balanced (char * exp)
{
    int i;
    for (i=0 ; exp [i] != '\0' ; i++)
      {
        if (exp [i] == '(')
          push (exp [i]);
        else if (exp [i] == ')')
          {
            if (top == NULL)
              return 0;
            PoP();
          }
      }
    if (top == NULL)
      Return 1; // True
    else
      Return 0; // false
}
```

More on Parenthesis Matching :- Different type of brackets

{ ([a+b] * [c-d]) / e }



Code :-

```
Struct Node
{ char data;
  Struct Node * next;
}* top = NULL;

int main()
{
  char * exp = " { ( [a+b] * [c+d] ) / e } ";
  int balanced (char * exp);
  int m = balanced (exp);
  printf ("%d", m);
}
Void Push (char x)
{
  Struct Node *t;
  t = (Struct Node *) malloc (sizeof (Struct Node));
  if (t == NULL)
    printf ("Stack is full \n");
  else
    { t -> data = x;
      t -> next = top;
      top = t;
    }
}
```

```
char pop()
{
  struct node *t;
  char x = -1;
  if (top == NULL);
     printf ("stack is empty\n");
  else
    {
      t = top;
      top = top -> next;
      x = t -> data;
      free (t);
    }
    return x;
}
int Balanced (char * exp)
{
  int i;
  char x;
  for (i=0 ; exp[i] = '\0' ; i++)
   {
     if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
        Push (exp[i]);
     else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}')
      {
        if (top == NULL)
           return 0;
        x = POP();
       if ((exp[i] == ')' && x != '(') || (exp[i] == ']' && x != '[') || (exp[i] == '}' && x != '{'))
           return 0;
      }
     if (top == NULL)
        return 1;
     else
  }      return 0.
```

Scanned with CamScanner

Infix to Postfix Conversion :-

1. What is Postfix ?
2. Why we need Postfix ?
3. Precedance.
4. Manual Conversion. → by Pen & paper

(A) Infix :- [redacted] operand operator operand

eg :- a+b

(B) Prefix :- operator operand operand

eg :- + ab → Add, what? a & b.

(C) Postfix :- operand operand operator

Eg :- ab + → a & b, what ?, add them.

Q Why we need Postfix | Prefix ?

A -

$$8 \underset{6}{+} \; 3 \underset{5}{*} \; \frac{(9-6)}{1} \underset{3}{/} \frac{2^2}{2} \; \underset{7}{+} \; 6 \underset{4}{/} 2$$

We see that the order of performance of operators by jumping here and there by looking into the Expression and first which one should i performed So, to find which one should i performed, we have to scan through this Expression, so to avoid this we need Postfix | Prefix. Now, in this We will not scan this expression multiple times, we will scan only once.

8 3 9 6 - 2 ^ / * + 6 2 / +

① $a + b * c$ ② $a + b + c * d$

Symbol	Precedence	Associativity
+, -	1	
*, /	2	
( )	3	

Now, for Conversion, we need to fully Parenthesis an expression, [redacted]

And Compiler need fully paranthesis function or expression to perform and if the expression is not paranthesis, then Compiler make it a paranthesis expression not physically but logically. by using Precedence.

Note :- Precedance doesnt mean order of execution but it means which operator has higher Precedance, put that one in paranthesis first

① $(a + (b * c))$

Prefix: $(a + [* bc])$ → $+ a * bc$

Postfix: $(a + [bc *])$ → $a\ bc * +$

② $a + b + c * d$

Prefix: $a+b + * cd$ → $[+ ab] + [* cd]$ → $+ +ab * cd$

Postfix: $a+b + cd*$ → $[ab +] + [cd *]$ → $ab + cd * +$

③ $(a+b) * (c-d)$

Prefix: $[+ab] * (c-d)$ → $[+ab] * [-cd]$ → $* +ab - cd$

Postfix: $[ab+] * (c-d)$ → $[ab+] * [cd-]$ → $ab + cd - *$

Note :-

We will always do postfix.

Associativity :-

If Precedences are equal in an expression, then we will go for Associativity.

Symbol	Precedence	Associativity
+, -	1	L-R
*, /	2	L-R
^	3	R-L
-	4	R-L
( )	5	L-R
=		R-L

Two Types :- ① Left - Right
② Right - Left

① **Left - Right :-**

a + b + c - d

(((a+b)+c) - d) → Postfix :- ab + c + d -

② **Right - Left :-**

a = b = c = 5

(a = (b = (c = 5))) → Postfix :- abc5 = = =

③ a ^ b ^ c

(a ^ (b ^ c)) → Postfix :- (a^ (b ^ c)) = a^ [bc^] = abc^^

④ -a

Prefix :- -a Postfix :- a-

⑤ (-(-a))

Prefix :- --a Postfix :- a--

⑥ *P

Prefix :- *P Postfix :- P*

⑦ (*(*P))

Prefix :- **P Postfix :- P**

⑧ n!

Postfix :- !n Prefix :- n

⑨ log x

Prefix :- log x Postfix :- x log

These all are Unary operators have more Precedence than other operators and Right to Left Associativity.

Postfix

- a + b * log n!

↓

- a + b * log a [n!]

↓

- a + b * [n! log]

↓

[a -] + b * [n! log]

↓

[a -] + [b n! log] *

↓

a - b n! log + *

Postfix

- a + b * log n!

↓

- a + b * log [! n]

↓

- a + b * [log [! n]

↓

[- a] + b * [log ! n]

↓

[- a] + [* b log ! n]

↓

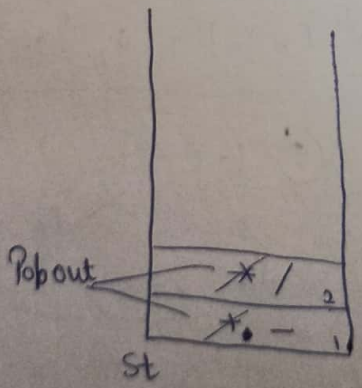
+ - a * b log ! n

★ Infix to Postfix using Stack Method -1 :-

a + b * c - d / e

Symbol	Pre	Associativity
+, -	1	L - R
*, /	2	L - R

"Now, we will Generate a postfix Expression by Scanning above expression and taking one Symbol at a time."



Postfix : a b c * + d e / -

Symbol	Stack	Postfix
a	----------	a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
-	-	abc*+
d	-	abc*+d
/	/, -	abc*+d
e	/, -	abc*+de
----	----	abc*+de/-

Procedure :-

① Send operand to postfix.

② Next is operator, check if there is something in a stack or not, if it is empty then push it. and write its precedance.

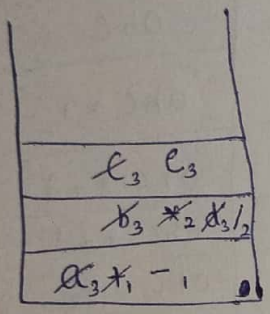
③ For "Next" operator, Look it into the stack, is there any operator or not, if there is any operator, then check its precedance and then check precedance of "Next" operator, if it is greater then push it into the stack. and if it is less then pop out the symbol having higher precedance or equal precedance and push "this" operator into the stack.

★ Infix to postfix using stack Method - 2 :-

we will push operands also into the stack. Every operands will be pushed to stack and operands will from infix have higher precedance than operators.

a+b *c - d/e

Symbol	Precedence	Associativity
+, -	1	L-R
*, /	2	L-R
a, b, c	3	L-R



St

Postfix : a b c * + d e / -

after end of expression, pop out all the things.

★ Program :-

infix

a	+	b	*	c	-	d	/	e	\0
0	1	2	3	4	5	6	7	8	9

```
char * Convert (char * infix)
{
  struct stack St; // Initialized
  char * Postfix = new char [strlen (infix)+1];
  int i, j=0;
  while (infix [i] != '\0')
  { if (isoperand (infix [i])
      Postfix [j++] = infix [i++];
    else
    { if (Pre (infix [i]) > Pre (stacktop (st))
        Push (&st, infix [i++]);
      else
        Postfix [j++] = Pop (&st);
    }
    while (! isempty (st))
    { Postfix [j++] = Pop (&st);
    }
    Postfix [j] = '\0';
```

```
int is operand (char x)
{
  if (x == '+' || x == '-' ||
      x == '*' || x == '/')
    return 0;
  else
    return 1;
}
```

```
int Pre (char x)
{ if (x == '+' || x == '-')
    return 1;
  else if (x == '*' || x == '/')
    return 2;
  else return 0;
}
```

de :-

```
Struct Node
{
    char data;
    Struct Node * next;
} * top = NULL;

int main ()
{
    char * infix = " a+b + c *d";
    char * int to post (char * infix);
    Void Push (char x);
    Push ('#'); ||

    char * Postfix = int to post ( infix);
    printf ( "%s", Postfix);
}
Void Push (char x)
{
    struct Node *t;
    t= (Struct Node *) malloc (size of (struct Node));
    if (t == NULL)
        printf ("stack is full \n");
    else
    {
        t → data = x;
        t → next = top;
        top = t;
    }
}
```

```
char Pop ()
{
    struct Node *t;
    char x = -1;
    if (top == NULL)
        printf ("Stack is Empty \n");
    else
    {
        t = top;
        top = top → next;
        x = t → data;
        free (t);
    }
    Return x;
}
int Pre (char x)
{
    if ( x == '+' || x == '-')
        Return 1;
    else if (x == '*' || x == '/')
        Return 2;
    else
        Return 0;
}
int isoperand (char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/')
        Return 0;
    else
        Return 1;
}
```

```
char * int to post (char * infix)
{
    int i=0, j=0;
    char * Postfix;
    int len = strlen (infix);
    Postfix = (char *) malloc ((len+2) * sizeof (char));
    while (infix [i] != '\0')
    {
        if (isoperand (infix [i])
            Postfix [j++] = infix [i++];
        else
        {
            if (Pre (infix [i]) > Pre (top → data)
                Push (infix [i++]);
            else
                Postfix [j++] = POP();
        }
    }
    while (top != NULL)
        Postfix [j++] = POP();
    Postfix [j] = '\0';
    return Postfix;
}
```

★ Student - Challenge :- Infix to Postfix Conversion having ^ and other operators.

$((a+b)*c)-d\wedge e\wedge f$

$([ab+]*c)-d\wedge e\wedge f$

$[ab+c*]-d\wedge e\wedge f$

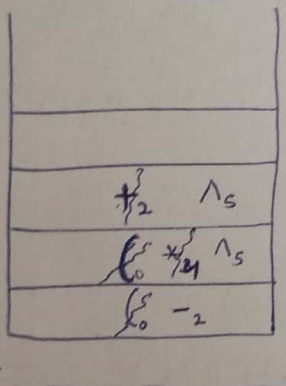
$[ab+c*]-d\wedge[ef\wedge]$

$[ab+c*]-[def\wedge\wedge]$

$ab+c*def\wedge\wedge-$

Symbol	Out stack Precedance	In stack Precedance
+, −	1	2
*, /	3	4
Right to Left Associativity ^	6	5
(	7	0
)	0	?

outside the Stack — Inside the Stack



Postfix :- $ab+c*def\wedge\wedge-$

Important Note :- ① When the Precedances are equal and they will be equal only in the case where we have opening bracket and closing bracket, otherwise in this Precedances, that is instack and outstack Precedances will never be equal.

② The Purpose of giving two Precedances is that, if it is left to right then increase it, when it is going into the stack. otherwise when it is right to left, let it decrease, so if we want to add more operators, we can add, depending upon the associativity, we can increase or decrease the Precedances.

de) :-

```
Struct Node
{
   int data;
   Struct Node * next;
} * top = NULL;

int main ()
{
   char * exp = "(a+b) * (c-d) ^ e ^ f";
   char * Result = intopost (exp);
   printf ("%s", Result);
  // printf (Result); // it will also work
}
void Push (char x)
{
   Struct Node *t;
   t = (struct Node *) malloc (size of (struct Node));
   if (t == NULL)
      printf ("Stack is full \n");
   else
     { t -> data = x;
       t -> next = top;
       top = t;
     }
  }
 Char Pop ()
 {
    struct Node *t;
      Char x = -1;
    if (top == NULL)
       printf ("Stack is Empty \n");
    else
     { t = top;
       top = top -> next;
       x = t -> data;
     } free (t);
  }  return x;
```

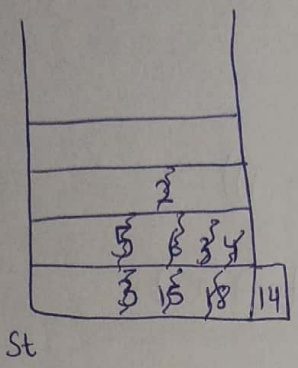
```
int isEmpty()
{
    return ((top == NULL) ? 1 : 0);
}
int isOperand(char ch)
{
    if (ch == '+' || ch == '-' || ch == '/' || ch == '*' || ch == '('
        || ch == '^')
        return 0;
    else
        return 1;
}
int outPre(char x)
{
    if (x == '+' || x == '-')
        return 1;
    if (x == '*' || x == '/')
        return 3;
    if (x == '^')
        return 6;
    if (x == '(')
        return 7;
    if (x == ')')
        return 0;
    else
        return -1;
}
int inPre(char x)
{
    if (x == '+' || x == '-') return 2;
    if (x == '*' || x == '/') return 4;
    if (x == '^') return 5;
    if (x == '(') return 0;
    else
        return -1;
}
```

```
char * intopost (char * infix)
{
  int i=0, j=0;
  char * Postfix;
  int len = strlen (infix);
  Postfix = (char *) malloc ((len+2) * (sizeof (char)));
  while (infix[i] != '\0')
  {
     if (isoperand (infix[i]))
       Postfix [j++] = infix [i++];
     else
      {
        if (top == NULL || outpre (infix[i] > inpre (top -> data))
          Push (infix [i++]);
        else if (outpre((infix[i]) < inpre (top -> data))
          Postfix [j++] = POP();
        else
          {
            POP();
            i++;
          }
      }
  }
  while (! isEmpty())
    Postfix [j++] = POP();
    Postfix [j] = '\0';
    return Postfix;
}
```

★ Evaluation of Postfix Expression :-

3*5 + 6/2 - 4 → infix form

Postfix form → 35 * 62/ + 4 -



3 * 5 = 15

6 / 2 = 3

15 + 3 = 18

18 - 4 = 14

Symbol	Stack	Operation
3	3	
5	5, 3	
*	15	3 * 5 = 15
6	6, 15	
2	2, 6, 15	
/	3, 15	6/2 = 3
+	18	15 + 3 = 18
4	4, 18	
-	14	18 - 4 = 14

Time Complexity :- O(n) → Single Scan through out the postfix Expression.

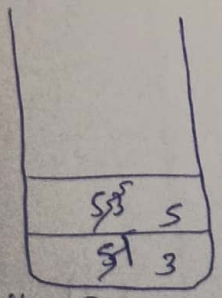
Procedure :- ★ We will scan through the expression by taking one symbol at a time and while scanning do these steps as follows :-

- if it is an operand, push it into the stack, next will be an operand, then again push it into the stack.
- If we get any operator, then pop out 2-symbols from the stack and perform that operation and the first value that popped out will come on Right-hand side and the second value will come on Left-hand side and we will perform this operation.
- Then Push this result into the stack
- Then Proceed so on by applying above steps.

Program :- Postfix

3	5	*	6	2	/	+	4	-	\0	→ characters
0	1	2	3	4	5	6	7	8	9	

```
int Eval (char * Postfix)
{
    struct stack st;
    int i, x1, x2, r;
    for (i=0 ; Postfix[i]!='\0' ; i++)
    { if (isoperand (Postfix[i]))
            Push (&st, Postfix[i] - '0'); // We are Subtracting
      ASCII Code of '0', to convert char to integer in stack
      else
      { x2 = Pop (&st) ; x1 = Pop (&st)
        switch (Postfix[i])
        { Case '+' ; r = x1 + x2 ; Push (&st, r) ; break;
          Case '-' ; r = x1 - x2 ; Push (&st, r) ; break;
          Case '*' ; r = x1 * x2 ; Push (&st, r) ; break;
          Case '/' ; r = x1 / x2 ; Push (&st, r) ; break;
        }
      }
    return Pop (&st);
}
```



Conversion of character to integer

$= '3' - '0'$
$= 51 - 48$
$= 3$

Code :-

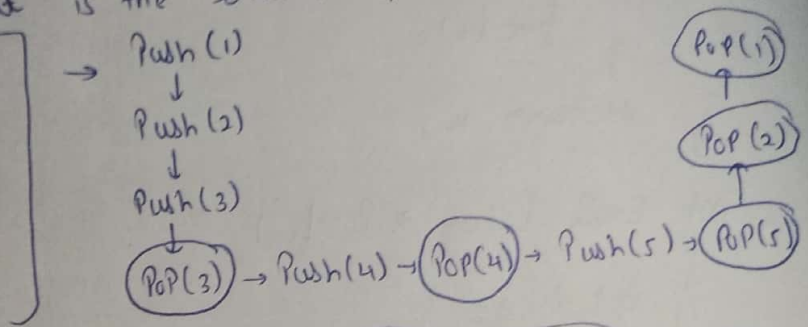
```
Struct Node
{
   int data;
   Struct Node *next;
 }* top = NULL;

int main ()
{
   char * Postfix = "234 * + 82 /- ";
   int eval (char * Postfix);
   int P = eval (Postfix);
   printf ("Result is %d", P);
}
Void Push (char x)
{
   Struct Node *t;
   t= (Struct Node *) malloc (sizeof (Struct Node
   if (t == NULL)
      printf ("Stack is Full \n");
   else
    {
       t → data = x;
       t → next = top;
       top = t;
    }
}
int isoperand (char x)
{
   if (x == '+' || x == '-' || x == '*' || x == '/')
      return 0;
   else
}     return 1;
```

```
int PoP()
{
  struct Node *t;
  int x = -1;
  if (top == NULL)
     printf ("Stack is Empty \n");
  else
    {
      t = top;
      top = top -> next;
      x = t -> data;
      free (t);
    }
  return x;
}
int Eval (char * Postfix)
{
  int i = 0;
  int x1, x2, r;
  for (i=0 ; Postfix [i] != '\0' ; i++)
    {
      if ( isoperand (Postfix [i]))
         Push (Postfix [i] - '0');
      else
        {
          x2 = PoP();
          x1 = PoP();
          switch (Postfix [i])
          {
            Case '+':
              r = x1 + x2;
              break;
            Case '-':
              r = x1 - x2;
              break;
            Case '*':
              r = x1 * x2;
              break;
            Case '/':
              r = x1 / x2;
              break;
          }
          Push (r);
        }
    }
```

PRACTISE SET

Ques-1 :- Which of the following Permutations can be obtained in the output (in the same order) using a stack assuming that the input is the sequence 1, 2, 3, 4, 5 in that order ?

(a) 3, 4, 5, 1, 2
(b) 3, 4, 5, 2, 1
(c) 1, 5, 2, 3, 4
(d) 5, 4, 3, 1, 2



Ques-2 :- The Postfix Expression for the infix Expression A + B * (C + D) / F + D * E is

Answer-2

A + B * (C + D) / F + D * E

A + B * [CD +] / F + D * E

A + [BCD +] / F + D * E

A + [BCD + * F /] + D * E

A + [BCD + * F /] + [DE *]

[ABCD + * F / ●] + [DE *]

ABCD + * F / ● + DE * +

Qus-3 :- Which of the following is essential for converting an infix Expression to the postfix form efficiently?

Answer-3 :- ~~(i)~~ An operator stack
(ii) An operand stack
(iii) A Parse tree.
(iv) An operand stack an operator stack

Qus-4 :- Assume that the operators +, -, * are left associative and ^ is right Associative. The order of Precedence (from highest to lowest) is ^, *, +, -, The postfix Expression corresponding to the infix expression a+b * c-d ^ e ^ f is

Answer-4 :-

a+b * c-d^e^f

a+b * c-d^[ef^]

a+b * c-[def^^]

a+[bc*]-[def^^]

[abc*+]-[def^^]

abc*+def^^-

Ques-5 :- The following Postfix Expression with Single digit operands is evaluated using a stack.

8 2 3 ^ / 2 3 * + 5 1 * -

Note that ^ is the exponentiation operator. The top two Elements of the stack after the first * is evaluated are.

Answer-5 :-

Symbol	Stack	Operation
8	8	
2	2, 8	
3	3, 2, 8	
^	8, 8	2 ^ 3 = 8
/	1	8/8 = 1
2	2, 1	
3	3, 2, 1	
*	6, 1	2 * 3 = 6

So, the required answer is 6, 1

Ques-6 :- Let S be a stack of size $n \geq 1$. Starting with the empty stack, suppose we push the first 'n' natural numbers in sequence, and then perform 'n' Pop operations. Assume that Push and Pop operation take x seconds each, and Y seconds elapse b/w the end of one such stack operation and the start of the next operation. For $m \geq 1$, define the stack-life of 'm' as the time elapsed from the end of Push(m) to the start of the Pop operation that removes m from S. The average stack-life of an element of this stack is :-

Ans - 6 :-

Now, Stack life time of last- Element = Y — (1)

(bcz it is popped as soon as it is pushed on the stack)

Now, Stack life time of (n-1) th element = Y + 2X + 2Y

= 2X + 3Y

Stack life time of (n-2) th element = 2X + 3Y + 2X + 2Y

= 4X + 5Y

So, Stack life time of 1st element = 2(n-1) X + (2n-1) Y

Now, Sum of all time spans of all the elements = $\Sigma(2(n-1)X) + \Sigma((2n-1)Y)$

$= 2X\,(1+2+\cdots+(n-1)) + Y\,(1+3+5+\cdots+(2n-1))$

$= (2X)\left(\frac{n(n-1)}{2}\right) + Y\left(\frac{n}{2}\right)(1+2n-1)$

$= n(n-1)X + n^2Y$

$= n[(n-1)X + nY]$

Now, Average Stack life of an element $= \frac{n[(n-1)X + nY]}{n}$

$= nX - X + nY$

$= n(X+Y) - X$

Question -7 :- A Single Array A[1... MAXSIZE] is used to implement two stacks. The two stack grow from opposite ends of the Array. Variable top1 & top2 (top1 < top2) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, then condition for stack full is :-

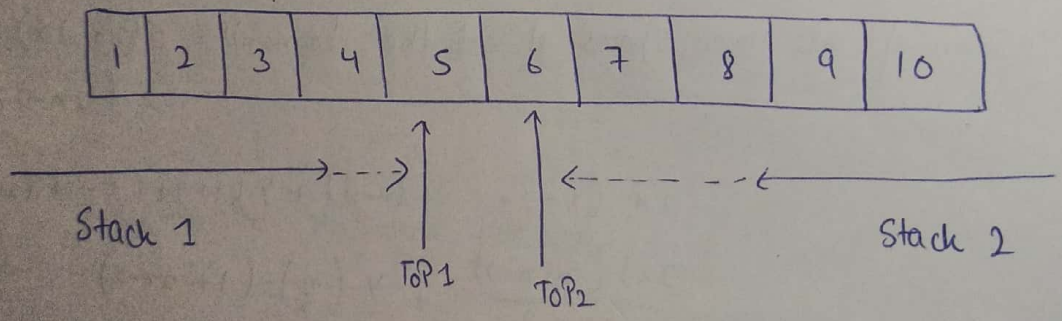
(a) (top1 = MAXSIZE/2) and (top2 = MAXSIZE/2 + 1)

(b) top1 + top2 = MAXSIZE

(c) (top1 = MAXSIZE/2) or (top2 = MAXSIZE)

(d) top1 = top2 - 1

Answer -7 :- Let MAXSIZE = 10



Now, if we are to use space efficiently then size of the any stack can be more than MAXSIZE/2. Both stacks will grow from both the ends and if any of the stack reaches near to the other top then stacks are full. So the condition will be top1 = top2 - 1

[∵ given that top1 < top2]