Recursion uses stack

STACK

→ **LIFO** : Last In First Out

It is a collection of elements that follow LIFO for insertion and deletion.

ADT Stack (Abstract data type)

Implementation of stack using
1. Array
2. Linked List

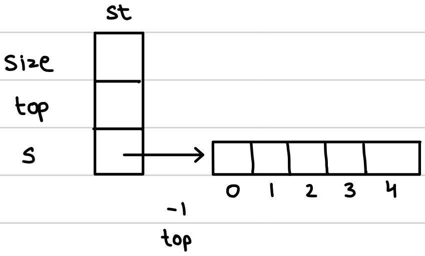
Data : 1. Space for storing elements
2. Top pointer

Operations: 1 push(x)
2. pop()
3. peek(index)
4. StackTop()
5. isEmpty()
6. isFull()

IMPLEMENTATION OF STACK USING ARRAY

```
Struct stack
{
      int  size;
      int  top;
      int  *s;
};

int  main()
{
      Struct stack st;
      printf(" Enter size of stack ");
      scanf (" %d ", &st.size);
      st.s =  new  int[st.size];    for heap memory
      st.top = -1;
 }
```



Stack empty — if (top == -1)
Stack full - if (top == size-1)

push() O(1)

```
void push ( stack *st , int x)
{
        if ( st → top ==  st → size -1)
                printf (" Stack Overflow");
        else
        {
                st → top ++;
                st → s [st → top] = x;
        }
}
```

peek()

```
int peek( stack st , int pos)
{
        int x = -1;

        if ( st top - pos +1 < 0)
                printf (" Invalid Position");
        else
                x = st . s [st . top - pos+1];

        return x;
}
```

Stacktop()

```
int stacktop (stack st)
{
        if (st . top == -1)
                return -1;
        else
                return st . s [st . top];
}
```

pop() O(1)

```
int pop (stack *st)
{
        int x = -1;

        if ( st → top == -1)
                printf (" Stack Underflow");
        else
        {
                x = st → s [st → top];
                st → top --;
        }
        return x;
}
```

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

isEmpty()

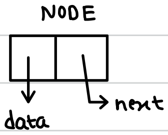
```
int isEmpty (stack st)
{
        if (st . top == -1)
                return 1;
        else
                return 0;
}
```

isFull()

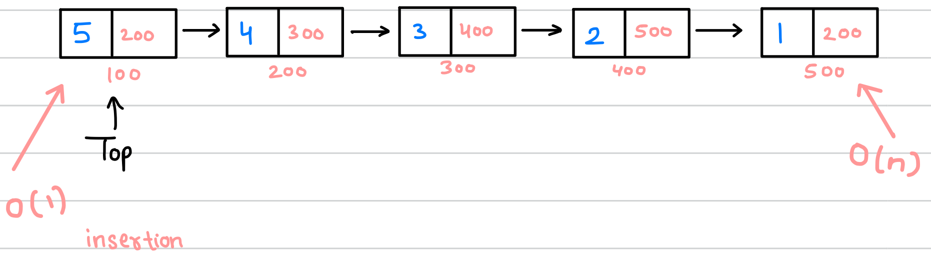
```
int isFull ( stack st)
{
        if (st . top == -1)
                return 1;
        else
                return 0;
}
```

STACK USING LINKED LIST

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



Empty : if (top == NULL)
Full : Node *t = new Node;
if (t==NULL)



push()

```
void push( int x)
{
        Node * t = new Node;

        if (t==NULL)
                printf("Stack Overflow");
        else
        {
                t -> data = x;
                t -> next = top;
                top = t;
        }
}
```

pop()

```
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;
}
```

peek()

```
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;
}
```

Stacktop()

```
int  stacktop()
{
        if(top)
             return  top→data;
        return  -1;
}
```

isEmpty

```
int  isEmpty()
{
        return  Top ? 0 : 1;
}
```

False

True

isFull

```
int   isFull()
{
        Node *t = new  Node;
        int  r = t ? 1 : 0;
        free(t);
        return r;
}
```

Paranthesis Matching

exp	(	(	a	+	b	)	*	(	(	-	d	)	)	\0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
int  isBalance(char *exp)
{
       Struct  stack  st;
       st.size = strlen (exp);
       St.top = -1;
       St.s  =  new  char[st size];

       for(i=0; exp[i]! = '\0'; i++)
       {
             if (exp[i] == '(')
                    push (&st, exp[i]);
             else  if (exp[i] == ')')
                   if (isEmpty (st))
                          return false;
                   else
                          pop (&st);
       }
       return  isEmpty(st)? true : false;
}
```

initializing of stack

ASCII

(	40
)	41
[	91
]	93
{	123
}	125

INFIX TO POSTFIX CONVERSION

1. What is postfix
2. Why postfix
3. Precedence
4. Manual Conversion

1. **Infix**: Operand Operator Operand
   a + b

2. **Prefix**: Operator Operand Operand
   +ab

3. **Postfix**: Operand Operand Operator
   ab +

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, −	1	L-R
*, /	2	L-R
^	3	R-L
—	4	R-L
()	5	L-R

Unary minus

Eg a + b*c

(a + (b*c)) First fully paranthecise it.

prefix

(a + [*bc])

[+a*bc]

postfix

(a + [bc*])

[abc*+]

ASSOCIATIVITY

Left to Right

a+b+c−d

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

Right to Left

a = b = c = 5

$(a=(b=(c=5)))$

Power operator Example

a ^ b ^ c

(a ^ (b ^ c))

Postfix: (a ^ [b c ^])

a b c ^ ^

Unary Operators Example

(1) − a negation of a

pre : − a

post : a −

(− (− a))

(2) * p

pre : * p

post : p *

(* (* p))

(3) n !

pre : ! n

post : n !

(4) log x

pre : log x

post : x log

Example : − a + b * log n !

− a + b * log [n !]

− a + b * [n ! log]

[a −] + b * [n ! log]

[a −] + [b n ! log *]

a − b n ! log * +

INFIX TO POSTFIX CONVERSION

a + b * c − d / e

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, −	1	L-R
*, /	2	L-R

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

abc * + de / − Ans

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=0 ; j=0 ;                          ↳ for null string

        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 ( &t );
                }
        }

        while ( ! isEmpty (st))
                postfix [j++] = pop ( &st);

        postfix [j] = '\0';
        return postfix ;
}
```

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

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

Q ((a+b)*c) − d^e^f
([ab+]*c) − d^e^f
[ab+c*] − d^e^f
[ab+c*] − d^[ef^]
[ab+c*] − [def^^]
ab+c*def^^−

SYMBOL	OUT STACK PRE	IN STACK PRE
+, −	1	2
*, /	3	4
^	6	5
(	7	0
)	0	?

because of R-L associativity

closing bracket cannot be pushed into Stack

EVALUATION OF POSTFIX

35*62/+4−

SYMBOL	STACK	OPERATION
3	3	
5	5, 3	
*		5*3
	15	
6	6, 15	
2	2, 6, 15	
/		6/2
	3, 15	
+		15+3
	18	
4	4, 18	
−		18−4
	14	

$x = 6+5+3*4$
$x = 65+34*+$

* Here + gets executed first instead of * because presedence and associativity are meant for parenthecisation, they don't decide which operator gets executed first.

PROGRAM FOR EVALUATION OF POSTFIX

postfix	3	5	*	6	2	/	+	4	−	\0
	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') ;
             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);
}
```

because operand will be pushed into the stack in its ASCII value because postfix expression is in char.

For eg : 3

'51' – '48' = 3