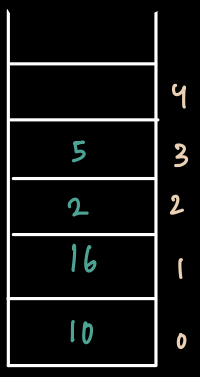
STACK

IN DSA

Introduction

LIFO : Last In First Out

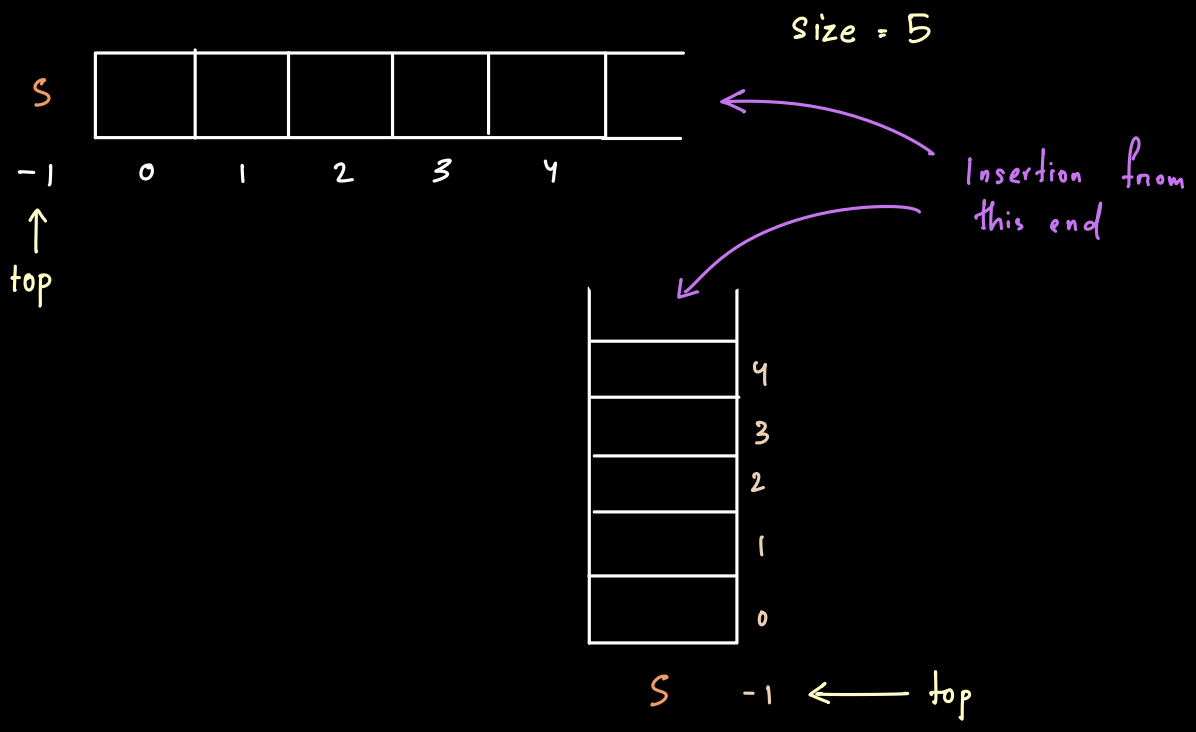
ADT STACK



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

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

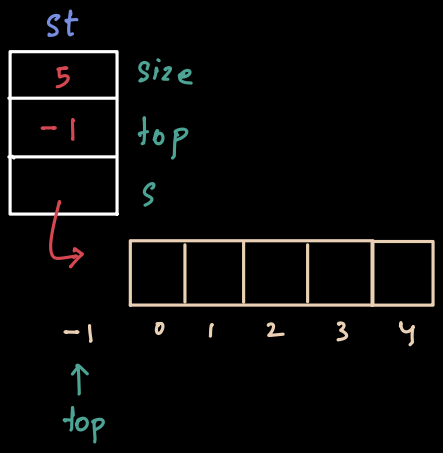
(1) Stack using Array



```
Struct Stack {
    int size ;
    int Top ;
    int *s ;          ---> pointer to an array
};

int main () {
    struct Stack st ;
    printf ("Enter size of stack : ");
    scanf ("%d", &st.size) ;

    st.s = new int [st.size] ;
    st.Top = -1 ;
}
```



- Empty

```
if ( Top == -1 )
```

- Full

```
if ( Top == size - 1 )
```

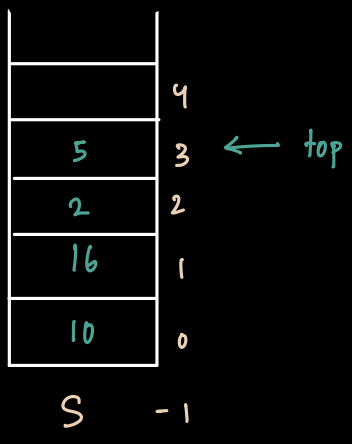
- Push ---> time complexity : constant

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

o pop → time complexity : constant

```
int push ( stack *st ) {
    int x = -1 ;
    if ( st->Top == -1 )
        printf ( "Stack underflow" ) ;
    else {
        x = st -> S [ st -> Top ] ;
        st -> Top -- ;
    }
    return x ;
}
```

o peek → time complexity : constant



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

```
int peek (stack *st, int pos ) {
    int x = -1 ;
    if ( st. Top - pos + 1 < 0)
        printf ( " pos invalid " ) ;
```

```
        else {
            x = st.s[st.Top - pos + 1];
        }
        return x;
    }
```

o stackTop —> const. time

```
int stackTop(Stack st) {
    if (st.Top == -1)
        return -1;
    else
        return st.s[st.Top];
}
```

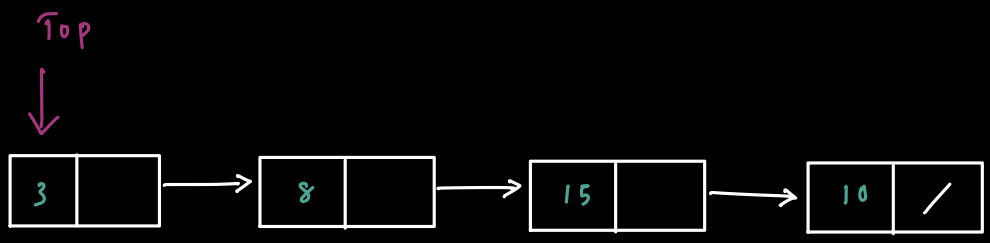
o isEmpty —> const. time

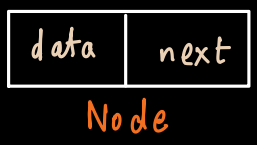
```
int isEmpty(Stack st) {
    if (st.Top == -1)
        return 1;
    else
        return 0;
}
```

o is Full ——> const. tim

```
int is Full (stack st){
    if ( st. Top = st.size - 1 )
            return 1 ;
    else
            return 0 ;
}
```

(2) Stack using Linked List






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

o Insertion & Deletion —> O(1)

o Empty

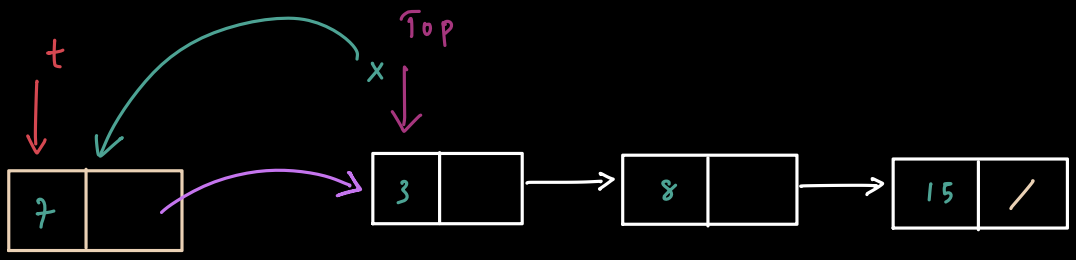
```
if ( Top == NULL )
```

o Full

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

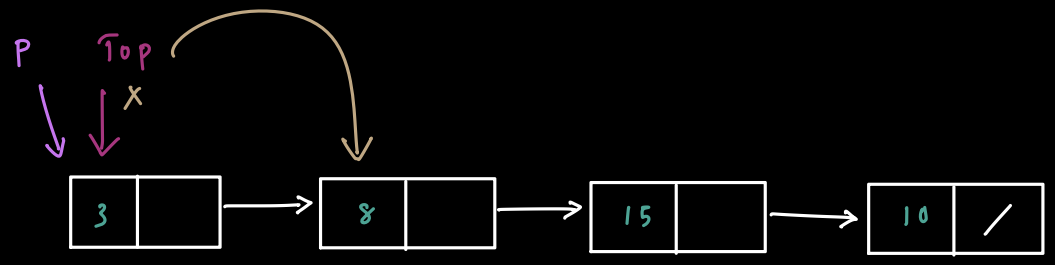
Stack never gets full untill all spaces in heap is utilised.

o Push



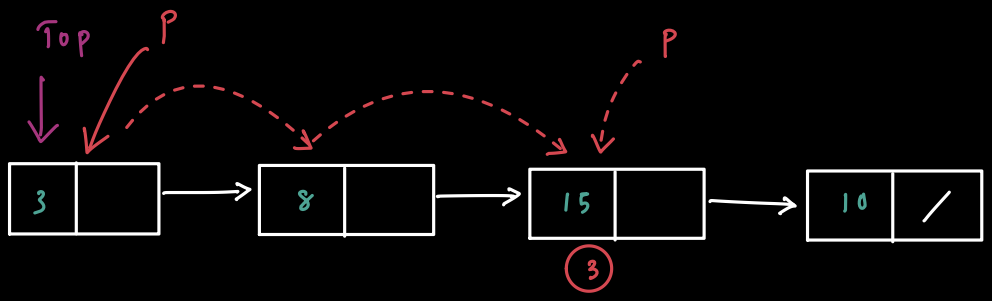

```
void push ( int x ) {
    Node *t = (Node *) malloc ( sizeof ( Node )) ;
    if ( t == NULL )
        printf ( " Stack Overflow " ) ;
    else {
        t -> data = x ;
        t -> next = Top ;
        Top = t ;
    }
}
```

o 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 ;
}
```

o peak




```
int peek ( int pos ) {

    Node *p = Top ;
    for ( int i=0 ; p != NULL && i<pos-1 ; i++ ) {
            p = p -> next ;
    }
    if ( p != NULL )
        return p -> data ;
    else
        return -1 ;
}
```

o stackTop

```
int stackTop ( ) {
    if ( Top )
        return Top -> data ;
    return -1 ;
}
```

o isEmpty

```
int isEmpty(){
    return top ? 0 : 1;          // If top is NULL -> empty
}
```

o isFull

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

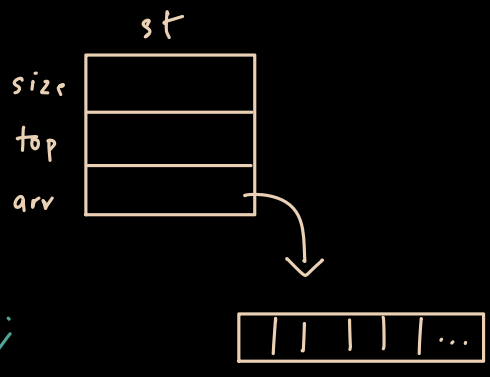
o Paranthesis matching

```
struct stack {
    int size;
    int top;
    char *arr;          // As string is being passed
}
```

exp

(	(	a	+	b	)	*	c	)	\0
0	1	2	3	4	5	6	7	8	9

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



```
// Initialise stack
for ( int 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 ;
```

Expression Conversion

1. Infix : Operand Operator Operand

   eg : a + b

2. Prefix : Operator Operand Operand

   eg. + a b

3. Postfix : Operand Operand Operator

   eg. a b +

1. Infix to Postfix Expression

1. a + b * c

symbol	Precedence	Associ.
+, −	1	L-R
*, /	2	L-R
(,)	3	

a + (b * c)

≡ (a + (b * c))

1

2

prefix

≡ (a + (* b c))

≡ + a * b c

postfix (mostly used)

≡ (a + (b c *))

≡ a b c * +

(2) a+b+c*d

prefix

≡ a+b+(*cd)

when operators of same precedence are there then take left to right.

≡ (+ab)+(*cd)

≡ ++ab*cd

postfix

≡ a+b+(cd*)

≡ (ab+)+(cd*)

≡ ab+cd*+

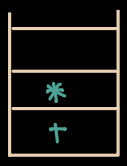
METHOD 1

symbol	Precedence	Associ.
+, −	1	L-R
*, /	2	L-R

ex: a+b*c−d/e → Infix

	Stack	Push/pop	postfix
1)		—	a
2)	+	push +	a
3)	+	—	ab

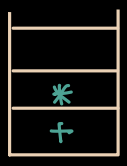
4)



push *
(as * has ↑ precedence)

ab

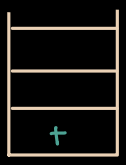
5)



—

abc

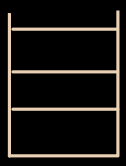
6)



pop *
(as − has ↓ precedance than *)

abc *

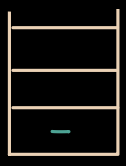
7)



pop +

abc * +

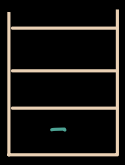
8)



posh −

abc * +

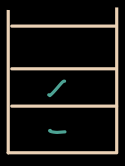
9)



—

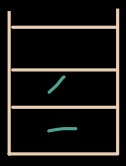
abc * + d

10)

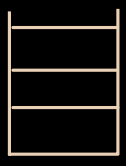


push /

abc * + d

11) 

abc*+de

12) 

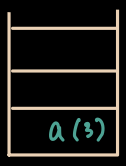
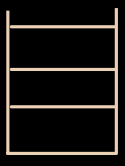
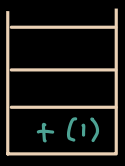
pop / and then -

abc*+de/-

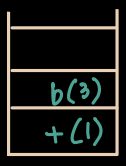
METHOD 2

exp: a+b*c-d/e

symbol	Precedence	Associ.
+, -	1	L-R
*, /	2	L-R
a,b,c,...	3	L-R

Stack	Push/pop	postfix
1) 	push a	
2) 	pop a	a
3) 	push +	a

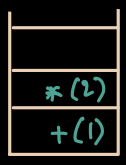
4)



push b

a

5)



pop b

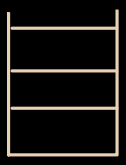
ab

6)

push c

ab

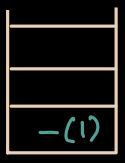
7)



pop c then
* and then +

abc * +

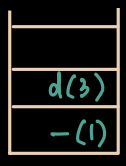
8)



push -

abc * +

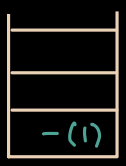
9)



push d

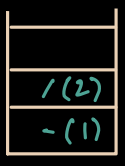
abc * +

10)



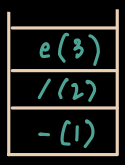
pop d

abc * + d

11) 

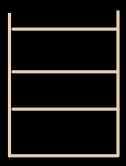
push /

abc*+d

12) 

push e

abc*+d

13) 

pop e, / and -

abc*+de/-

Program

```
struct stack {
    int size;
    int top;
    char *arr;
}

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

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

char *Convert(char *infix){
    struct stack *st;
    char *postfix = new char[strlen(infix)+1];
    int i=0, j=0;

    while(infix[i]!='\0'){
        if(isOperand(infix[i])){
            postfix[j++] = infix[i++];    // char stored in j and both
                                          // i,j incremented
        }else{
            if(presedence(infix[i]) > presedence(stackTop(st)))
                push(&st, infix[i++]);
            else
                postfix[j++] = pop(&st);
        }
        while(!isEmpty(st)){
            postfix[j++] = pop(&st);
        }
        postfix[j] = '\0';
        return postfix;
    }
}
```

Postfix Evaluation

eg: $3*5+6/2-4$ —BODMAS→ $15+3-4$

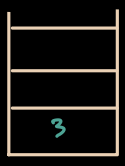
$= 18-4$

$= 14$

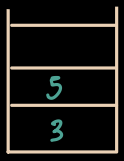
Postfix: $(35*)+(62/)-4$

$= (35*62/+)-4$

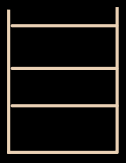
$= 35*62/+4-$

1) 

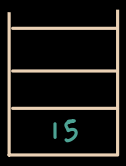
push 3

2) 

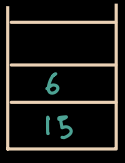
push 5

3) 
pop two elements and perform operation.

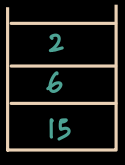
$3*5 = 15$

4) 

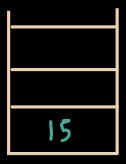
push 15

5) 

push 6

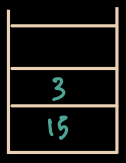
6) 

push 2

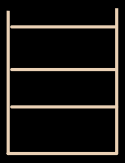
7) 

/ encountered. 6 / 2 = 3

pop 2 elements.

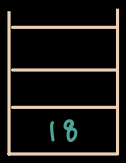
8) 

push 3

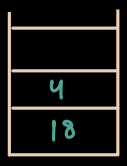
9) 

+ encountered. 15 + 3 = 18

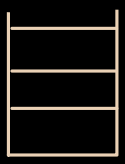
pop 3 and 15

10) 

push 18

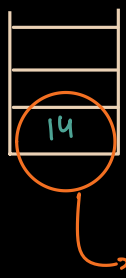
11) 

push 4

12) 

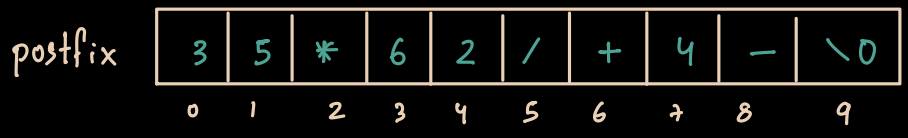
− encountered 18 − 4 = 14

pop 4 and 18

13) 

push 14

Program




```
int Eval ( char *postfix )
{
    struct stack st ;
    int x1, x2, r ;
    for ( int 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 ;
```

// 3 when pushed to stack, its asci code is pushed. Type casting doesn't help. Instead substn. '0' from a no. in asci gives its int val.

eg: '3' - '0'
-> 51 - 48
= 3

```
            case '*' :  r = x1 * x2 ;
                        push(&st, r);
                        break;

            case '/' :  r = x1 / x2 ;
                        push(&st, r);
                        break;
        }
    }
    return  pop(&st);
}
```