STACKS

Stack is a linear non-primitive data structure where in the insertion and the deletion operation are done from the same end referred as top of the stack.

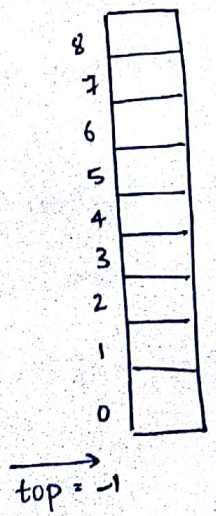
The insertion operation is referred as push and deletion operation is reffered as pop.

In a stack, the last element inserted will be the first one to be retreived out. Hence stack is also termed as LIFO

Implementation of a stack. (using an array)

Let 'SIZE' denotes the size of a stack which is implemented using an array data[SIZE].

Let 'top' denote the end used for 'push' and 'pop' operation with an initial value of -1 which is represented through the following diagram.



push operation

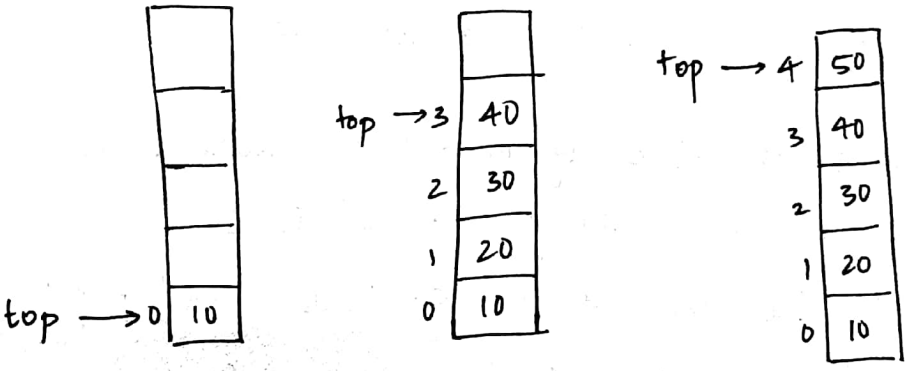
before inserting an element onto a stack, we need to check whether stack is full or not. The condition to denote stack full is

```
if (top == SIZE-1)
```

if stack is not empty, the push operation can be performed by the statement

```
data [++ top] = item;
```

If stack is full and if we try to perform push, it is called as overflow.



A C function to perform push operation.

```
void push (int ele)
{
    if (top == SIZE-1)
        pf ("Stack Overflow");
    else
        data (++top) = ele;
}
```

'pop' operation

The main objective in the 'pop' is to retrieve the top most element from the stack. Before retrieving, check whether stack is empty or not. The condition to check stack empty is

```
if (top == -1)
```

If stack is not empty, the popped element is

```
data[top--];
```

A c function to perform pop operation.

```
int pop()
{
    if (top == -1)
    { pf("Stack underflow\n");
      return -1; }
    else
        return data[top--];
}
```

'display' operation

```
void display()
{   int i;
    if (top == -1)
    { pf("Stack is empty\n");
    else
    { pf("Stack content is\n");
      for (i=top; i>0; i--)
      {
          printf("%d\n", data[i]);
      }
    }
}
```

C program to implement stack of integers

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 20
int top=-1;
int data[SIZE];
// push function
// pop function
// display function.
int main()
{
    int ch, ele, e;
    for(;;)
    {
        pf("1-push\n2-pop\n3-display\n4-terminate\n");
        pf("Read choice:\n");
        sf("%d", &ch);
        switch(ch)
        {
            case 1: pf("Enter element to be pushed\n");
                    scanf("%d", &ele);
                    push(ele);
                    break;
            case 2: e=pop()
                    if(e!=-1)
                    pf("popped element is %d", &e);
                    break;
            case 3: display.
                    break;
        }
    }return 0;
```

Program to implement stack using structures and pointers

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
struct stack
{
    int top;
    int data[SIZE];
};
typedef struct stack STACK;
void push(STACK *p, int ele)
{
    if (p->top == SIZE-1)
        pf("Stack Overflow\n");
    else
        p->data[++(p->top)] = ele;
}
int pop(STACK *s)
{
    if (s->top == -1)
    {   pf("Stack underflow\n");
        return -1;
    }
    else
        return s->data[(s->top)--];
}
void display(STACK s)
{
    int i;
    if (s.top == -1)
        pf("Stack is empty\n");
    else
    {
```

```
        pf("Stack contents are\n");
        for (i=s.top ; i>-1 ; i--)
        {
            pf("%d\n", s.data[i]);
        }
    }

int main()
{
    ... STACK s;
    s.top = -1;
    :
    push(&s, ele)
    :
    pop(&s)
    :
    display(s)
}
```

function to pop n elements.

```
void popn(STACK *s, int n)
{
    if (n > s->top+1)
    {   pf("error")
    else
    {
        pf ...
        for(i=0; i<n ; i++)
            pf("%d\n", pop(s));
    }
```

function to pop the bottom 3rd element

```
void popb (STACK *s, int n)
{
    STACK s1; int n;
    s1.top = -1; int x;
    n = (s -> top + 1);
    if (n<3)
        pf("Invalid");
    else
    {
        for (i = s -> top; i > 2; i--)
            push (&s1, pop(s));
        x = pop (s);
        pf("Element popped is %d", x);
        push(s, x);
        while (s1.top != -1)
            push (s, pop (&s1));
    }
}
```

function to pop 3rd element from the top

```
void popt (STACK *s)
{
    STACK s1; int n;
    s1.top = -1; int y;
    ...
    else
    {
        push (&s1, pop(s));
        push (&s1, pop(s));
```

```
        y = pop(s);
    push(s,y);
    pf("Element popped is %d", y);
    while (s1.top != -1)
        push(s, pop(&s1));
  }
}
```

WAP to check if a given string is palindrome or not using stack.

```
struct stack
{
    int top;
    char data[SIZE];
};

typedef struct stack STACK;

void push(STACK *s, char ele)
{
    s->data[++(s->top)] = ele;
}

char pop(STACK *s)
{
    return s->data[(s->top)--];
}

int check
```

Expression

The meaningful collection of operands and operators is termed as an expression.

There are 3 types of expressions, mainly

1. Infix expression

   here, the operator is found in between 2 operands

   eg: -a+b

2. Postfix (suffix) expression.

   here, the operator is found at the end of after 2 operands. eg: ab+

3. Prefix expression

   .... before 2 operands.

   eg: +ab.

following rules need to be followed while converting an expression from one form to another form.

1. Always the expression should be evaluated from left to right

2. The operators with the higher preference should be evaluated first

   higher precedence → $ (expomential)

   medium → *, /

   lower → +, -

3, If the part of the expression is in (), should be evaluated first.

① a+b*c

a+ bc*

a bc* + (postfix)

a+ *bc

+a*bc (prefix)

② a+b-c

ab+ -c

ab+c- (postfix)

+ab-c

-+abc (prefix)

③ (a+b)*c

(ab+)*c

ab+c* (postfix)

+ab*c

*+abc (prefix)

④ (((a+(b-c)*d)$e)+f)

(((a+ ~~bc~~ T_1 *d)$e)+f)

T_1 = bc-

((a+ ~~T1~~ T_2) $e)+f)

T_2 = T_1 d*

((T_3 $e)+f)

T_3 = aT_2 +

(T_4+f)

T_4 = T_3 e$

T_4f +

T_3e $f+

aT_2+ e$f+

aT_1d* +e $f+

abc-d*+e$f+ (postfix)

(((a +T_1 *d)$e)+f)

+ $+a*-bcdef (prefix)

Note:-

the exponential operator is right associated.

abc - d/e

abc - d/e

abc$$ - d/e

abc $$ - de/

abc $$ de/-

Algorithm to convert the given infix exp to post fix.

step 1: scan the given infix exp from left to right.

step 2: if the scanned symbol is -

1) an operand - place it on the postfix exp.
2) if left paranthesses - push it onto stack.
3) if right paranthesses - pop the content of stack and place it on postfix exp until we get a corresponding left paranthesses
4) if operator - if stack is empty or top of stack is having left paranthesses, then push the operator onto stack.

else check the precedence of top of stack with the precedence of the scanned symbol,

if sym ≤ stack[top] → if it is greater or equal to, pop the stack content and place on postfix exp. and push the scanned symbol onto the stack

else,
push symbol onto stack

step 3 : until stack become empty, pop the stack content and place on postfix exp.

① a+b*c

ANS IS IMPORTANT

symbol	stack	postfix.
a	empty	a
+	+	a
b	+	ab
*	+,*	ab.
c	+,*	abc
	+	abc*
	empty	abc*+

② (a+b)*c

symbol	stack	postfix
(	(	empty.
a	(	a
+	(,+	a
b	(+	ab
)	~~(~~ empty.	ab+
*	~~(~~*	ab+
c	~~(~~*	ab+c
		ab+c*

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

symbol	stack	postfix.
(	(((	empty
a	(((	a
+	(((+	a
(	(((+(	a.
b	((+(	ab
-	(((+(-	ab
c	(((+(-	abc
)	(((	abc-
*	(((*	abc-
d	(((*	abc-d
)	((	abc-d*+
$	(($	abc-d*+
e	(($	abc-d*+e
)	(	abc-d*+e$
+	(+	abc-d*+e$
f	(+	abc-d*+e$f
		abc-d*+e$f+

a*b-c

sym	stack	postfix
a	empty	a
*	*	a
b	*	ab
-	-	ab*
c	-	ab*c
		ab*c-

NOTE.
pop until condition is false. (comparing precedence).

~~a~~	+/*	abc
-	+-	abc*/

Write C func to convert a given infix exp to postfix.

```
void infix to postfix (STACK *s, char infix[15])
{   char symbol; int i,j=0; char postfix[15], temp
    for (i=0 ; infix[i]!='\0' ; i++)
    {
        symbol = infix[i];
        if (~~isalpha~~ isalnum (symbol))
            postfix[j++] = symbol;
        else
        {
            switch (symbol)
            {
                case '(': push(s, symbol);
                          break;
                case ')': temp = pop(s)
                          while (temp!='(')
                          {
```

```
                postfix[j++] = temp;
                temp = pop(s);
            }

        case '+':
        case '-':
        case '*':
        case '/':
        case '$' : if (s->top==-1||s->data[s->top]!=(

                    push(s,symbol);

                else
                {
                  while
                (preced(s->data[s->top] >= preced(symbo
                  && s->top == -1 &&
                   s->data[s->top] != '(')
                     postfix[j++] = pop(s);
                push(s,symbol);
                }

        default: pf("\n Invalid");
                 exit(0);
    }
}
while (s->top != -1)
    postfix[j++] = pop(s);
postfix[j] = '\0';
pf("\n The postfix expr is %s", postfix.
}
```

WAP to convert infix to postfix.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 20

struct stack
{
    int top;
    char data[15];
};
struct stack STACK;

// push func
// pop func

int preced(char a)
{
    switch(a)
    {
        case '$': return 5;

        case '*':
        case '/': return 3;

        case '+':
        case '-': return 1;
    }
}

// infix to postfix

int main()
{
    STACK s
    s.top = -1;
    char infix[20];
```

```
    pf ("Read infix\n");
    sf ("%s", infix);
    infix to postfix (infix);
}.
```

WAP to check for parantheses balancement using stack.

```
int check (char brack[20])
{
    int, i
    STACK *s
    s.top = -1;
    if (strlen(brack)/2 != 0)
    {
        return 0;
    }
    else
    {
        for (i=0 ; brack[i] != '\0'; i++)
        {
            if (brak[i] == '(' || brack[i] == '{' || brack[i] == '[')
                push (s, brack[i]);
            if (brack[i] == ')' || brack[i] == '}' || brack[i] == ']')
            { if (s->top == -1)
                return 0;
              else
              {
                if (brack[i] == s->data[s->top])
                    pop (s);
                else
                    return 0;
              }
        }
    }
```

```
    if (s->top == -1)
        return 1;
    return 0;
}
```

Evaluation of a postfix expression :-

ALGORITHM.

Step 1: Scan the given postfix expression from left to right.

Step 2: If the scanned symbol is :

i) operand : push it onto stack.

ii) operator : pop 2 elements from stack and assign them to operand 2 and operand 1 respectively and find the value operand 1 operator operand 2 and push the value onto the stack.

step 3: pop content of the stack to get final answer.

Trace the algo for the following inputs.

① 345*+

Symbol	Stack	Op1	Op2	value
3	3	-	-	-
4	3,4	-	-	-
5	3,4,5	-	-	-
*	3,20	4	5	20
+	23	3	20	23

final answer = 23.

(2) 632 - 5* + 2$3+

Symbol	Stack	op1	op2	value
6	6	-	-	-
3	6, 3	-	-	-
2	6, 3, 2	-	-	-
-	6, 1	3	2	1
5	6, 1, 5	-	-	-
*	6, 5	1	5	5
+	11	6	5	11
2	11, 2	-	-	-
$	121	11	2	121
3	121, 3	-	-	-
+	124 (pop)	121	3	124

final answer = 124.

A C function to evaluate postfix expression.

```
float evaluate_postfix (STACK *s, char postfix[20])
{
    int i,
    char symbol,
    float val, op1, op2, res
```

```
    for (i=0; postfix[i]!='\0'; i++)
    {
        symbol = postfix[i];
        if (isdigit(symbol))
        {   push(s, symbol - '0');
        else if (isalpha(symbo))
        {
            pf("Enter the value for character %c:" symbol);
            sf("%f", &val);
            push(s, val);
        }
        else
        {
            op2= pop(s);
            op1= pop(s);
        res= operate (symbol, op1, op2);
            push(s, res);
        }
    }
    return  pop(s);
}
float  operate (char  s, float op1, float op2)
{
     switch(s)
     {
         case '+': return op1+op2;
         case '-' : return op1-op2;
         case '*' : return op1* op2;
        case '/' : return  op1/ op2;
         case '$' : return  pow(op1,op2);
     }
}
```

convert the following postfix expression to infix.

① 345 * +

symbol	stack	intermediate exp.
3	3	-
4	3,4	-
5	3,4,5	-
*	3, (4*5)	(4*5)
+	3+4*5	3+4*5

② 632 -5 * + 2 $3 +

symbol	stack	iex.
6,3,2.	6,3,2	-
-	6, 3-2	3-2
5	6, 3-2, 5	-
*	6, (3-2)*5	(3-2)*5
+	6+(3-2)*5	6+(3-2)*5
2	6+(3-2)*5, 2	-
$	(6+(3-2)*5)$2	(6+(3-2)*5)$2
3	(6+(3-2)*5)$2, 3	
+	((6+(3-2)*5)$2)+3	

③ abc-+de-fg-h+/*

symbol	stack	i exp.
	a, b, c	-
-	a, (b-c)	b-c
+	(a+(b-c))	a+(b-c)
d, e	(a+(b-c)), d, e	-
-	(a+(b-c)), (d-e)	(d-e)
f, g	(a+(b-c)), (d-e), f, g	-
-	(a+(b-c)), (d-e), (f-g)	(f-g)
h	(a+(b-c)), (d-e), (f-g), h	-
+	(a+(b-c)), (d-e), [(f-g)+h]	~~
/	(a+(b-c)), (d-e)/[(f-g)+h]	~~
*	[a+(b-c)] * [(d-e)/[(f-g)+h]]	~~

Algo to convert infix to prefix

step 1. Scan the expr from right to left

step 2. If scanned symbol is

i) operand - place it of prefix expr.

ii) right parantheses - push it onto stack

iii) If it is left parantheses, pop the content of stack and place them on prefix until you get a corresponding right parantheses

iv) If operator, if stack is empty or top of stack is having right parantheses, push operator onto stack.

else, while stack isn't empty and top of stack is not right paranthesis and precedence of top of stack is greater than or equal to precedence of ~~stacked~~ scanned symbol, pop the stack content and place them on ~~post~~ pre fix and push the operator onto stack.

step3. until stack is empty, pop the stack content and place of prefix.

step4. reverse the prefix to get the actual result

(a+b)*c

symbol	stack	prefix.
c	-	c
*	*	c
)	*)	c
b	*)	cb
+	*)+	cb
a	*)+	cba
(	*	cba+
		cba+*

⇓

*+abc

convert the following prefix to infix.

⑤ - + abc

symbol	stack	intermediate exp.
c	c	-
b	c, b	-
a	c, b, a	-
+	c, (a+b)	(a+b)
-	(a+b)-c	(a+b)-c

⑥ + $ + a * - bcd ef

symbol	stack	intermediate
	f e d c b	-
-	f e d, (b-c)	b-c
*	f e, [(b-c)*d]	(b-c)*d.
a	f e , [(b-c)*d], a	-
+	f e, a+[(b-c)*d]	a+[(b-c)*d]
$	f, (a+((b-c)*d))$e	(a+((b-c)*d))$e
+	[(a+((b-c)*d))$e]+f	