# STACKS and Queues

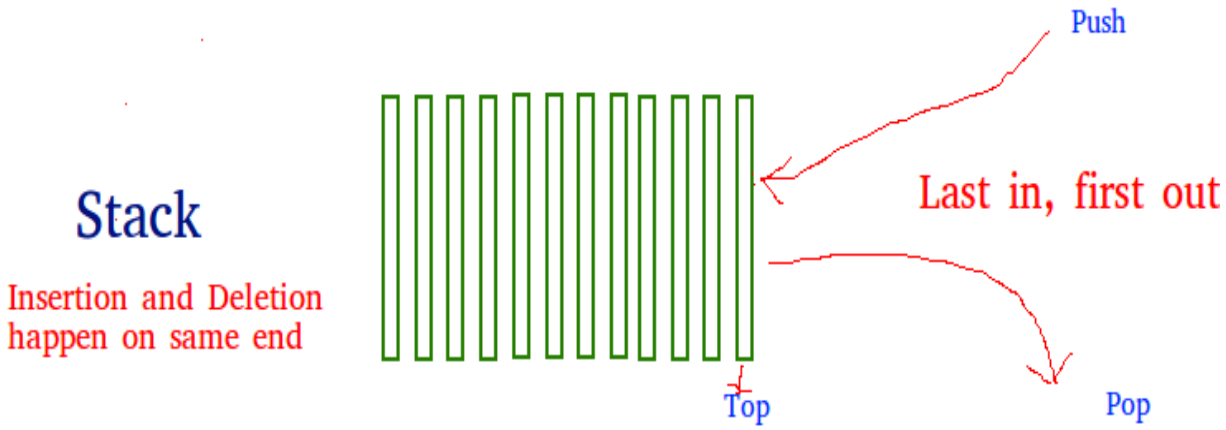
**Unit-II: Stacks and Queues**

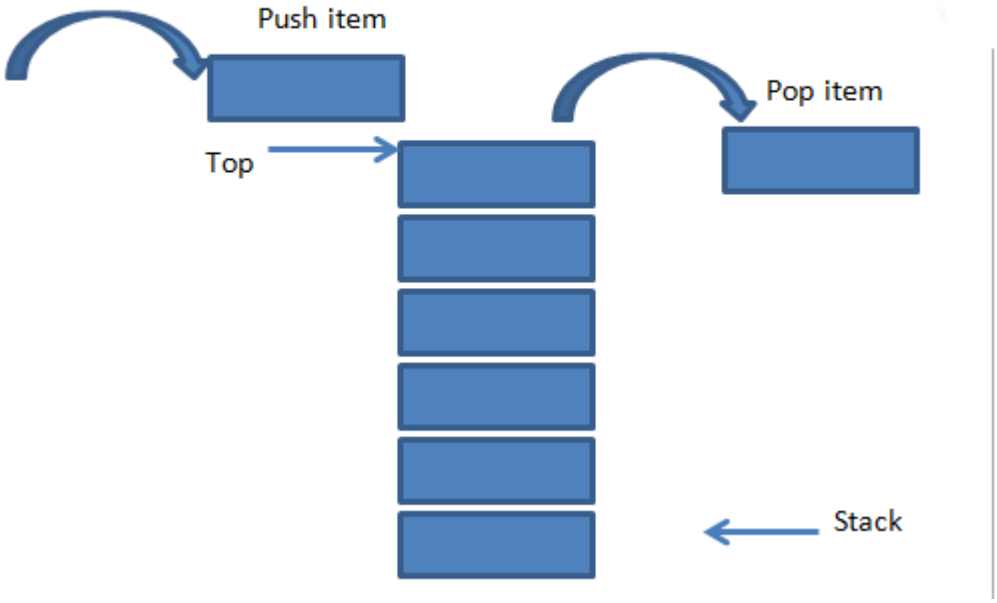
**Stacks:** The Stack: Definition, operations, implementation using arrays, linked list and **Stack applications:** Infix to postfix expression conversion, Evaluation of Postfix expressions, balancing the symbols.**Queue:** definition, operations, implementation using arrays, linked list&it'sApplications.**Circular queue**: definition&its operations, implementation, **Dequeue**: definition & its types, implementation.

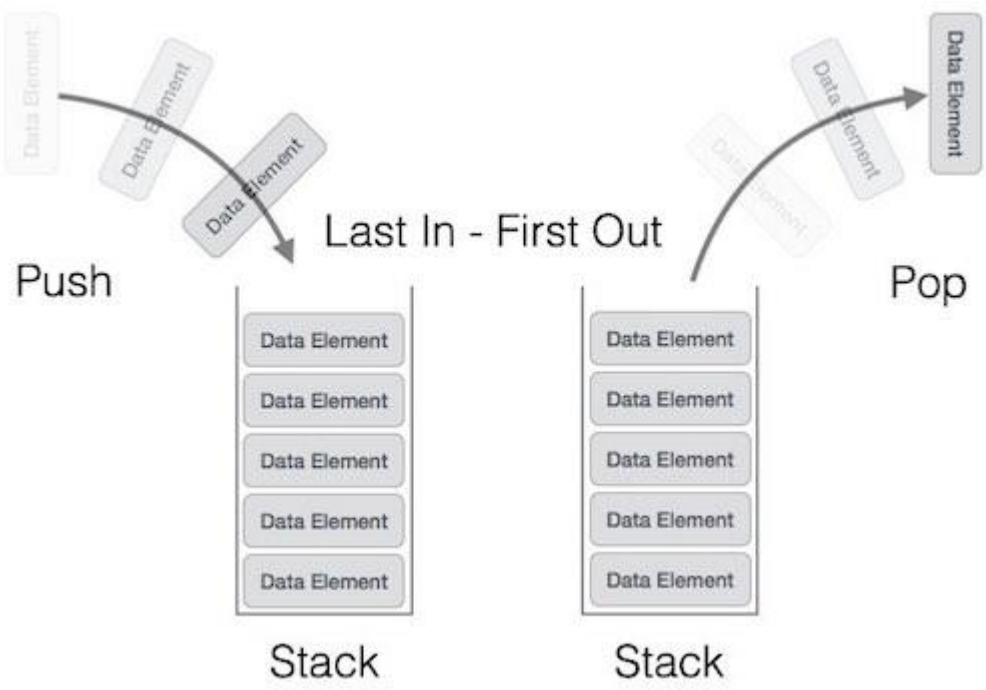
## Learning Material

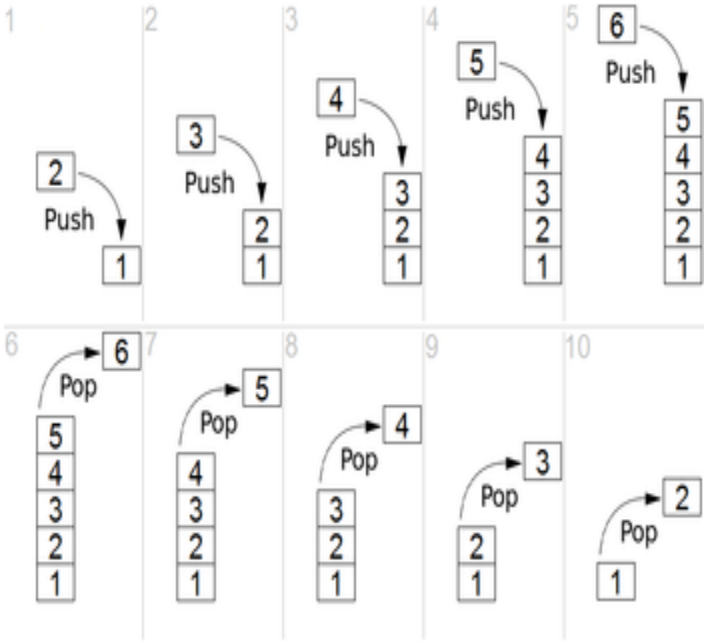
- Stack is a linear data structure.
- Stack can be defined as a collection of homogeneous elements, where insertion and deletion operations takes place at only one end called **TOP**.
- The insertion operation is termed as PUSH and deletion operation is termed as POP operation.
- The PUSH and POP operations are performed at TOP of the stack.
- An element in a stack is termed as ITEM.
- The maximum number of elements that stack can accommodate is termed as SIZE of the stack.
- Stack Pointer ( SP ) always points to the top element of the stack.
- Stack follows LIFO principle. i.e. **L**ast **I**n **F**irst **O**ut i.e. the element which is inserted last into the stack will be deleted first from the stack.

**Diagram of a stack**


Stack
Insertion and Deletion happen on same end
Push
Last in, first out
Top
Pop


Push item
Top
Pop item
Stack




**Stack Operations:-** We can perform mainly 3 operations on stack.

1.Push :- Inserting an element into the stack

2.Pop:- Deleting an element from the stack

3.Peep:- Displaying top most element of the stack

**Representation of stack**

There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

## 1. Array representation of a stack.

First we have to allocate memory for array.

Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.

**Stack Intial Condition:**

Initial condition of stack implemented using arrays is **top=-1**

**Stack Full condition or overflow condition**

Trying to PUSH an item into full stack is known as Stack overflow.

Stack overflow condition is top = = ARRAYSIZE - 1

**Stack underflow or Empty condition**

Trying to POP an item from empty stack is known as Stack underflow. Stack underflow condition is top = = -1

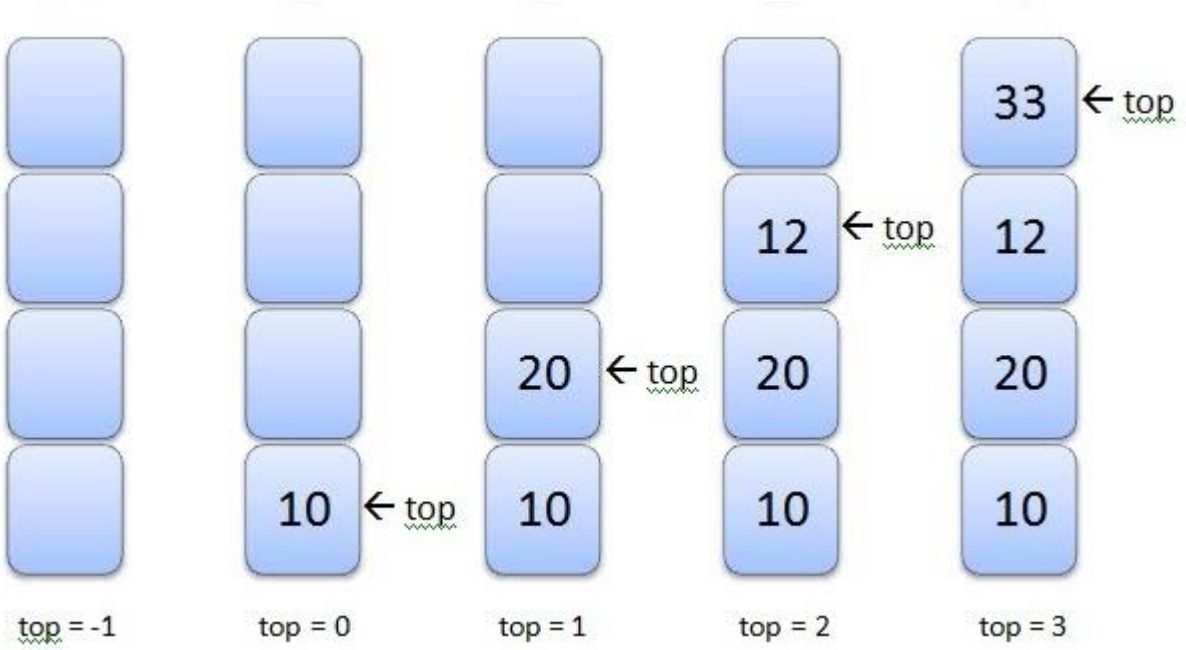
**Algorithm Stack_PUSH(item)**

**Input:** item is new item to push into stack

**Output**: pushing new item into stack at top whenever stack is not full.

1. if(top == ARRAYSIZE-1)
    a) print(stack is full, not possible to perform push operation)
2. else
    a) top=top+1
    b) read item
    c) s[top]=item

**End**
**Stack_PUSH**

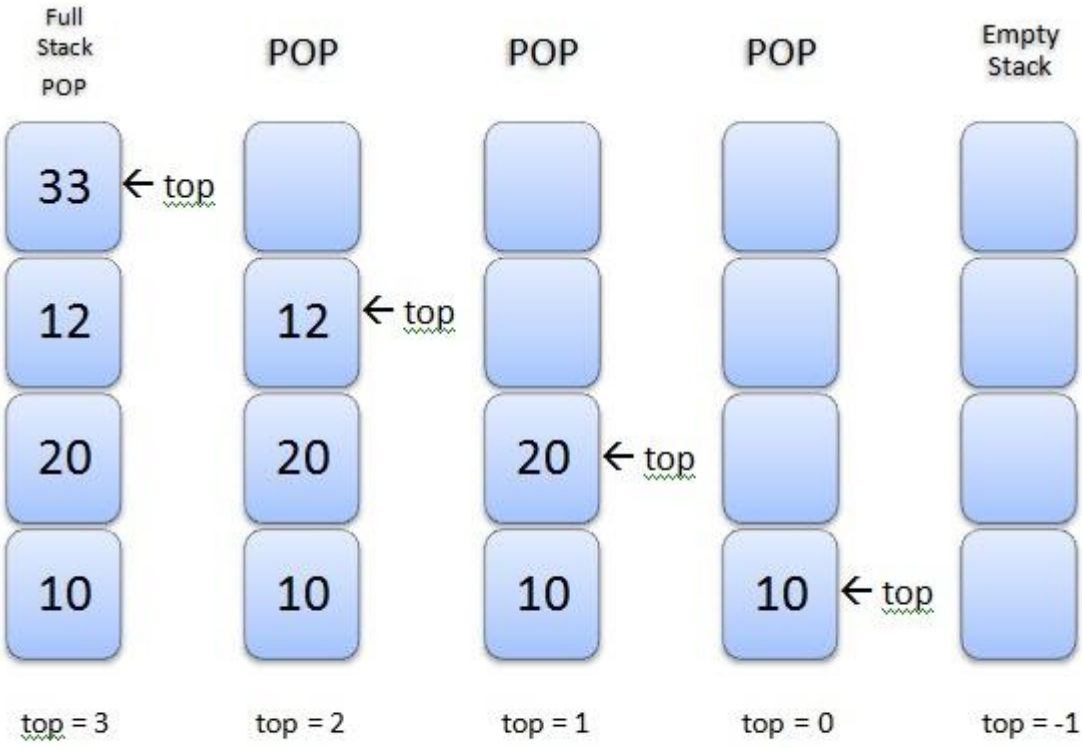



**Algorithm Stack_POP( )**

**Input:** Stack with some elements.

**Output**: item deleted at top most end.

1. if(top ==-1)
    a) print(stack is empty not possible to pop)
2. else
    a) item=s[top]
    b) top=top-1
    c) print(deleted item)




POP operation on Stack with the position of top pointer

Each time an element is popped, then top = top - 1.

**End Stack_POP**

**C Program to implement stack using arrays**

```
#include<stdio.h>
#include<conio.h>
void  push(int ele );
void  pop(  );
void peep(  );
void display(  );
int  stack[10], top= -1, ele;
void main( )
{
        int  choi, ele;
        clrscr( );
        while(1)
        {
                printf(" \t\t Stack ADT Using Arrays\n");
                printf("\n1.Push \n2.Pop \n3.Peep \n4.Display \n5.Exit");
                printf("\nEnter  your choice");
                scanf("%d",&choi);
                switch(choi)
                {
                        case 1: printf("\nEnter the element to be inserted into the stack");
                                scanf("%d",&ele);
                                 push(ele );
                                break;
                        case 2: pop( );
                                break;
                        case 3: peep( );
                                break;
                        case 4: display( );
                                break;
                        case 5: exit(0);
                }
        }
        getch( );
```

```
}

void  push( int ele)
{
        if( top==9)
                printf("\nStack  is full");
        else
        {
                top++;

                stack[top]=ele;
        }
}



void  pop( )
{
        if( top==-1)
                printf("\nStack is empty");
        else
        {
                ele=stack[top];
                printf("\nThe deleted element from the stack is %d",ele);
                top--;
        }
}

void  peep( )
{
        if( top == -1)
                printf("\nStack is empty");
        else
        {
                ele=stack[top];
        printf("\nThe top most element of the stack is %d",ele);
        }
}

void  display( )
{
        int  i;
        if( top==-1)
                printf("\nStack  is empty");
        else
        {
                        printf("\nThe elements of the stack are:\n");
                        for(i=top;i>=0;i--)
                                printf("%d\n",stack[i]);
        }
}
```

**2. Linked List representation of a stack**

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for

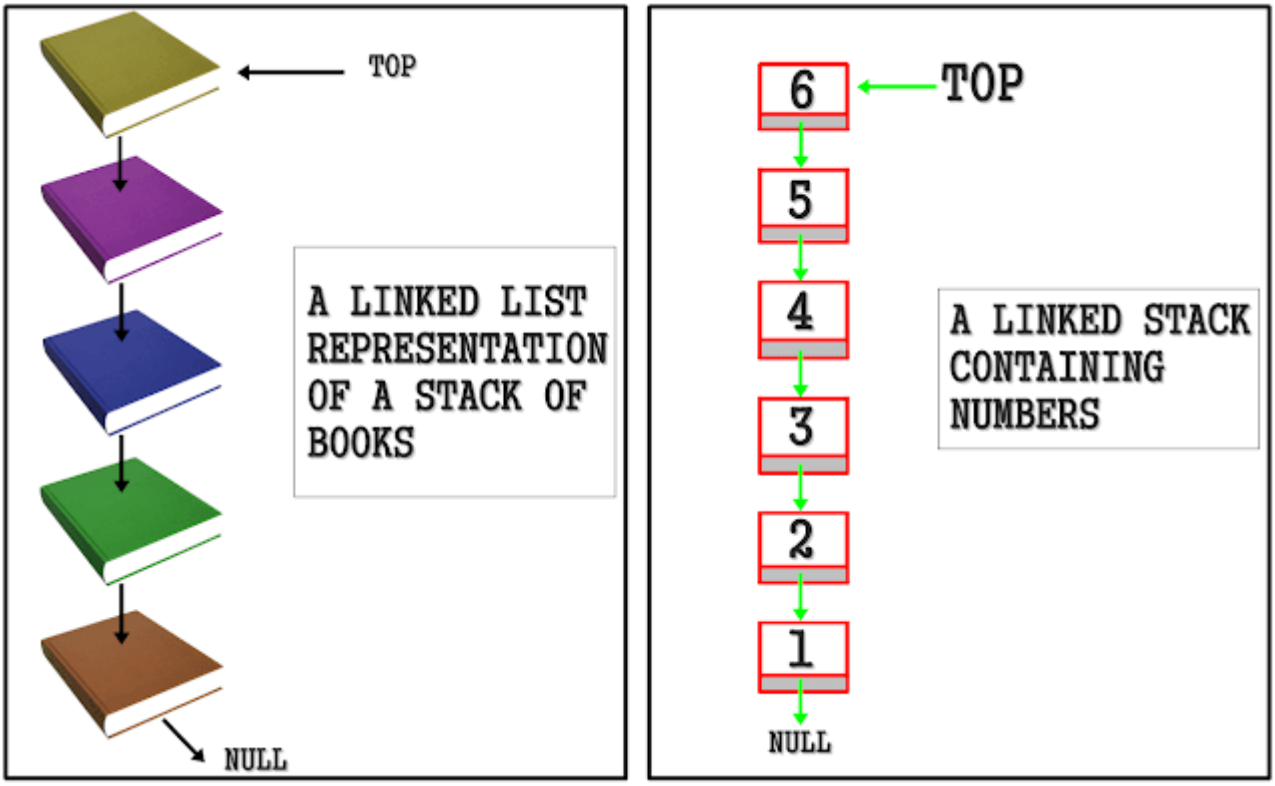
the item and link field points to next node.

- Empty stack condition is

  top = = NULL

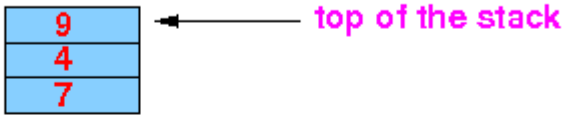
- Full condition is not applicable for Linked List representation of stack. Because here memory is dynamically allocated.
- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

**Linked list representation of stack:-**

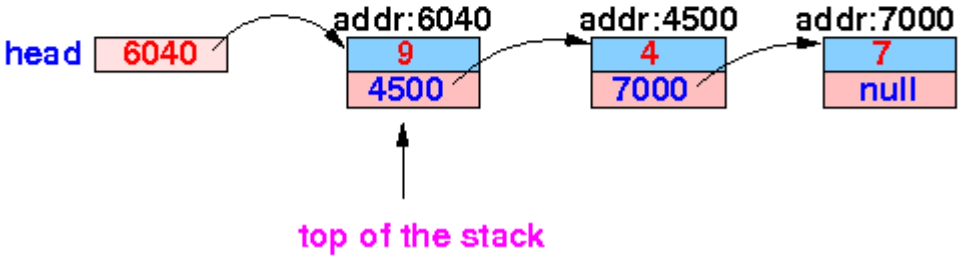


## *Representing a stack with a list:*

**Stack:**




**List:**




**C program to implement stacks using linked list:-**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *top=NULL;
void push(int ele);
void pop();
void display();
void main()
{
        int ele,choice;
        while(1)
        {
                clrscr();
                printf("\t\t Stack ADT using Linked List \n");
                printf("1.push\n");
                printf("2.pop\n");
                printf("3.display\n");
                printf("4.exit\n");
                printf("enter choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:printf("\n enetr ele ");
                                scanf("%d",&ele);
                                push(ele);
```

```
                        break;
                case 2:pop();
                        break;
                case 3:display();
                        break;
                case 4:exit(0);
                }
        }
}
void push(int ele)
{
        struct node *temp;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->info=ele;
        if(top==NULL)
        {
                temp->next=NULL;
                top=temp;

        }
        else
        {
                temp->next=top;
                top=temp;
        }
}
void pop()
{
        struct node *temp;
        if(top==NULL)
        printf("\n stack is  empty");
        else if(top->next==NULL)
        {
                temp=top;
                top=NULL;
                free(temp);
        }
        else
        {
                temp=top;
                top=top->next;
                free(temp);
        }
        getch();
}
void display()
{
        struct node *p;
```

```
if(top==NULL)
printf("\n stack is empy");
else
{
p=top;
while(p!=NULL)
{
        printf("\n elements are %d",p->info);
        p=p->next;
}
}
getch();

}
```

**Arithmetic expressions or Algebraic expressions:-**

An expression is a combination of operands and operators.

Eg. c= a + b

In the above expression a, b, c are operands and +, = are called as operators.

We have 3 notations for the expressions.

i. Infix notation
ii. Prefix notation
iii. Postfix notation

Infix notation: Here operator is present between two operands. eg. a + b

The format for Infix notation as follows

<operand> <operator> <operand>

Prefix notation: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows

<operator> <operand> <operand>

Postfix notation: Here operator is present after two operands.

eg. a b +

The format for Postfix notation as follows

<operand> <operand> <operator>

**Applications of stack**

1. Infix to postfix conversion
2. Evaluation of postfix expression
3. Balancing symbols or delimiter matching

**1. Infix to postfix conversion**

While conversion of infix expression to postfix expression, we must follow the precedence (priority) of the operators.

| **Operator** | **priority** |
|---|---|
| ( | 0 |
| + - | 1 |
| * / % | 2 |
| ^ or $ | 3 |

To convert an infix expression to postfix expression, we can use one stack.
Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

**Algorithm Conversion of infix to postfix**

**Input:** Infix expression.

**Output:** Postfix expression.

1. Perform the following steps while reading of infix expression is not over

a) if symbol is left parenthesis then push symbol into stack.

b) if symbol is operand then add symbol to post fix expression.

c) if symbol is operator then check stack is empty or not.

i) if stack is empty then push the operator into stack.

ii) if stack is not empty then check priority of the operators.

(I) if priority of current operator > priority of operator present at top of stack then push operator into stack.

(II) else if priority of operator present at top of stack >= priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step I)

d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.

2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

**End conversion of infix to postfix**

Contin.....

## Conversion of Infix to Postfix

### Example to Convert Infix to Postfix using stack

a + (b*c)

| Read character | Stack | Output |
|---|---|---|
| a | Empty | a |
| + | + | a |
| ( | +( | a |
| b | +( | ab |
| * | +(* | ab |
| c | +(* | abc |
| ) | + | abc* |
| | | abc*+ |

# Infix to Postfix using stack

- Example A*B+C become AB*C+

| | current symbol | operator stack | postfix string |
|---|---|---|---|
| 1 | A | | A |
| 2 | * | * | A |
| 3 | B | * | A B |
| 4 | + | + | A B * {pop and print the '*' before pushing the '+'} |
| 5 | C | + | A B * C |
| 6 | | | A B * C + |

The infix expression is

$(P/(Q-R)*S+T)$

| Symbol | Stack | Expression |
|---|---|---|
| ( | ( | — |
| *P* | ( | *P* |
| */* | (/ | *P* |
| ( | (/( | *P* |
| *Q* | (/( | *PQ* |
| – | (/(– | *PQ* |
| *R* | (/(– | *PQR* |
| ) | (/ | *PQR–* |
| * | (* | *PQR–/* |
| *S* | (* | *PQR–/S* |
| + | (+ | *PQR–/S** |
| *T* | (+ | *PQR–/S*T* |
| ) | | *PQR–/ST*+* |

So, the postfix expression is *PQR–/ST+**.

| Infix Expression : A+B*(C^D-E) | | | | |
|---|---|---|---|---|
| **Token** | **Action** | **Result** | **Stack** | **Notes** |
| A | Add **A** to the result | A | | |
| + | Push **+** to stack | A | + | |
| B | Add **B** to the result | AB | + | |
| * | Push ***** to stack | AB | * + | * has higher precedence than + |
| ( | Push **(** to stack | AB | ( * + | |
| C | Add **C** to the result | ABC | ( * + | |
| ^ | Push **^** to stack | ABC | ^ ( * + | |
| D | Add **D** to the result | ABCD | ^ ( * + | |
| - | Pop **^** from stack and add to result | ABCD^ | ( * + | - has lower precedence than ^ |
| | Push **-** to stack | ABCD^ | - ( * + | |
| E | Add **E** to the result | ABCD^E | - ( * + | |
| ) | Pop **-** from stack and add to result | ABCD^E- | ( * + | Do process until ( is popped from stack |
| | Pop **(** from stack | ABCD^E- | * + | |
| | Pop ***** from stack and add to result | ABCD^E-* | + | Given expression is iterated, do Process till stack is not Empty, It will give the final result |
| | Pop **+** from stack and add to result | **ABCD^E-*+** | | |
| **Postfix Expression : ABCD^E-*+** | | | | |

| | RPN | Stack | Input Expression |
|---|---|---|---|
| 1 | | | A+(B*(C-D)/E) |
| 2 | A | | +(B*(C-D)/E) |
| 3 | A | + | (B*(C-D)/E) |
| 4 | A | ( + | B*(C-D)/E) |
| 5 | AB | ( + | *(C-D)/E) |
| 6 | AB | * ( + | (C-D)/E) |
| 7 | AB | ( * ( + | C-D)/E) |
| 8 | ABC | ( * ( + | -D)/E) |
| 9 | ABC | ( * ( + | D)/E) |
| 10 | ABCD | ( * ( + | )/E) |
| 11 | ABCD- | * ( + | /E) |
| 12 | ABCD-* | / ( + | E) |
| 13 | ABCD-*E | / ( + | ) |
| 14 | ABCD-*E/ | + | |
| 15 | ABCD-*E/+ | | |

Consider the following arithmetic infix expression P

P=A+(B/C-(D*E^F)+G)*H

| Character scanned | Stack | Postfix Expression (Q) |
|---|---|---|
| A | ( | A |
| + | (+ | A |
| ( | (+( | A |
| B | (+( | AB |
| / | (+(/ | AB |
| C | (+(/ | ABC |
| - | (+(- | ABC/ |
| ( | (+(-( | ABC/ |
| D | (+(-( | ABC/D |
| * | (+(-(* | ABC/D |
| E | (+(-(* | ABC/DE |
| ^ | (+(-(*^ | ABC/DE |
| F | (+(-(*^ | ABC/DEF |
| ) | (+(- | ABC/DEF^* |
| + | (+(+ | ABC/DEF^*- |
| G | (+(+ | ABC/DEF^*-G |
| ) | (+ | ABC/DEF^*-G+ |
| * | (+* | ABC/DEF^*-G+ |
| H | (+* | ABC/DEF^*-G+H |
| ) | | ABC/DEF^*-G+H*+ |

**Write a Turbo C program to convert the Infix expression to Postfix Expression using Stacks implementation of Arrays.**

EXAMPLE 10.6. *Convert X: $A+(B*C-(D/E\uparrow F)*G)*H$ into postfix form showing stack status after every step in tabular form.*

**Solution.**

| Symbol Scanned | | Stack | Expression Y |
|---|---|---|---|
| 1. | A | ( | A |
| 2. | + | ( + | A |
| 3. | ( | ( + ( | A |
| 4. | B | ( + ( | A B |
| 5. | * | ( + ( * | A B |
| 6. | C | ( + ( * | A B C |
| 7. | − | ( + ( − | A B C * |
| 8. | ( | ( + ( − ( | A B C * |
| 9. | D | ( + ( − ( | A B C * D |
| 10. | / | ( + ( − ( / | A B C * D |
| 11. | E | ( + ( − ( / | A B C * D E |
| 12. | ↑ | ( + ( − ( / ↑ | A B C * D E |
| 13. | F | ( + ( − ( / ↑ | A B C * D E F |
| 14. | ) | ( + ( − | A B C * D E F ↑ / |
| 15. | * | ( + ( − * | A B C * D E F ↑ / |
| 16. | G | ( + ( − * | A B C * D E F ↑ / G |
| 17. | ) | ( + | A B C * D E F ↑ / G * − |
| 18. | * | ( + * | A B C * D E F ↑ / G * − |
| 19. | H | ( + * | A B C * D E F ↑ / G * − H |
| 20. | ) | | A B C * D E F ↑ / G * − H * + |

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator

| Symbol | Scanned | STACK | Postfix Expression | Description |
|---|---|---|---|---|
| 1. | | ( | | Start |
| 2. | A | ( | A | |
| 3. | + | (+ | A | |
| 4. | ( | (+( | A | |
| 5. | B | (+( | AB | |
| 6. | * | (+(* | AB | |
| 7. | C | (+(* | ABC | |
| 8. | - | (+(- | ABC* | '*' is at higher precedence than '-' |
| 9. | ( | (+(-( | ABC* | |
| 10. | D | (+(-( | ABC*D | |
| 11. | / | (+(-(/ | ABC*D | |
| 12. | E | (+(-(/ | ABC*DE | |
| 13. | ^ | (+(-(/^ | ABC*DE | |
| 14. | F | (+(-(/^ | ABC*DEF | |
| 15. | ) | (+(- | ABC*DEF^/ | Pop from top on Stack, that's why '^' Come first |
| 16. | * | (+(-* | ABC*DEF^/ | |
| 17. | G | (+(-* | ABC*DEF^/G | |
| 18. | ) | (+ | ABC*DEF^/G*- | Pop from top on Stack, that's why '^' Come first |
| 19. | * | (+* | ABC*DEF^/G*- | |
| 20. | H | (+* | ABC*DEF^/G*-H | |
| 21. | ) | Empty | ABC*DEF^/G*-H*+ | END |

**C Program to convert infix to postfix expression:-**

```
#define SIZE 50          /* Size of Stack */
#include <ctype.h>
char s[SIZE];
int top=-1;       /* Global declarations */
void push(char elem)
{                    /* Function for PUSH operation */
       s[++top]=elem;
}

char pop()
{                    /* Function for POP operation */
```

```
        return(s[top--]);
}

int pr(char elem)
{               /* Function for precedence of operators*/
   switch(elem)
   {
        case '#': return 0;
         case '(': return 1;
         case '+':
        case '-': return 2;
         case '*':
         case '/': return 3;
   }
}

void main()
{                       /* Main Program */
        char infx[50],pofx[50],ch,elem;
        int i=0,k=0;
        printf("\n\nRead the Infix Expression ? ");
        scanf("%s",infx);
        push('#');
        while( (ch=infx[i++]) != '\0')
         {
                if( ch == '(')
                push(ch);
                else if(isalnum(ch))
                        pofx[k++]=ch;
                else if( ch == ')')
                {
                          while( s[top] != '(')
                           pofx[k++]=pop();
                           elem=pop(); /* Remove ( */
                }
                else
                {       /* Operator */
                         while( pr(s[top]) >= pr(ch) )
                        pofx[k++]=pop();
```

```
                push(ch);
            }
        }   //end of while
        while( s[top] != '#')     /* Pop from stack till empty */
            pofx[k++]=pop();
        pofx[k]='\0';          /* Make pofx as valid string */
        printf("\n\nGiven Infix Expn: %s  Postfix Expn:
%s\n",infx,pofx);
}
```

**2.Evaluation of postfix expression**

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

**Algorithm PostfixExpressionEvaluation**

**Input:** Postfix expression

**Output:** Result of Expression

1. Repeat the following steps while reading the postfix expression.

    a) if the read symbol is operand, then push the symbol into stack.

    b) if the read symbol is operator then pop the top most two items of the stack and apply the operator on them, and then push back the result to the stack.

2. Finally stack has only one item, after completion of reading the postfix expression. That item is the result of expression.

**End PostfixExpressionEvaluation**

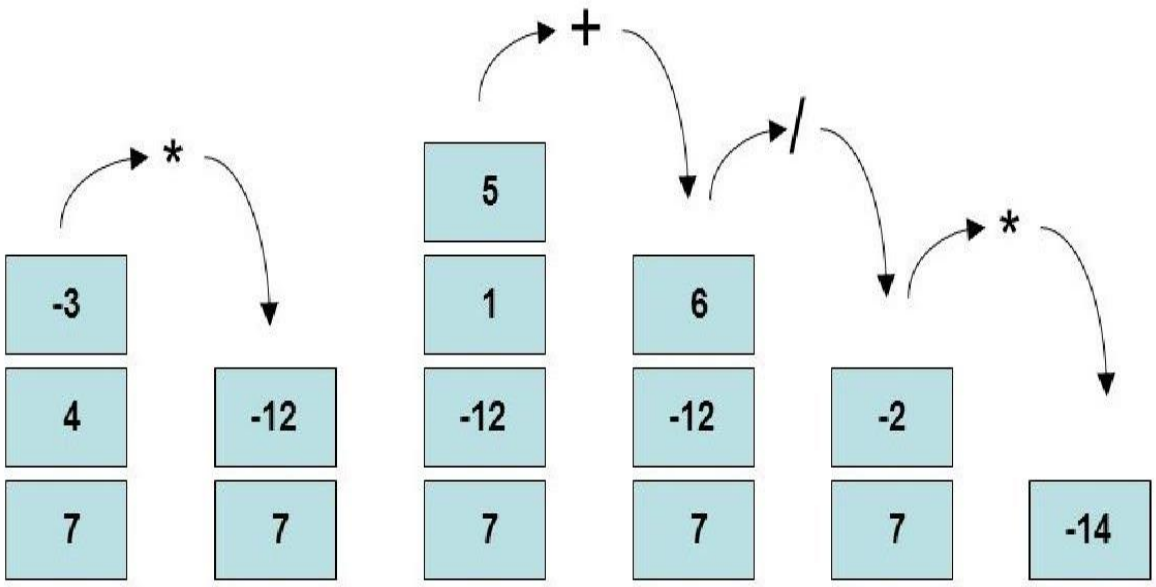
**Example: 4325*-+ = 4+3-2*5= -3**

| Symbol | opnd1 | opnd2 | value | opndstack |
|---|---|---|---|---|
| 4 | | | | 4 |
| 3 | | | | 4, 3 |
| 2 | | | | 4, 3, 2 |
| 5 | | | | 4, 3, 2, 5 |
| * | 2 | 5 | 10 | 4, 3 |
| | | | | 4, 3, 10 |
| - | 3 | 10 | -7 | 4 |
| | | | | 4, -7 |
| + | 4 | -7 | -3 | |
| | | | | -3 |

result

# Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *



# Evaluation of Postfix Expression

Postfix → a b c * + d - Let, a = 4, b = 3, c = 2, d = 5

Postfix → 4 3 2 * + 5 -

| Operator/Operand | Action | Stack |
|---|---|---|
| 4 | Push | 4 |
| 3 | Push | 4, 3 |
| 2 | Push | 4, 3, 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 |

# Evaluating Arithmetic Postfix Expression

PostFix Expression 2 3 4 + * 5 *

| Move | Token | Stack |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 3 | 2 3 |
| 3 | 4 | 2 3 4 |
| 4 | + | 2 7 (3+4=7) |
| 5 | * | 1 4 (2*7=14) |
| 6 | 5 | 14 5 |
| 7 | * | 70 (14*5=70) |

Evaluate below postfix expression 234+*6-

| Step No. | Value of i | Operation | Stack |
|---|---|---|---|
| 1 | 2 | Push 2 in stack | **2** |
| 2 | 3 | Push 3 in stack | **3**<br>**2** |
| 3 | 4 | Push 4 is stack | **4**<br>**3**<br>**2** |
| 4 | + | Pop 2 elements from stack and perform addition operation. And push result back to stack.<br>i.e. 4+3 = 7 | 7<br>**2** |
| 5 | * | Pop 2 elements from stack and perform multiplication operation. And push result back to stack.<br>i.e. 7 * 2 = 14 | **14** |
| 6 | 6 | Push 6 in stack | **6**<br>**14** |
| 7 | - | Pop 2 elements from stack and perform substraction operation. And push result back to stack.<br>i.e. 14 - 6 = 8 | **8** |
| 8 | | Pop result from stack and display | |

**C Program to evaluate postfix expression:-**

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
int st[100],top=-1;
int cal(char post[]);
void push_item(int);
int pop_item();
void main()
{
        char in[50];
        int result;
        clrscr();
        printf("\n \t enter the postfix expression");
        gets(in);
        result=cal(in);
        printf("\n result=%d",result);
        getch();
}
void push_item(int it)
{
        if(top==99)
        printf("stack is overflow\n");
        else
        {
        top++;
        st[top]=it;
        }
}
int pop_item()
{
        int it;
        if(top==-1)
        printf("stack underflow\n");
        else
        {
        return (st[top--]);
        }
}
int cal(char post[])
{
        int m,n,x,y,j=0,len;
        len=strlen(post);
        while(j<len)
        {
                if(isdigit(post[j]))
                {
                x=post[j]-'0';
                push_item(x);
                }
                else
                {
                        m=pop_item();
                        n=pop_item();
                        switch(post[j])
                        {
                        case '+':x=m+n;
                                break;
                        case '-':x=m-n;
                                break;
                        case '*':x=m*n;
                                break;
                        case '/':x=m/n;
                                break;
                        }
                        push_item(x);
                }
```

```
                j++;
 }
                if(top>0)
                {
                printf("no of operands are more than operators");
                exit(0);
                }
                else
                {
                y=pop_item();
                return(y);
                }
}
```

**3.Balancing Symbols or Delimiter matching :-**

The objective of this application is to check the symbols such as parenthesis ( ) , braces { } , brackets [ ] are matched or not.

Thus every left parenthesis, brace and bracket must have its right counterpart.

**Algorithm for Balancing Symbols or Delimiter matching :-**

1. Make an empty stack.
2. Read an expression from left to right.
3. If the reading character is opening symbol, then push it into stack.
4. If the reading character is closing symbol and if the stack is empty, then report as unbalanced expression.
5. If the reading character is closing symbol and if the stack is not empty, then pop the stack.
6. If the symbol popped is not the corresponding opening symbol, then report as unbalanced expression.
7. After processing the entire expression and if the stack is not empty then report as unbalanced expression.
8. After processing the entire expression and if the stack is empty then report as balanced expression.

**C program to implement balancing symbols or delimiter matching:-**

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void push(char);
char  pop();
char  stack[20];
int  top=-1;
main()
{
        char expr[20],ch;
```

```
	int i;
	clrscr();
	printf("\nEnter an expression\n");
	gets(expr);
	for(i=0;expr[i]!='\0';i++)
	{
		ch=expr[i];
		if(ch=='(' || ch=='{' || ch=='[')
			push(ch);
		else if(ch==')')
		{
			if(top==-1)
			{
				printf("\nUnbalanced expression");
				exit();
			}
			else if( (ch=pop())!='(')
			{
				printf("\nUnbalanced expression");
				exit();
			}
		}
		else if(ch=='}')
		{
			if(top==-1)
			{
				printf("\nUnbalanced expression");
				exit();
			}
			else if( (ch=pop())!='{')
			{
				printf("\nUnbalanced expression");
				exit();
			}
		}
		else if(ch==']')
		{
			if(top==-1)
			{
				printf("\nUnbalanced expression");
				exit();
			}
			else if( (ch=pop())!='[')
			{
				printf("\nUnbalanced expression");
				exit();
			}
		}
	}
	if(top==-1)
		printf("\nBalanced expression");
	getch();
}
```

```
void  push(char x)
{
        top++;
        stack[top]=x;
}
int  pop()
{
        return stack[top--];
}
```