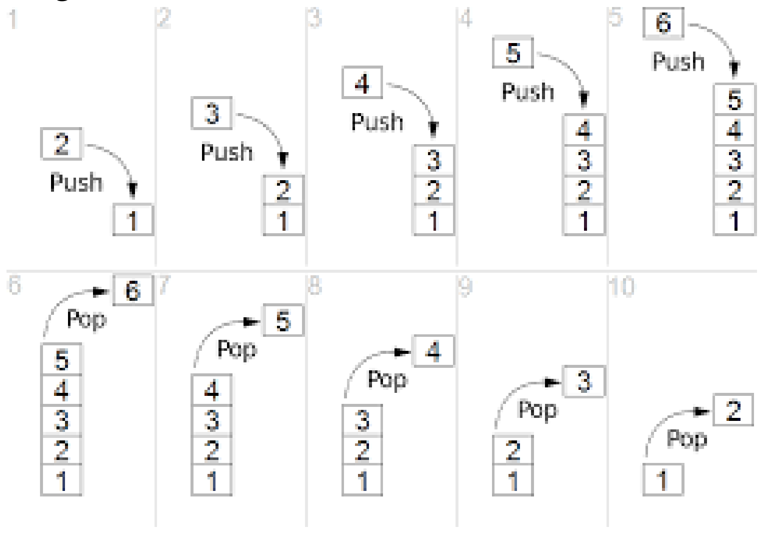
UNIT-II

Stacks

- A stack is an ordered collection of items into which new item may be inserted and from which items may be deleted at one end, called the top of the stack.
- The most important attribute of a stack is the last element inserted into the stack is the first deleted element. So, stack is sometimes called a Last-In-First-Out (LIFO) list.
- When an item is added to a stack, it is called **pushed** into a stack and when an item is removed from the stack, it is called **pop** from the stack.
- Given a stack S and an item i, the operation **push(S, i)** means add the item i to the top of the stack S. the operation **i = pop(S)** means remove the element at the top of stack S and assign its value to i.



Array representation of stack:

A stack is declared as a structure containing two objects

1. An array to hold the elements of the stack.
2. Stack pointer is an integer (top) to indicate the position of the current stack top within the array.

The declaration of the stack is

```
struct stack
{
        int top;
        int items[20];
};
struct stack s;
```

Initially take **s.top = -1.**

Stack operations:

Empty Stack:

The empty stack contains no elements. Therefore top=-1.

The procedure is

```
int empty ( struct stack *s)
{
        if (s→top = = -1)
                return 1;
        else
                return 0;
}
```

Pop operation:

The pop operation performs the following three actions.

1. If the stack is empty, print a warning message and halt the execution.
2. Remove the top element from the stack.
3. Return this element to the calling function.

The procedure is

```
int pop ( struct stack *s)
{
        int a;
        if (s→top = = -1)
                return  -1;
        else
        {
                a = s→items[s→top];
                s→top--;
                return a;
        }
}
```

Push operation:

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if **TOP = MAX – 1,** because in this case the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

The procedure is

```
void push(struct stack *s, int a)
{
        if(s→top = = MAX-1)
                printf("\nStack is OVERFLOW\n");
        else
        {
                s→top++;
                s→items[s→top] = a;
        }
}
```

Peek operation:

Peek is an operation that return the value of the topmost element of the stack without deleting it from the stack.

The procedure is

```
int peek(struct stack *s)
{
        int a;
        if (s→top = = -1)
                return -1;
        else

        {
                a = s→items[s→top];
                return a;
        }
}
```

Program stack operations using array

```
/*stack operations using arrays*/
#define max 10
#include<stdio.h>
struct stack
{
        int top;
        int items[max];
};
void push(struct stack *, int a);
int pop(struct stack *);
int empty(struct stack *);
int peek(struct stack *);
void display(struct stack *);
main()
{
        int choice,x;
        struct stack s;
        s.top=-1;
        while(1)
        {
        printf("\n\n****************************\n\n");
        printf("\t\tMENU\n");
        printf("\n****************************\n");
        printf("1.push\n2.pop\n3.empty\n4.peek\n5.display\n6.exit\n");
        printf("Enter your choice:");
        scanf("%d",&choice);
        switch(choice)
        {
                case 1:printf("\nEnter the element:");
                        scanf("%d",&x);
                        push(&s,x);
                        break;
                case 2: x=pop(&s);
                        if(x==-1)
                                printf("\nstack is underflow");
                        else
                                printf("\ndeleted element is %d",x);
                        break;
                case 3: x=empty(&s);
                        if(x==1)
                                printf("\nstack is empty");
                        else
                                printf("\nstack is not empty");
                        break;
                case 4: x=peek(&s);
                        if(x==-1)
                                printf("\nStack is empty");
                        else
                                printf("\nTop of stack element:%d",x);
                        break;
                case 5: printf("\nElements are\n");
                        display(&s);
                        break;
                case 6: exit(0);
        }
        }
}
```

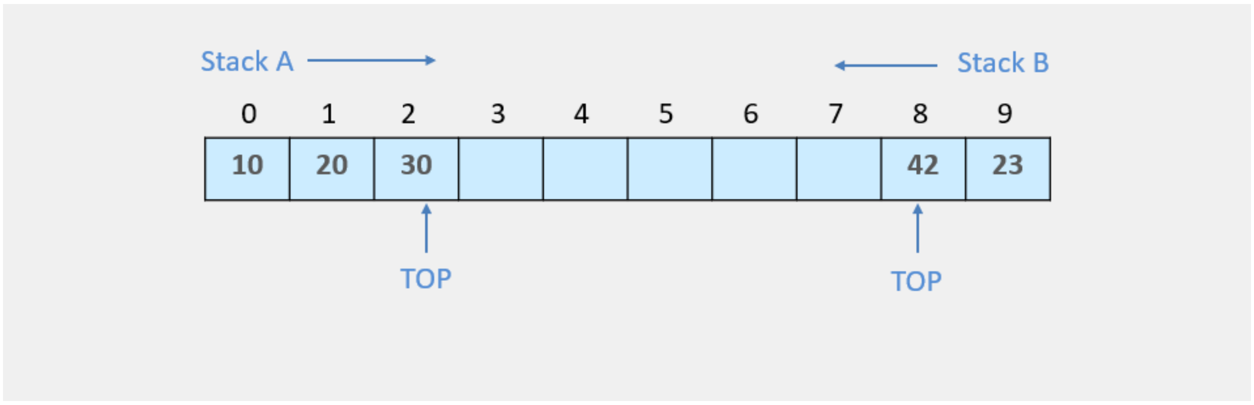
```
void push(struct stack *st,int a)
{
        if(st->top==max-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top++;
                st->items[st->top]=a;
        }
}
int pop(struct stack *st)
{
        int a;
        if(empty(st))
                return -1;
        else
        {
                a=st->items[st->top];
                st->top--;
                return a;
        }
}
void display(struct stack *st)
{
        int i;
        for(i=0;i<=st->top;i++)
                printf("%d\t",st->items[i]);
}
int empty(struct  stack *st)
{
        if(st->top==-1)
                return 1;
        else
                return 0;
}
int peek(struct stack *st)
{
        if(st->top==-1)
                return -1;
        else
```

```
                return st->items[st->top];
}
```

Multiple stacks:

We maintain two stacks in the same array. An array of size ‘n’ is used to represent two stacks **stack1** and **stack2**. The value of n is such that the combined size of both stacks will never exceed n. While operating on these stacks, it is important to stack1 will grow from left to right, whereas stack2 will grow from right to left at the same time.



Extending this concept to multiple stacks, a stack can also be used to represent n number of stacks in the same array.

/*Multiple stack operations program*/

```
/*Multiple stack operations*/
#define max 10
#include<stdio.h>
struct stack
{
        int top1,top2;
        int items[max];
};
void push1(struct stack *,int);
void push2(struct stack *,int);
int pop1(struct stack *);
int pop2(struct stack *);
int empty1(struct stack *);
int empty2(struct stack *);
void display1(struct stack *);
void display2(struct stack *);
int peek1(struct stack *);
int peek2(struct stack *);
```

```
main()
{
        int choice,x,ch;
        struct stack s;
        s.top1=-1;
        s.top2=max;
        while(1)
        {
        printf("\n\n****************************\n\n");
        printf("\t\tMENU\n");
        printf("\n****************************\n");
        printf("1.push\n2.pop\n3.display\n4.empty\n5.peek\n6.exit\n");
        printf("Enter your choice:");
        scanf("%d",&choice);
        switch(choice)
        {
                case 1:printf("\nEnter the element:");
                        scanf("%d",&x);
                        printf("pushed in\n1.stack-1\n2.stack-2\n");
                        printf("Enter your choice:");
                        scanf("%d",&ch);
                        if(ch==1)
                                push1(&s,x);
                        else if(ch==2)
                                push2(&s,x);
                        else
                                printf("\nwrong choice");
                        break;
                case 2: printf("popped in\n1.stack-1\n2.stack-2\n");
                        printf("Enter your choice:");
                        scanf("%d",&ch);
                        if(ch==1)
                                x=pop1(&s);
                        else if(ch==2)
                                x=pop2(&s);
                        else
                                printf("\nwrong choice");
                        if(x==-1)
                                printf("\nstack is underflow");
                        else
                                printf("\ndeleted element is %d",x);
                        break;
                case 4: printf("Check empty in\n1.stack-1\n2.stack-2\n");
```

```
                        printf("Enter your choice:");
                        scanf("%d",&ch);
                        if(ch==1)
                                x=empty1(&s);
                        else if(ch==2)
                                x=empty2(&s);
                        else
                                printf("\nwrong choice");

                        if(x==1)
                                printf("\nstack is empty");
                        else
                                printf("\nstack is not empty");
                        break;
                case 3: printf("Display in\n1.stack-1\n2.stack-2\n");
                        printf("Enter your choice:");
                        scanf("%d",&ch);
                        printf("\nElements are\n");
                        if(ch==1)
                                display1(&s);
                        else if(ch==2)
                                display2(&s);
                        else
                                printf("\nwrong choice");
                        break;
                case 5: printf("Peek in\n1.stack-1\n2.stack-2\n");
                        printf("Enter your choice:");
                        scanf("%d",&ch);
                        if(ch==1)
                                x=peek1(&s);
                        else if(ch==2)
                                x=peek2(&s);
                        else
                                printf("\nwrong choice");
                        if(x==-1)
                                printf("\nStak is empty");
                        else
                                printf("\nTopmost element of stack:%d",x);
                        break;
                case 6: exit(0);
        }
        }
}
```

```
void push1(struct stack *st,int a)
{
        if(st->top1==st->top2-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top1++;
                st->items[st->top1]=a;
        }
}
void push2(struct stack *st,int a)
{
        if(st->top1==st->top2-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top2--;
                st->items[st->top2]=a;
        }
}
int pop1(struct stack *st)
{
        int a;
        if(st->top1==-1)
                return -1;
        else
        {
                a=st->items[st->top1];
                st->top1--;
                return a;
        }
}
int pop2(struct stack *st)
{
        int a;
        if(st->top2==max)
                return -1;
```

```
	else
	{
		a=st->items[st->top2];
		st->top2++;
		return a;
	}
}
void display1(struct stack *st)
{
	int i;
	for(i=0;i<=st->top1;i++)
		printf("%d\t",st->items[i]);
}
void display2(struct stack *st)
{
	int i;
	for(i=max-1;i>=st->top2;i--)
		printf("%d\t",st->items[i]);
}
int empty1(struct  stack *st)
{
	if(st->top1==-1)
		return 1;
	else
		return 0;
}
int empty2(struct  stack *st)
{
	if(st->top2==max)
		return 1;
	else
		return 0;
}
int peek1(struct stack *st)
{
	int a;
	if(st->top1==-1)
		return -1;
	else
	{
		a=st->items[st->top1];
		return a;
	}
```

```
}
int peek2(struct stack *st)
{
        int a;
        if(st->top2==max)
                return -1;
        else
        {
                a=st->items[st->top2];
                return a;
        }
}
```

Applications of stack:

- Reversing a list
- Infix to Postfix Conversion
- Infix to Prefix Conversion
- Evaluating postfix Expression
- Parathesis matching
- Factorial calculation

Reversing a list:

A list of numbers can be reversed by reading each number from array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

/* Reversing the List Using Stack Program */

```
#define max 50
#include<stdio.h>
#include<conio.h>
struct stack
{
        int top;
        int items[max];
};
void push(struct stack *,int);
int pop(struct stack *);
int empty(struct stack *);
void main()
{
        int a[max],n,i;
        struct stack s;
        s.top=-1;
        clrscr();
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements in array\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\nArray Elements are\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        for(i=0;i<n;i++)
                push(&s,a[i]);
        i=0;
        while(!empty(&s))
        {
                a[i]=pop(&s);
                i++;
        }
        printf("\nReverse list is\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}
void push(struct stack *s, int a)
{
        if(s->top==max-1)
                printf("\nStack is FULL\n");
        else
        {
                s->top++;
                s->items[s->top]=a;
        }
}
int pop(struct stack *s)
{
        int a;
        if(s->top==-1)
                return -1;
        else
        {
                a=s->items[s->top];
                s->top--;
                return a;
        }
```

```
}
int empty(struct stack *s)
{
        if(s->top==-1)
                return 1;
        else
                return 0;
}
```

Infix to Postfix:

Consider the sum of A & B.
A + B, where A and B are operands and + is an operator.
A + B is called Infix Expression
AB+ is called Postfix Expression
+AB is called Prefix Expression
In infix expression may contain parentheses, operands, and operators. We use the operators addition(+), subtraction(-), multiplication(*), division(/), exponentiation($). The precedence's of operators are

High priority $ (right to left)
Next priority *, / (left to right)
Low priority +, - (left to right)

Example:
Convert the following infix expression to postfix expression.
A * (B + C)

Solution:
A * (B + C) → A * (BC+) (parenthesis)
→ ABC+* (multiplication)

Example:
Convert the following infix expression to postfix expression.
A * B + C

Solution:
A * B + C → (AB*) + C (multiplication)
→ AB*C+ (addition)

Example:
Convert the following infix expression to prefix expression.
A * (B + C)

Solution:
A * (B + C) → A * (+BC) (parenthesis)
→ *A+BC (multiplication)

Example:
Convert the following infix expression to prefix expression.

A * B + C

Solution:

A * B + C → (*AB) + C (multiplication)

→ +*ABC (addition)

Infix	Postfix	Prefix
A + B	AB+	+AB
A + B – C	AB+C-	-+ABC
(A + B) * (C-D)	AB+CD-*	*+AB-CD
A $ B * C – D + E / F / (G + H)	AB$C*D-EF/GH+/+	+-*$ABCD//EF+GH
((A + B) * C – (D - E)) $ (F + G)	AB+C*DE—FG+$	$-*+ABC-DE+FG
A – B / (C * D $ E)	ABCDE$*/-	-A/B*C$DE

Conversion of infix to postfix expression:

- Operator precedence play an important role in transforming infix expression to postfix expression.
- If we read an open parenthesis, then it must be pushed into the stack.
- If we read a closed parenthesis, all operators up to the first open parenthesis must be popped from the stack and add to the postfix string.
- If we read an operand, then the operand must be add to the postfix string.
- If we read an operator with precedence is less than or equal to the precedence of currently top of the stack operator, then the operator must be popped from the stack and add to the postfix string until reading operator precedence grater than precedence of top of the stack operator or empty stack.
- If we read an operator with precedence is grater than to the precedence of currently top of the stack operator, then the operator must be pushed into the stack.

Algorithm:

```
opndstk = the empty stack;
while(not end of input)
{
        symb = next input character;
        if(symb is an operand)
                add symb to the postfix string;
        else if(symb is an open parenthesis)
                push(opensdk, symb);
        else if(symb is a closed parenthesis)
        {
                topsymb = pop(opndstk);
                while(topsymb is not an open parenthesis)
                {
                    add topsymb to the postfix string;
                    topsymb = pop(opndstk);
                }
        }
        else
        {
                while(!empty(opndstk) && (prcd(symb) <= prcd(peek(opndstk))))
                {
                    topsymb = pop(opndstk);
                    add topsymb to the postfix string;
                }
                push(opndstk,symb);
        }
}
while(!empty(opndstk))
{
        topsymb = pop(opndstk);
        add topsymb to the postfix string;
}
```

Example:

convert the following infix expression into postfix expression using algorithm.

((A - (B + C)) * D) $ (E + F)

Symb	Postfix string	Opndstk
(	--	(
(	---	((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	--
$	ABC+-D*	$

(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
	ABC+-D*EF+$	--

Example:
convert the following infix expression into postfix expression using algorithm.

A $ B * C – D + E / F / (G + H)

Symb	Postfix string	Opndstk
A	A	--
$	A	$
B	AB	$
*	AB$	*
C	AB$C	*
-	AB$C*	-
D	AB$C*D	-
+	AB$C*D-	+
E	AB$C*D-E	+
/	AB$C*D-E	+/
F	AB$C*D-EF	+/
/	AB$C*D-EF/	+/
(	AB$C*D-EF/	+/(
G	AB$C*D-EF/G	+/(
+	AB$C*D-EF/G	+/(+
H	AB$C*D-EF/GH	+/(+
)	AB$C*D-EF/GH+	+/
	AB$C*D-EF/GH+/	+
	AB$C*D-EF/GH+/+	---

/*Infix to postfix Program*/

```
#define MAX 50
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
struct stack
{
        int top;
```

```
            char items[MAX];
};
struct stack s;
char postfix[MAX];
int isp(char);
int icp(char);
void push(struct stack *,char);
char pop(struct stack *);
int empty(struct stack *);
void  intopost(char []);
int isoper(char);
char peek(struct stack *);
int main()
{

            char infix[MAX];
            clrscr();
            s.top=-1;
            printf("enter the infix expression");
            scanf("%s",infix);
            printf("\nthe original infix expression is %s",infix);
            intopost(infix);
            getch();
            return 0;
 }

void  intopost(char expr[])
{
            int i,j=0;
            char symb,topsymb;

            for(i=0;(symb=expr[i])!='\0';i++)

                        if(isoper(symb))
                                    postfix[j++]=symb;
                        else if(symb=='(')
                                    push(&s,symb);
                        else if(symb==')')
                        {       topsymb=pop(&s);
                                    while((!empty(&s))&&(topsymb!='('))
                                    {
                                                postfix[j++]=topsymb;
                                                topsymb=pop(&s);
                        }
                }
                else
                {
                        while(!empty(&s)&&icp(symb)<=isp(peek(&s)))
                        {
                                topsymb=pop(&s);
                                postfix[j++]=topsymb;
                        }
                        push(&s,symb);
                }
                while(!empty(&s))
                {
                        topsymb=pop(&s);
                        postfix[j++]=topsymb;
                }
                postfix[j]='\0';
                printf("\npostfix expression is: %s",postfix);


}
int isoper(char symb)
{
        if((symb>='0'&&symb<='9')||(symb>='a'&&symb<='z')||(symb>'A'&&symb<='Z'))
                return 1;
        else
                return 0;
}
void push(struct stack *st,char a)
{
        if(st->top==MAX-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top++;
                st->items[st->top]=a;
        }

}
```

```
char pop(struct stack *st)
{
        char a;
        if(empty(st))
                return -1;
        else
        {
                a=st->items[st->top];
                st->top--;
                return a;
        }
}
int empty(struct  stack *st)
{
        if(st->top==-1)
                return 1;
        else
                return 0;
}
char peek(struct stack *st)
{
        if(empty(&s))
                return -1;
        else
                return st->items[st->top];
}
int isp(char c)
{
        switch(c)
        {
                case '(':return 0;
                case '+':
                case '-':return 1;
                case '*':
                case '/':return 2;
                case '$':return 3;

        }
}
int icp(char c)
{
        switch(c)
        {
                case '+':
                case '-':return 1;
                case '*':
                case '/':return 2;
                case '$':return 3;
        }
}
```

Conversion of Infix to prefix expression:

Convert the given infix expression into prefix expression using the following algorithm

Step-1: Reverse the infix expression. Note that while reversing the string you must interchange left and right parenthesis.

Step-2: convert the resulting infix expression into postfix expression.

Step-3: Reverse the resulting postfix expression to get prefix expression.

/*Infix to prefix Program*/

```
#define MAX 50
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
struct stack
{
        int top;
        char items[MAX];
};
struct stack s;
char postfix[MAX],prefix[MAX];
int isp(char);
int icp(char);
void push(struct stack *,char);
char pop(struct stack *);
int empty(struct stack *);
void  intopost(char []);
int isoper(char);
char peek(struct stack *);
void main()
{

        char c,infix[MAX],infix1[MAX];
        int i;
        clrscr();
        s.top=-1;
        printf("enter the infix expression");
        scanf("%s",infix);
        for(i=0;(c=infix[i])!='\0';i++)
        {
                if(c==')')
                        push(&s,'(');
                else if(c=='(')
                        push(&s,')');
                else
                        push(&s,c);
        }
        i=0;
        while(!empty(&s))
        {
                c=pop(&s);
                infix1[i]=c;
                i++;
        }
        infix1[i]='\0';
        printf("reverse expression is:%s",infix1);
        intopost(infix1);
        getch();
 }

void  intopost(char expr[])
{
        int i,j=0;
        char symb,topsymb;

        for(i=0;(symb=expr[i])!='\0';i++)

                if(isoper(symb))
                        postfix[j++]=symb;
                else if(symb=='(')
                        push(&s,symb);
                else if(symb==')')
                {      topsymb=pop(&s);
                        while((!empty(&s))&&(topsymb!='('))
                        {
                                postfix[j++]=topsymb;
                                topsymb=pop(&s);
                        }
                }
                else
                {
                        while(!empty(&s)&&icp(symb)<isp(peek(&s)))
                        {
                                topsymb=pop(&s);
                                postfix[j++]=topsymb;
                        }
                        push(&s,symb);
                }
                while(!empty(&s))
                {
                        topsymb=pop(&s);
                        postfix[j++]=topsymb;
                }
                postfix[j]='\0';
                printf("\npostfix expression is:%s",postfix);
                for(i=0;(symb=postfix[i])!='\0';i++)
                                push(&s,symb);
                i=0;
                while(!empty(&s))
                {
                        topsymb=pop(&s);
                        prefix[i]=topsymb;
                        i++;
                }
                prefix[i]='\0';
                printf("\nprefix expression is: %s",prefix);


}
int isoper(char symb)
{
        if((symb>='0'&&symb<='9')||(symb>='a'&&symb<='z')||(symb>'A'&&symb<='Z'))
                return 1;
        else
                return 0;
}
void push(struct stack *st,char a)
{
        if(st->top==MAX-1)
        {
                printf("\nstack is overflow");
```

```
	}
	else
	{
		st->top++;
		st->items[st->top]=a;
	}

}

char pop(struct stack *st)
{
	int a;
	if(empty(st))
		return -1;
	else
	{
		a=st->items[st->top];
		st->top--;
		return a;
	}
}
int empty(struct  stack *st)
{
	if(st->top==-1)
		return 1;
	else
		return 0;
}
int isp(char c)
{
	switch(c)
	{
		case '(':return 0;
		case '+':
		case '-':return 1;
		case '*':
		case '/':return 2;
		case '$':return 3;

	}
}
char peek(struct stack *st)
{
        if(empty(&s))
                return -1;
        else
                return st->items[st->top];
}
int icp(char c)
{
        switch(c)
        {
                case '+':
                case '-':return 1;
                case '*':
                case '/':return 2;
                case '$':return 3;
        }
}
```

Evaluation of a postfix expression:

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Algorithm:

```
opndstk = the empty stack;
while(not end of input)
{
        symb = next input character;
        if(symb is an operand)
                push(opndstk, symb);
        else
        {
                opnd2 = pop(opndstk);
                opnd1 = pop(opndstk);
                value = result of applying symb to opnd1 and opnd2;
                push(opndstk, value);
        }
}
return(pop(opndstk));
\
```

Example:

Evaluate the following postfix expression

6 2 3 + - 3 8 2 / + * 2 $ 3 +

symb	Opnd1	Opnd2	value	opndstk
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3		52

/*Postfix Evaluation program*/

```
#define MAX 20
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
struct stack
{
        int top;
        double items[MAX];
};
struct stack s;
double oper();
void push(struct stack *,double);
double pop(struct stack *);
int empty(struct stack *);
double eval(char []);
int isdigit(char);
main()
{
        char expr[MAX];
        clrscr();
        s.top=-1;
        printf("enter the postfix expression");
        scanf("%s",expr);
        printf("\nthe original postfix expression is %s",expr);
        printf("\n%lf",eval(expr));
 }

double eval(char expr[])
{
        int i;
        char c;
        double opnd1,opnd2,value;

        for(i=0;(c=expr[i])!='\0';i++)

                if(isdigit(c))
                        push(&s,(double)c-'0');
                else
                {
                        opnd2=pop(&s);
                        opnd1=pop(&s);
                        value=oper(c,opnd1,opnd2);
                        push(&s,value);

                }
        return(pop(&s));
}
int isdigit(char symb)
{
        return(symb>='0'&&symb<='9');
}
double oper(char symb,double op1,double op2)
{
        switch(symb)
        {
                case '+': return(op1+op2);

                case '-': return(op1-op2);

                case '*': return(op1*op2);
                case '/': return(op1/op2);

                case '$': return(pow(op1,op2));

                default: printf("\nillegal operation");
                        exit(0);
        }
}
void push(struct stack *st, double a)
{
        if(st->top==MAX-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top++;
                st->items[st->top]=a;
        }
}
double pop(struct stack *st)
{
        double a;
        if(empty(st))
                return -1;
        else
        {
                a=st->items[st->top];
                st->top--;
                return a;
        }
}
int empty(struct  stack *st)
{
        if(st->top==-1)
                return 1;
        else
                return 0;
}
```

Parenthesis Matching:
Stack can be used to check the validity of parenthesis in any algebraic expression. An algebraic expression is valid if for every open bracket there is a corresponding closed bracket.
Example:

1. (A + B} is not valid expression
2. {(A + B) * C} is valid expression

/*paranthesis matching Program*/

```
#define max 20
#include<stdio.h>
struct stack
{
        int top;
        char items[max];
};
struct stack s;
void push(struct stack*,char);
char pop(struct stack*);
int empty(struct stack*);
void eval(char []);
void main()
{

        char infix[max];
        clrscr();
        s.top=-1;
        printf("enter the infix expression");
        scanf("%s",infix);
        eval(infix);
        getch();
 }

void eval(char infix[])
{
        char ch,c;
        int i,valid=1;
        for(i=0;(ch=infix[i])!='\0';i++)
        {
                if(ch=='('||ch=='['||ch=='{')
                push(&s,ch);
                if(ch==')'||ch==']'||ch=='}')
```

```
                        if(empty(&s))
                          valid=0;
                else
                {
                        c=pop(&s);
                        if(((ch==')') &&(c!='('))||((ch==']')&&(c!='['))||((ch=='}')&&(c!='{')))
                                valid=0;
                }
        }
        if(!empty(&s))
        valid=0;
        if(valid)
            printf("the paranthesis are matched");
        else
            printf("paranthesis are not matched");
}


void push(struct stack *st,char a)
{
        if(st->top==max-1)
        {
                printf("\nstack is overflow");

        }
        else
        {
                st->top++;
                st->items[st->top]=a;
        }
}
char pop(struct stack *st)
{
        char a;
        if(empty(st))
                return -1;
        else
        {
                a=st->items[st->top];
                st->top--;
                return a;
        }
}
```

```
int empty(struct  stack *st)
{
        if(st->top==-1)
                return 1;
        else
                return 0;
}
```

Factorial calculation:

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.

Every recursive solution has two major cases.

1. Base case: in which the problem is simple to solved directly without making any further calls to the same function.
2. Recursive case: in which first the problem is divide into smaller subparts. Second the function calls itself but with subparts of the problem obtained in the first step. Third, the result is obtained by coming the solutions of simpler subparts.

Example: factorial calculation.

n! = n * (n-1) * (n-2) * …….*2*1

5! = 5*4*3*2*1 = 120

This can be written as

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1 = 120

```
#include<stdio.h>
int fact(int n);
main()
{
        int n, val;
        printf(“\nenter the number:”);
        scanf(“%d”, &n);
        val = fact(n);
        printf(“\nFactorial of %d = %d”,n,val);
}
int fact(int n)
{
        if(n= =1)
                return 1;
        else
                return n*fact(n-1);
}
```