Data Structure: Data Structure is a collection of organized data that are related to each other.
Data Structures can be classified into two types.

1. Linear Data Structure
2. Non-Linear Data Structure

Linear Data Structure:

A data structure is linear if every item is related (or attached) to its previous and next item. Linear Data Items are arranged in a linear sequence.

Linear Data Structures include Arrays and lists.

Non-Linear Data Structure:

A Data Structure is non linear if every item is attached to many other items in specific ways to reflect relationships. In non-linear data structure data items are not in a sequence.

Non-Linear Data Structures include trees and graphs.

STACKS

Definitions:

1. A Stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called top of the stack.
2. A stack is a linear Data Structure in which a data item is inserted and deleted at one end.

A stack is called LIFO (Last In First Out) structure because the data item that is inserted last into the stack is the first data item to be deleted from the stack.
A single end of the stack is designated as the stack top. New items may be put on the top of the stack(in which case the top of the stack moves upward to correspond to the new highest element or items which are at the top of the stack may be removed (in which case the top of the stack moves downward to correspond to the new highest element.
The two basic operations of stack is

PUSH()- To push an element into the stack
POP()- To return an element from the stack.

PUSH()

PUSH() operation checks the stack there is still some room left in the stack, and if there is any, It adds the received item to the stack .

```
If(top==MAXSIZE)
    Display ‘ Stack Overflows’
Else
    Insert an element into stack.
```

POP()

POP() operation checks the stack whether it contains any element, if So, it returns an element from the stack and decrements stack top.

```
If(top==-1)
   Display “ No elements in the stack”
Else
```

Return element from the stack top;

An Example representation of stack:

Operation	**Content of the stack after the operation**
PUSH(A)	A
PUSH(B)	AB
POP()	A
PUSH(C)	AC
PUSH(D)	ACD
PUSH(E)	ACDE
POP()	ACD
POP()	AC
POP()	A
POP()	Stack empty

Applications of stack:

1. Stacks are extensively used in Computer Applications mostly in system software(such as compilers, operating systems etc).

- For example, when one function in C calls another function, and passes some parameters, these parameters are passed using stack.
- Many compilers stores the local variables inside a function on the stack.
- Stacks are used in recursions.

1. Stack operations are implemented in machines to do some basic tasks, such as evaluating expressions, handling dynamic memory allocation etc.

They are mainly used in converting

- an infix expression to postfix expression.
- an Infix expression to prefix expression
- Postfix expression evaluation.
-Prefix expression evaluation

Template based Program on stacks using Arrays.

```
#include<iostream>
using namespace std;
#include<conio.h>
#define MAX 20
template<class T>
class stack
 {
```

```
   public:
   T s[100];
   int top;
   T x;
   public:stack()
            {
              top=-1;
            }
          void push(T);
          T pop(void);
          void display(void);
};
template<class T>
void stack<T>::push(T x)
 {
   if(top==MAX-1)
     cout<<"\nStack full";
   else
      {
        s[++top]=x;
      }
 }
template<class T>
T stack<T>::pop(void)
{

   if(top==-1)
   {
      cout<<"\nStack Underflow";
   }
    else
      return(s[top--]);
}
template<class T>
void stack<T>::display(void)
 {
    if(top==-1)
      cout<<"\nNo elements in the stack";
    else
      {
        for(int i=top;i>=0;i--)
          cout<<s[i]<<"\t";
      }
 }
main()
  {
       stack <char> s;
       int ch;
```

```
    cout<<"\n1.PUSH";
    cout<<"\n2.POP";
    cout<<"\n3.DISPLAY";
    cout<<"\n4.EXIT";
    cout<<"\nEnter choice";
    cin>>ch;
    while(1)
     {
      switch(ch)
      {
       case 1: cout<<"\nEnter element to be pushed";
               cin>>s.x;
               s.push(s.x);
               break;
       case 2:
               if(s.top==-1)
                 cout<<"\nStack Underflow";
               else
                 cout<<"\nPopped element is"<<s.pop();
               break;
       case 3: s.display();
               break;
       case 4: exit(0);
       default:cout<<"\nInvalid choice";
      }
     cout<<"\nEnter choice";
     cin>>ch;
    }
    getch();
}
```

APPLICATIONS OF STACKS

In the arithmetic expression, a+b*c, the addition operation is not evaluated first. This is because, operators are evaluated in the order of their precedence in the expression. So, the entire equation is examined to determine whether there is any operator with higher precedence. This tracing is best implemented with stack operations. The various stack-oriented notations are:

- Infix
- Prefix
- Postfix

Infix notation: An ordinary mathematical expression is called infix notation. When using infix notation, the operators are placed between the operands in an expression.
Ex: (a/b)+(c*d)-e
Postfix notation: The operators are placed after the operands, i.e., the operators are preceded by

the operands. This is also known as reverse polish notation (RPN).
Ex: ab+cd-*
Prefix notation: The operators are placed before the operands in a mathematical expression. This notation is also called as Polish notation.
Ex:*+ab-cd

Advantages:

1.The advantage of using prefix and postfix notations is that the use of parenthesis and operator precedence rules is avoided completely.
2. Any complex Arithmetic Expressions can be evaluated easily.

Conversion from Infix to postfix notation

The following algorithm is used to convert an infix expression to an equivalent postfix expression:

1. First, initialize the stack to be empty.
2. For each character in the input string,

- If the input string is an operand,

  append to the output.
- Else If the input string is a left parenthesis,

  push it onto the stack
- Else If stack empty or the operator has higher priority than the operator on the top of the stack or the top of the stack is opening parenthesis

  Then
  Push the operator on to the stack
  Else
  Pop the operator from the stack and append to the output.
- Else If the input string is a closing parenthesis, pop operator from the stack and append the operators to the output until an opening parenthesis is encountered. Pop the opening parenthesis from the stack and discard it.
- I If the end of the input string is encountered, then iterate the loop until the stack is not empty.
- Pop the stack and append the remaining input string to the output.

EXAMPLE: Consider the conversion of the infix expression. a*b/(c-d)+e*(f-g), to its equivalent postfix notation:

Input string	Stack operation	Postfix Notation
A	Empty	a
*	*	a
B	*	ab
/	/	ab*
(	/(	ab*
C	/(	ab*c
-	/(-	ab*c
D	/(-	ab*cd
)	/	ab*cd-
+	+	ab*cd-/
E	+	ab*cd-/e
*	+*	ab*cd-/e
(	+*(	ab*cd-/e
F	+*(	ab*cd-/ef
-	+*(-	ab*cd-/ef
G	+*(-	ab*cd-/efg
)	Empty	ab*cd-/efg-*+

Examples:

Infix	**Postfix**
(a+b)cd*e-f/g	ab+cde*fg/-
(a+b)*(c-d)	ab+cd-*
(a+b)$c-(d*e)/f	ab+c$de*f/-
a+b-c	ab+c-
((a+b)*c-(d-e))$(f+g)	ab+c*de—fg+$
a-b/(c*d$e)	abcde$*/-
a$b*c-d+e/f/(g+h)	ab$c*d-ef/gh+/+

Program to convert infix to postfix

```
#include<iostream>
using namespace std;
#include<conio.h>
class stackapp
 {
    public:
       char s[100];
       char exp[100];
       int top;
```

```
        stackapp()
          {
           top=-1;
          }
        void push(char);
        char pop(void);
        int insop(char);
        int ineop(char);
        char stacktop(void);
};
void stackapp::push(char x)
 {
    s[++top]=x;
 }
char stackapp::pop(void)
 {
    return(s[top--]);
 }
char stackapp::stacktop(void)
 {
   return(s[top]);
 }
int stackapp::insop(char c)
 {
   if(c=='+' || c=='-')
     return 1;
   else if(c=='*' || c=='/' || c=='%')
     return 2;
   else if(c=='(')
     return 0;
   else if(c=='#')
     return -1;
   else if(c=='$')
   return 3;
}

int stackapp::ineop(char c)
 {
   if(c=='+' || c=='-')
     return 1;
   else if(c=='*' || c=='/' || c=='%')
     return 2;
   else if(c=='(')
     return 0;
   else if(c=='$')
```

```
    return 3;
 }
main()
 {
    stackapp s;
    int i;
    cout<<"\nEnter infix expression";
    cin>>s.exp;
    i=0;
    while(s.exp[i]!='\0')
      {
        if((s.exp[i]>='a' || s.exp[i]>='A') && (s.exp[i]<='z' || s.exp[i]<='Z'))
            cout<<s.exp[i];
        else if((s.exp[i]=='+' || s.exp[i]=='-' || s.exp[i]=='*' || s.exp[i]=='/' || s.exp[i]=='%' ||
           s.exp[i]=='$')  && (s.top==-1))
            s.push(s.exp[i]);
         else if(s.exp[i]=='(')
           s.push(s.exp[i]);
         else if(s.exp[i]==')')
          {
             while((s.stacktop())!='('
               cout<<s.pop();
             s.pop();
          }
        else if((s.insop(s.stacktop()))<(s.ineop(s.exp[i])))
          s.push(s.exp[i]);
        else if((s.insop(s.stacktop()))>=(s.ineop(s.exp[i])))
          {
              while((s.insop(s.stacktop()))>=(s.ineop(s.exp[i]))&& s.top>=0)
                 cout<<s.pop();
              s.push(s.exp[i]);
          }
        i++;
      }
      while(s.top>=0)
        {
           cout<<s.pop();
        }

     getch();

}
```

Conversion from infix to prefix notation

The following algorithm is used to convert an infix expression to prefix expression.

Algorithm

- First, initialize the stack to be empty and reverse the given input string.
- For each character in the input string

  - If the input string is a right parenthesis, push it onto the stack
  - Else If the input string is an operand, append to the output.
  - Else If the stack is empty or the operator has higher or equal priority than the operator on the top of the stack or the top of the stack is a right parenthesis. Then
    Push the operator onto the stack
    Else
    Pop the operator from the stack and append to the output.
  - If the input string is a left parenthesis, pop operators from the stack and append all the operators to the output until the right parenthesis is encountered. Pop the right parenthesis from the stack and discard it.

- If the end of the input string is encountered, then iterate the loop until the stack is not empty Pop the stack, and append the remaining input string to the output and reverse the output string.

Consider the conversion of the infix expression, a*b/(c-d)+e*(f-g), to its equivalent prefix notation. First, to convert the infix expression to the prefix notation, reverse the given input string as follows:
)g-f(*e+)d-c(/b*a

Input string	Stack operation	Prefix Notation
)	)	
G	)	g
-	)-	g
F	)-	gf
(	Empty	gf-
*	*	gf-
E	*	gf-e
+	+	gf-e*
)	+)	gf-e*
D	+)	gf-e*d
-	+)-	gf-e*d

C	+	gf-e*dc
(	+	gf-e*dc-
/	+/	gf-e*dc-
B	+/	gf-e*dc-b
*	+/*	gf-e*dc-b
A	Empty	gf-e*dc-ba*/+

Now, reverse the output string, gf-e*dc-ba*/+ as +/*ab-cd*e-fg to obtain the prefix notation.

Examples:

Infix	**Prefix**
(a+b)cd*e-f/g	-*$$+abcde/fg
(a+b)*(c-d)	*+ab-cd
(a+b)$c-(d*e)/f	-$+abc/*def
a+b-c	-+abc
((a+b)*c-(d-e))$(f+g)	$-*+abc-de+fg
a-b/(c*d$e)	-a/b*c$de
a$b*c-d+e/f/(g+h)	+-*$abcd//ef+gh

Program to convert infix to prefix

```
#include<iostream>
using namespace std;
#include<conio.h>
class stackapp
 {
    public:
       char s[100];
       char exp[100];
       int top;
       stackapp()
          {
           top=-1;
          }
       void push(char);
       char pop(void);
       int insop(char);
       int ineop(char);
       char stacktop(void);
 };
```

```
void stackapp::push(char x)
 {
    s[++top]=x;
 }
char stackapp::pop(void)
 {
    return(s[top--]);
 }
char stackapp::stacktop(void)
 {
   return(s[top]);
 }
int stackapp::insop(char c)
 {
   if(c=='+' || c=='-')
     return 1;
   else if(c=='*' || c=='/' || c=='%')
     return 2;
   else if(c==')')
     return 0;
   else if(c=='#')
     return -1;
   else if(c=='$')
   return 3;
  }
int stackapp::ineop(char c)
 {
   if(c=='+' || c=='-')
     return 1;
   else if(c=='*' || c=='/' || c=='%')
     return 2;
   else if(c=='(')
     return 0;
   else if(c=='$')
     return 3;
 }
main()
 {
     stackapp s;
     int i;
     int j=0;
     char result[20];
     cout<<"\nEnter infix expression";
     cin>>s.exp;
     strrev(s.exp);
```

```
i=0;
s.push('#');
while(s.exp[i]!='\0')
  {
     if((s.exp[i]>='a' || s.exp[i]>='A') && (s.exp[i]<='z' || s.exp[i]<='Z'))
         {
          result[j]=s.exp[i];j++;
         }
      else if(s.exp[i]==')')
         s.push(s.exp[i]);
     else if(s.exp[i]=='(')
       {
          while((s.stacktop())!=')')
            {
              result[j]=s.pop();j++;
            }
          s.pop();
       }
     else if((s.insop(s.stacktop()))<=(s.ineop(s.exp[i])))
       s.push(s.exp[i]);
     else if((s.insop(s.stacktop()))>(s.ineop(s.exp[i])))
       {
          while((s.insop(s.stacktop()))>(s.ineop(s.exp[i])))
             {
               result[j]=s.pop();j++;
             }
          s.push(s.exp[i]);
       }
     i++;
  }
  while(s.top>0)
    {
       result[j]=s.pop();
       j++;
    }
   result[j]='\0';
 cout<<"\nResult"<<strrev(result);
 getch();

}
```

Evaluation of postfix expression

Stacks are used to evaluate the postfix expression. To evaluate the postfix expression, consider the following steps:

- Scan the expression from left to right.

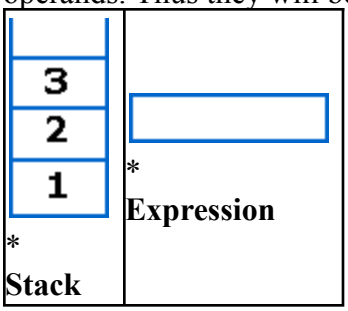
- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack

Example:

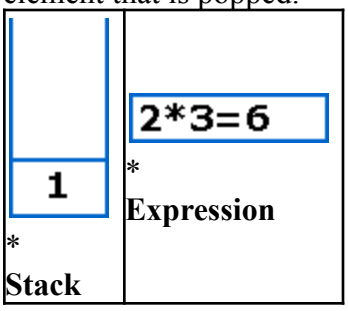
Let us see how the above algorithm will be implemented using an example.

Postfix String : 123*+4-

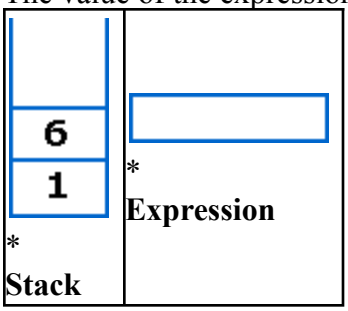
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.



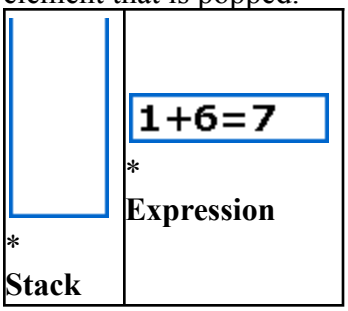
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



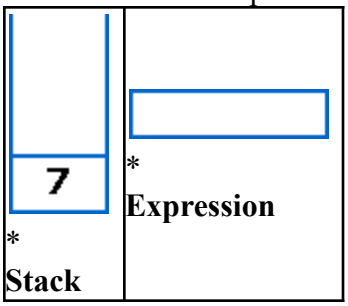
The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.



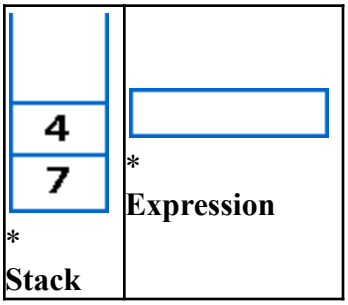
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



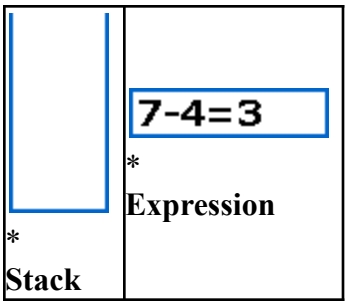
The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.



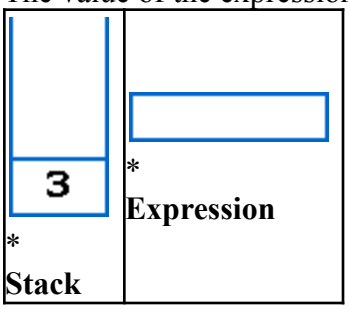
Next character scanned is "4", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3

Program on postfix evaluation

```
#include<iostream>
using namespace std;
#include<conio.h>
#include<math.h>
class posteval
 {
    public:
          int s[100];
          char exp[100];
          int top;
          void push(int);
          int pop(void);
          posteval()
            {
                  top=-1;
```

```
            }
};
void posteval::push(int c)
 {
   s[++top]=c;
 }
int posteval::pop(void)
 {
    return(s[top--]);
 }
main()
 {
    posteval p;
    int i=0;
    int x1,x2;
    cout<<"\nEnter postfix expression";
    cin>>p.exp;
    while(p.exp[i]!='\0')
      {
        if(p.exp[i]>='0' && p.exp[i]<='9')
          p.push(p.exp[i]-'0');
        else if(p.exp[i]=='+')
          {
            x1=p.pop();
            x2=p.pop();
            p.push(x1+x2);
           }
        else if(p.exp[i]=='-')
          {
            x1=p.pop();
            x2=p.pop();
            p.push(x2-x1);
          }
        else if(p.exp[i]=='*')
          {
            x1=p.pop();
            x2=p.pop();
            p.push(x1*x2);
          }
        else if(p.exp[i]=='/')
          {
            x1=p.pop();
            x2=p.pop();
            p.push(x2/x1);
          }
```

```
            else if(p.exp[i]=='%')
              {
                 x1=p.pop();
                 x2=p.pop();
                 p.push(x2%x1);
              }
               else if(p.exp[i]=='$')
              {
                 x1=p.pop();
                 x2=p.pop();
                 p.push(int(pow(x2,x1)));
              }
            i++;
        }
        while(p.top>=0)
          {
            cout<<p.pop();
          }
          getch();
}
```

Evaluation of prefix Expression:

- Scan the expression from right to left.
- If the input string is an operand, then push it onto the stack.
- If the input string is an operator, then the first two operands on the stack are evaluated using this operator by popping them from the stack and the result is also placed onto the stack