Chapter 2
The Stack

1. INTRODUCTION

Suppose you placed some books one above another in a table. It forms a stack of books. If you want to pick up a book from the stack of books, then the book you placed at last is the book you can pick up first from the top of the books. In addition, if you want to place a new book in the stack, you can place it only on the top of the stack. In real World, there are many situations for example pile of plates in which objects are placed and removed from one end only as in the example of stacks of books. When you implement this real World scenario to store the data in computer, it is called as a stack data structure. The important features of a stack are 1) it is ordered list and 2) the access to it is restricted to only one end.

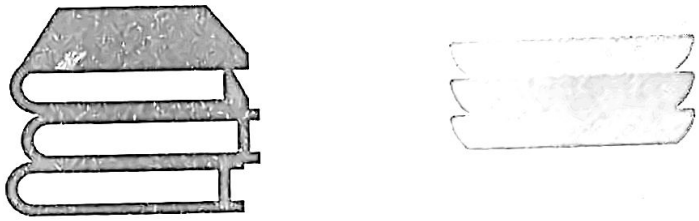

Fig. 2.1: A stack of books and a pile of plates

A stack is a linear data structure. It is an ordered list of items into which new items can be inserted and from which items can be deleted. The insertion and deletion of items are made at only one end called top of the stack. The last item you insert into the stack is the first item to delete from the stack. Therefore, stack is also called as last-in first-out (LIFO) list. Such LIFO list is named as a stack since it is manipulated and accessed as a real-world stack like a deck of cards or a pile of plates.

2. PRIMITIVE OPERATIONS

In a stack of objects, the basic operations that can be done include insertion of objects into the stack and deletion of objects from the stack. In addition, you can also do other operations to check whether the stack is empty or it is full. The insertion of a data item into a stack is called Push operation and deletion of the data item from the stack is called Pop operation. These both operations in stack are performed at only one end. Therefore, the item that is pushed last is the first item to be popped out.

The Fig. 2.2 illustrates the push and pop operations that can be performed with items in stack. The Fig. 2.2 (a) shows a stack with no items. It is called empty stack. Items L, A, P, E and N are pushed into the stack in Fig. 2.2 (b) to (f). Then, items are popped from the top of the stack from Fig. 2.2 (g) to (k). Notice that the items are popped from the stack are in reverse order. Also note that the item N is inserted first into the stack and it is the last item which is popped out from the stack.

You cannot insert an item into a stack which is already full. Similarly, you cannot delete an item from the stack which is already empty. The result of an illegal attempt to push an item into a full stack is called overflow and the result of an attempt to pop or access an item from an empty stack is called underflow. Therefore, to avoid such illegal attempts, you have to check whether a stack is full or empty before you insert an item into the stack or remove an item from the stack. However, if a stack is implemented using a linked list, the overflow may not occur since there will not be the size limit as in implementation using an array.

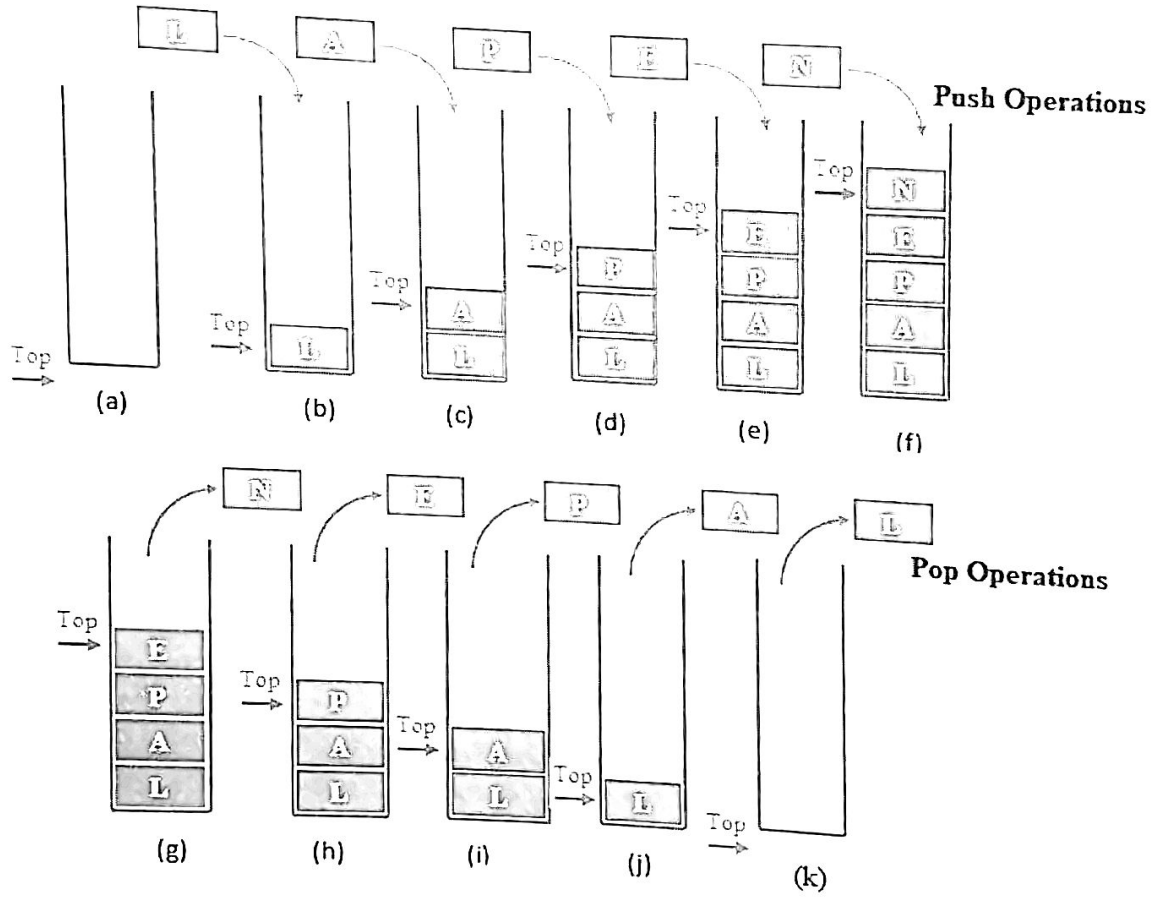

Fig. 2.2: Illustration of Inserting and deleting elements into/from a stack

Example: Stack is used to maintain browser history. Go to a browser and open a page. Again open a new page from some link in the page. Go on doing so. You are creating a history of pages you have visited. The history is maintained in stack. Every time you open a new page, you are pushing the link of the page in a stack. Now, clack on the back icon, you will be drive to the last page you opened. Here, you are doing a pop operation. Now click on the forward button, it represents the push operation.

3. STACK AS AN ABSTRACT DATA TYPE (ADT)

A stack is an abstract data type. The stack ADT is an ordered collection of objects/items/data and a set of operations such as push and pop that can operate on that ordered collection of objects/items/data. Using the concept of class or structure, a stack ADT can be defined or declared.

a) Stack ADT

A stack ADT can be represented in template class as:

```
template <class T>
class Stack
{
//objects: a finite ordered list with zero or more elements.
public:
        Stack(int MaxSize);
                //create an empty stack whose maximum size is MaxSize
        Boolean IsFull();
                //if (number of elements in stack == MaxStackSize)
                //    return TRUE
                //else  return FALSE
        void push(T item);
                //if (IsFull(stack)) then StackFull()
                //else insert item into top of stack
        Boolean IsEmpty();
                //if number of elements in stack is 0, return TRUE
                //else return FALSE
        T pop();
                //if (IsEmpty()), then return 0
                //else remove and return the top item of the stack
        ~Stack();
                // default destructor
};
```

b) Implementation of Stack Operations

Stack(int MaxSize)

```
template <class T>
stack<T>::stack(int maxsize){
      size = maxsize;
      STACK = new T[size];
      top = -1;
}
```

IsFull()

```
template <class T>
bool stack<T>::full()
{
      if(top == size-1)
            return true;
      else
            return false;
}
```

sEmpty()

```
template <class T>
bool stack<T>::empty()
{
      if(top == -1)
            return true;
      else
            return false;
}
```

push(const T &*item*);

```
template <class T>
void stack<T>::push(const T &item)
{
      if(full())
      {
            cout<<"Stack overflow.";
      }
      else
      {
            top++;
            STACK[top] = item;
      }
}
```

T pop()

```
template <class T>
T stack<T>::pop()
{
      T item;
      if(empty())
      {
            cout<<"Stack Underflow";
            return NULL;
      }
      else
      {
            item = STACK[top];
            top--;
            return item;
      }
}
```

An abstract data type (ADT) is a type or class for objects whose behavior is defined by a set of value and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

c) Implementation of Stack Using C++

The implementation of operations can be done separately using simple array or link list data structure.

(i) Implementation of Stack ADT Using Array

The following program illustrate the implementation of Stack ADT using class template.

```
#include<iostream>
#include<conio.h>
using namespace std;

template <class T>
class stack
{
      int size, top;
      T *STACK;
      public:
            stack(int);
            bool full();
            bool empty();
            T pop();
            void push(const T &item);
            void display();
            ~stack() {delete [] STACK;}
};
template <class T>
stack<T>::stack(int maxsize){
      size = maxsize;
      STACK = new T[size];
      top = -1;
}
template <class T>
bool stack<T>::full()
{
      if(top == size-1)
            return true;
      else
            return false;
}
template <class T>
bool stack<T>::empty()
{
      if(top == -1)
            return true;
      else
```

```
            return false;
}
template <class T>
T stack<T>::pop()
{
      T item;
      if(empty())
      {
            cout<<"Stack Underflow";
            return NULL;
      }
      else
      {
            item = STACK[top];
            top--;
            return item;
      }
}
template <class T>
void stack<T>::push(const T &item)
{
      if(full())
      {
            cout<<"Stack overflow.";
      }
      else
      {
            top++;
            STACK[top] = item;
      }
}
template <class T>
void stack<T>::display()
{
      int i;
      for(i=top; i>=0; i--)
      {
            if(!empty() && i==top)
                  cout<<"top -> "<<STACK[i]<<endl;
            else
                  cout<<"       "<<STACK[i]<<endl;
      }
}
int main()
{
      int n, size;
      int item;

      cout<<"\nEnter Stack Size:";
      cin>>size;

      stack<int> s(size);

       while(1)
        {
        cout<<"MENU- STACK OPERATIONS"<<endl<<endl;
        cout<<"1. PUSH an ITEM"<<endl;
        cout<<"2. POP an ITEM"<<endl;
```

```
        cout<<"3. Exit"<<endl;

        cout<<endl<<"_____________________________";
        cout<<endl<<"Stack [Size: "<<size<<"]"<<endl;
        s.display();
        cout<<"_____________________________"<<endl;

        cout<<endl<<endl<<"Choose one of the above option. [1-4]: ";
        cin>>n;

            switch (n)
            {
            case 1:
          cout<<endl<<"Enter an ITEM:";
          cin>>item;
          s.push(item);
          break;
             case 2:
          item = s.pop();
          if(item != NULL)
             cout<<endl<<"Popped ITEM from stack: "<<item;
          break;
             case 3:
          exit(0);
             default:
          cout<<endl<<"Enter correct option!Try again.";
              }

          cout<<endl<<endl<<"--";
          cout<<endl<<endl<<"Continue? Enter any key... ";
          getch();
          system("cls");
          }
        return 0;
}
```

(ii) Implementation of Stack ADT Using Linked List

The following program illustrate the implementation of Stack ADT using linked list.

```
#include<iostream>
#include<conio.h>
using namespace std;

class node
{
  public:
       int info;
       node *link;
};

node *top = NULL;
node *tempNode;

node *createNode(int item)
```

```
    {
          tempNode = new node;
          tempNode->info = item;
          tempNode->link = NULL;
          return (tempNode);
    }
    void push(node *n)
    {
          if(top == NULL)
          {
                top = n;
          }
          else
          {
                n->link = top;
                top = n;
          }
    }
    void pop()
    {
          if(top == NULL)
          {
                cout<<"Stack underflow.";
          }
          else
          {
                cout<<"Popped item is "<<top->info;
                tempNode = top;
                top = top->link;
                delete tempNode;
          }
    }
    void displayStack()
    {
          tempNode = top;

          if(tempNode==NULL)
          {
                cout<<"Stack Empty.";
          }
          else
          {
                cout<<"top ->";
                while(tempNode!=NULL)
                {
                      cout<<tempNode->info<<endl;
                      tempNode = tempNode->link;
                      cout<<"      ";
                }
          }
    }
    int main()
    {
          int n,x;
          node *nptr;

          while(1)
          {
```

```
        cout<<"Stack Operations Menu\n  1. Push\n  2. Pop\n  3. Exit";

        cout<<endl<<"_____________________________";
    cout<<endl<<"Stack:\n------"<<endl;
    displayStack();
    cout<<endl<<"_____________________________"<<endl;

        cout<<"\nEnter your choice [1-3]: ";
        cin>>n;

        switch(n){
              case 1: cout<<"\nEnter item: ";
                      cin>>x;
                      nptr=createNode(x);
                      push(nptr);
                      break;

              case 2: pop();
                      break;

              case 3: exit(0);

              default: cout<<"\nWrong choice!!";
         }
         getch();
         system("cls");
    }
    return 0;
}
```

4. APPLICATIONS OF STACKS

Stack is a restricted list which follows last in first out (LIFO) strategy. It is restricted in the sense that insertion and deletion of items take place from only one end called the top of the stack. It means the item that is added at last will be removed at first. The important property of a stack is that items are deleted in the reverse order. Stack is a widely used data structure in many applications. Some of the applications of stacks are listed below:

a) Expression Evaluation and Expression Conversion

Stack is used to evaluate prefix, postfix and infix expressions. It is also used to convert one form of expression to another.

b) Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Compiler's syntax check for matching braces is implemented by using stack.

c) Memory management

Any modern computer environment uses a stack as the primary memory management model for a running program. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack.

d) Backtracking

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal. For example, 1) to find your way through a maze or 2) to find a path from one point in a graph (roadmap) to another point or 3) to play a game in which there are moves to be made (checkers, chess). In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative. Consider the maze problem. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking. An example of a backtracking algorithm is depth-first search, which finds all vertices of a graph that can be reached from a specified starting vertex.

e) Function Call

Another great use of stack is during the function call and return process. When we call a function from a point, after calling the function, we also have to come back to the point from which we called the function to continue the execution. It is done using stack. Space for parameters and local variables is created internally using a stack.

Almost all calling conventions—the ways in which subroutines receive their parameters and return results—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack.

f) Recursive Solutions

Recursion is implemented by a stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements.

Recursion is technique of solving any problem by calling same function again and again until a condition satisfied where recursion stops and it starts calculating the solution from that point. In recursion, last function called needs to be completed first. The stack also works as a LIFO mechanism. Therefore, stack is used to implement recursion.

g) Undo and Redo Mechanism

The undo and redo mechanisms in text editors or any other editors are accomplished by keeping all text changes in a stack.

h) Back and Forward Mechanism

The history of web pages that we have opened during web browsing is stored in stack so that we can go back or forward pages from the current page. The stack is used in the similar application for example window folder explorer.

5. INFIX, PREFIX AND POSTFIX EXPRESSION

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.

a) Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as `A * ( B + C ) / D` is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets `( )` to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

b) Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to `A B C + * D /`
The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit: `( (A (B C +) *) D /)`. Thus, the

"*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

c) Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to
`/ * A + B C D`
As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:
`(/ (* A (+ B C) ) D)`

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: `A - B` does not mean the same as `B - A`; the former is equivalent to `A B -` or `- A B`, the latter to `B A -` or `- B A`).

Examples:

Infix	Postfix	Prefix	Notes
`A * B + C / D`	`A B * C D / +`	`+ * A B / C D`	multiply A and B, divide C by D, add the results
`A * (B + C) / D`	`A B C + * D /`	`/ * A + B C D`	add B and C, multiply by A, divide by D
`A * (B + C / D)`	`A B C D / + *`	`* A + B / C D`	divide C by D, add B, multiply by A

d) Converting between these notations

The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation. For example:

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

You can convert directly between these bracketed forms simply by moving the operator within the brackets e.g. (X + Y) or (X Y +) or (+ X Y). Repeat this for all the operators in an expression, and finally remove any superfluous brackets.

6. Evaluation of Infix, Postfix and Prefix Expressions

a) Evaluation of Infix Expression

We will use two stacks. One stack is Operand stack which is used to keep operands and another is Operator stack which is used to keep operators (+, -, *, / and ^). Let us consider a "process" means,

a. pop operand stack once (value1)
b. pop operator stack once (operator)
c. pop operand stack again (value2)
d. compute value2 operator value1
e. push the value obtained in operand stack

(i) Algorithm

1. Given an infix expression, scan one character from the expression at a time from left to right. Until the end of the expression is reached, perform only one of the following steps (a) through (f):
   a. If the character is an operand, push it onto the operand stack.
   b. If the character is an operator, and the operator stack is empty then push it onto the operator stack.
   c. If the character is an operator and the operator stack is not empty, and the character's precedence is greater than the precedence of the stack top of operator stack, then push the character onto the operator stack.
   d. If the character is "(", then push it onto operator stack.
   e. If the character is ")", then "process" as explained above until the corresponding "(" is encountered in operator stack. At this stage POP the operator stack and ignore "(."
   f. If cases (a), (b), (c), (d) and (e) do not apply, then process as explained above.
2. When there are no more input characters, keep processing until the operator stack becomes empty.

3. The values left in the operand stack is the final result of the expression.

(ii) Example

Infix expression: **3+4*5*(4+3)-1/2+1**

Evaluation of the expression:

SN	Character Scanned	Operand Stack	Operator Stack	Operation Performed
1	3	3		
2	+	3	+	
3	4	3 4	+	
4	*	3 4	+ *	
5	5	3 4 5	+ *	
6	*	3 20	+ *	Pop 5, pop *, pop 4, perform 4 * 5 and push result (20) into operand stack
7	(	3 20	+ * (	
8	4	3 20 4	+ * (	
9	+	3 20 4	+ * (+	
10	3	3 20 4 3	+ * (+	
11	)	3 20 7	+ *	pop 3, pop +, pop 4, pop (and compute 4+3 = 7, push 7 into operand stack, ignore)
12	-	3 140	+	pop 7, pop *, pop 20, compute 20*7 = 140, push 140 into operand stack
		143	-	Pop 140, pop +, pop 3, compute 3 + 140 = 143 , push 143 into operand stack, push – into operator stack
	1	143, 1	-	
	/	143, 1	- /	
	2	143 1 2	- /	
	+	143 0.5	-	Pop 2, pop /, pop 1, compute ½ = 0.5, push 0.5 into operand stack
		142.5	+	Pop 0.5, pop -, pop 143, compute 143-0.5=142.5, push 142.5 into operand stack and push + into operator stack
	1	142.5 1	+	
		143.5		Pop 1, pop +, pop 142.5, compute 142.5+1=143.5, push 143.5 into operand stack

The final value at operand stack = 143.5

Therefore, the value of the expression = 143.5

(iii) Evaluation of Infix Expression using Two Stacks in C++

```
/*
Q: Evaluation of Infix Expression using Two Stacks.
Algo: Here we have two stacks.
1) Operator Stack
2) Operand Stack.
```

```
Here we process the given infix expression string from left to right.
1) If we come across an operand , We push the operand into the Operand
Stack.
2) If we come across an an operator, We follow the following algorithm
for operators

*********
Priority of operators
^ --> 4
*,/ --> 3
+,- --> 2
( --> 1
a) If the operator stack is empty, push the operator into the stack.
b) If the operator stack is not empty, compare the priority of operator
on the top of the stack to the current operator
c) let t = priority of operator on the top of the stack, p = priority.
of current operator
while(t >= p):
pop(t)
after popping all operators whose priority is greater than the current
operator, push this operator on the stack.
**. if the operator is ( just push in to the stack, without considering
priority.
**. if the operator is ) , Pop the operators and append till we encounter
matching (. don't add (,) to the output.

After processing completing string. pop all the operators from stack

*** When an operator is popped from the stack, perform calculation for
the top two operands of operand stack with this operator.
Push the result on to the operand stack .

*/

#include<iostream>
#include<stack>

using namespace std;

int pri(char ch)
{
      switch (ch)
      {
            case '(':
                  return 1;
            case '+':
                  //return 2;
            case '-':
                  return 3;

            case '*':
                  //return 4;
            case '/':
```

```
                return 5;

            case '^':
                return 6;
        }
        return -1;
}


float calculate(char op, float l , float r)
{
        if(op == '+')
        {
                return l + r;
        }
        else if(op == '-')
        {
                return l - r ;
        }
        else if(op == '*')\
        {
                return l * r;
        }
        else if(op == '/')
        {
                if(r > 0)
                {
                        return l/r;
                }
                return 0;
        }
        else if(op == '^')
        {
                int b = l; // l is made int and stored at b
                int p = r; // r is made int and stored at p
                return b ^ p;
        }
        return -1;
}

int main()
{
char str[] = "3+4*5*(4+3)-1/2+1";
//char str[] = "3+4*5*4+3-1/2+1";
float l = sizeof(str)/sizeof(char);
int k = 0;
stack<char> s;
stack<float> op_s;
cout <<"InFix Expression: " << str << endl;
int i = 0;
while(str[i] != '\0')
{
        if(str[i] == '(')
        {
                s.push('(');
        }else if(str[i] == ')')
        {
                while(s.top() != '('){
```

```
                float r = op_s.top();
                op_s.pop();
                float l = op_s.top();
                op_s.pop();
                float re = calculate(s.top(),l,r);
                op_s.push(re);
                s.pop();
            }
            s.pop();
      }else  if(str[i]=='+' || str[i]=='-' || str[i]=='*' || str[i]=='/' ||
str[i]=='^'){
            float pC = pri(str[i]);
            while(!s.empty() && pri(s.top()) >= pC){
                  float r = op_s.top();
                  op_s.pop();
                  float l = op_s.top();
                  op_s.pop();
                  float re = calculate(s.top(),l,r);
                  op_s.push(re);
                  s.pop();
            }
            s.push(str[i]);
      }else{
            op_s.push(int(str[i])- 48);
      }
      i++;
}
while(!s.empty()){
      float r = op_s.top();
      op_s.pop();
      float l = op_s.top();
      op_s.pop();
      float re = calculate(s.top(),l,r);
      op_s.push(re);
      s.pop();
}
cout <<"Result: " << op_s.top() << endl;
return 0;
}
```

Output:

Result: 143.5

Time complexity is O(n).

b) Evaluation of Postfix Expressions

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

(i) Algorithm:

1) Create an operand stack to store operands.

2) Scan the given expression from left to right a character at a time and do the following for every scanned character.
    a. If the character is a number, push it into the stack
    b. If the character is an operator (say op), pop operands from stack. The first popped is the second operand (say value1) and the second popped is the first operand (say value2). Compute "value2 op value1" and push the result back to the stack.
3) When the scanning of expression is finished, the number in the stack is the evaluated value of expression.

(ii) Example:

Postfix expression: **2 3 1 * + 9 –**

Tracing of evaluation of this expression:

We scan all elements one by one from left to right.
1) Scan '2', it's a number, so push it to stack. Stack contains '2'
2) Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
3) Scan '1', again a number, push it to stack, stack now contains '2 3 1'
4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 2 + 3 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
6) Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.
7) Scan '-', it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.
8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

(iii) Evaluation of a postfix expression using a stack in C++

```
#include <iostream>
#include <string.h>
#include <malloc.h>

using namespace std;

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
```

```
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}


// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
```

```
            {
                int val1 = pop(stack);
                int val2 = pop(stack);
                switch (exp[i])
                {
                case '+': push(stack, val2 + val1); break;
                case '-': push(stack, val2 - val1); break;
                case '*': push(stack, val2 * val1); break;
                case '/': push(stack, val2/val1); break;
                }
            }
        }
        return pop(stack);
    }

    // Driver program to test above functions
    int main()
    {
        char exp[] = "231*+9-";
        cout<<"postfix evaluation: "<< evaluatePostfix(exp);
        return 0;
    }
```

Output:

```
postfix evaluation: -4
```

Time complexity of evaluation algorithm is O(n) where n is number of characters in input expression. There are following limitations of above implementation:

1. It supports only 4 binary operators '+', '*', '-' and '/'. It can be extended for more operators by adding more switch cases.
2. The allowed operands are only single digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.

(iv) Evaluation of a postfix expression having multiple digit operands

Below given is the extended program which allows operands having multiple digits.

```
#include <bits/stdc++.h>
using namespace std;

// Stack type
class Stack
{
    public:
    int top;
    unsigned capacity;
    int* array;
};
```

```
// Stack Operations
Stack* createStack( unsigned capacity )
{
    Stack* stack = new Stack();

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = new int[(stack->capacity * sizeof(int))];

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(Stack* stack)
{
    return stack->top == -1 ;
}

int peek(Stack* stack)
{
    return stack->array[stack->top];
}

int pop(Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(Stack* stack,int op)
{
    stack->array[++stack->top] = op;
}


// The main function that returns value
// of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;
```

```
    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        //if the character is blank space then continue
        if(exp[i] == ' ')continue;

        // If the scanned character is an
        // operand (number here),extract the full number
        // Push it to the stack.
        else if (isdigit(exp[i]))
        {
            int num=0;

            //extract full number
            while(isdigit(exp[i]))
            {
            num = num * 10 + (int)(exp[i] - '0');
                i++;
            }
            i--;

            //push the element in the stack
            push(stack,num);
        }

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);

            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;

            }
        }
    }
    return pop(stack);
}

// Driver code
int main()
{
    char exp[] = "100 200 + 2 / 5 * 7 +";
    cout << evaluatePostfix(exp);
```

```
    return 0;
}
```

Output :
757

c) Evaluation of Prefix Expressions

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions.

(i) Algorithm

EVALUATE_PREFIX (STRING)

1. Scan the prefix expression from right to left (that is from the end of the expression)
2. If character is an operand, push it to Stack
3. If the character is an operator pop two elements from the Stack. Operand that is popped at first is the first operand (say operand1) and the operator that is popped at second is the second operator (say operand2). Compute operand1 operator operand2, and push the result back to the Stack
4. Repeat the above steps until there are no characters left to be scanned in the expression.
5. The Result is stored at the top of the Stack, return it
6. End

(ii) Example 1:

Expression: +9*26

Character Scanned	Operand Stack	Processing
6	6	6 is an operand. Push to Stack.
2	6 2	2 is an operand. Push to the stack.
*	12	* is an operator. Pop 2 and 6, multiply (2*6) and push result to the stack.
9	12 9	9 is an operand, push to Stack.
+	21	+ is an operator, pop 9 and 12 add them and push result to Stack.

```
Result: 21
```

(iii)Example 2:

Prefix Expression : -/*2*5+3652		
Reversed Prefix Expression: 2563+5*2*/-		
Token	**Action**	**Stack**
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
6	Push **6** to stack	[2, 5, 6]
3	Push **3** to stack	[2, 5, 6, 3]
+	Pop **3** from stack	[2, 5, 6]
	Pop **6** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
5	Push **5** to stack	[2, 5, 9, 5]
*	Pop **5** from stack	[2, 5, 9]
	Pop **9** from stack	[2, 5]
	Push **5*9=45** to stack	[2, 5, 45]
2	Push **2** to stack	[2, 5, 45, 2]
*	Pop **2** from stack	[2, 5, 45]
	Pop **45** from stack	[2, 5]
	Push **2*45=90** to stack	[2, 5, 90]
/	Pop **5** from stack	[2, 5]
	Pop **90** from stack	[2]
	Push **90/5=18** to stack	[2, 18]
-	Pop **18** from stack	[2]
	Pop **2** from stack	[]
	Push **18-2=16** to stack	[16]
Result : 16		

(iv)Evaluation of a prefix expression using c++

```
#include <bits/stdc++.h>
using namespace std;

bool isOperand(char c)
{
```

```
    // If the character is a digit then it must
    // be an operand
    return isdigit(c);
}

double evaluatePrefix(string exprsn)
{
    stack<double> Stack;

    for (int j = exprsn.size() - 1; j >= 0; j--) {

        // Push operand to Stack
        // To convert exprsn[j] to digit subtract
        // '0' from exprsn[j].
        if (isOperand(exprsn[j]))
            Stack.push(exprsn[j] - '0');

        else {

            // Operator encountered
            // Pop two elements from Stack
            double o1 = Stack.top();
            Stack.pop();
            double o2 = Stack.top();
            Stack.pop();

            // Use switch case to operate on o1
            // and o2 and perform o1 O o2.
            switch (exprsn[j]) {
            case '+':
                Stack.push(o1 + o2);
                break;
            case '-':
                Stack.push(o1 - o2);
                break;
            case '*':
                Stack.push(o1 * o2);
                break;
            case '/':
                Stack.push(o1 / o2);
                break;
            }
        }
    }
```

```
    return Stack.top();
}
int main()
{
    string exprsn = "+9*26";
    cout << evaluatePrefix(exprsn) << endl;
    return 0;
}
```

Output:

```
21
```

Note: To perform more types of operations only the switch case table needs to be modified. This implementation works only for single digit operands. Multi-digit operands can be implemented if some character like space is used to separate the operands and operators.

7. EXPRESSION CONVERSION

a) Conversion of Infix Expressions to Prefix and Postfix

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that A + B * C can be written as (A + (B * C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression (B * C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C *, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (see Fig 2.3).

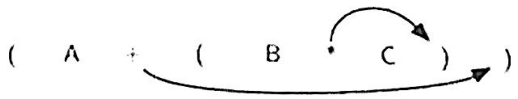

Fig 2.3: Moving Operators to the Right for Postfix Notation

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Fig. 2.4). The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

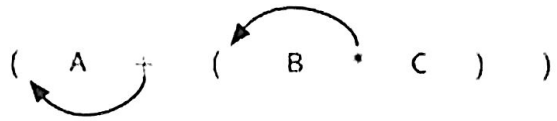

Fig. 2.4: Moving Operators to the Left for Prefix Notation

So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

Here is a more complex expression: (A + B) * C - (D - E) * (F + G). Fig. 2.5 shows the conversion to postfix and prefix notations.

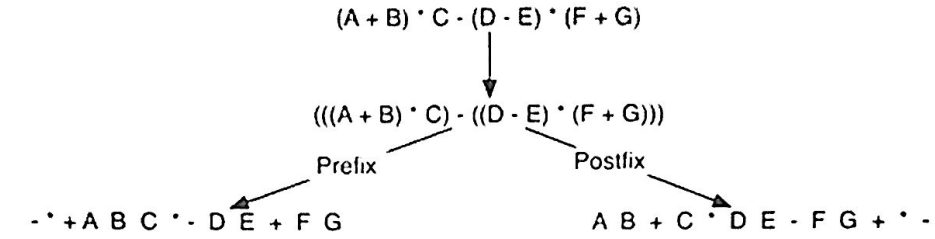

Fig. 2.5: Converting a Complex Expression to Prefix and Postfix Notations¶

b) Infix to Postfix Conversion

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression A + B * C. As shown above, A B C * + is the postfix equivalent. We have already noted that the operands A, B, and C stay in their relative positions. It is only the operators that change position. Let's look again at the operators in the infix expression. The first operator that appears from left to right is +. However, in the postfix expression, + is at the end since the next operator, *, has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about (A + B) * C? Recall that A B + C * is the postfix equivalent. Again, processing this infix expression from left to right, we see + first. In this case, when we see *, + has already been placed in the result

expression because it has precedence over * by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

(i) Algorithm

In this algorithm, all operands are printed (or sent to output) when they are read.

Assume the infix expression is a string of tokens. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

1. Create an empty stack called opstack for keeping operators. Create an empty list for output.
2. Scan a single token in the infix expression from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the opstack.
    - If the token is a right parenthesis, pop the opstack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
    - If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.
3. When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.

The converted postfix expression will be in the output list.

(ii) Example 1:

Infix expression: A * (B + C * D) + E

Equivalent postfix expression: A B C D * + * E +

Tracing of Conversion Process:

SN	Current token	operator stack (opstack)	postfix string (output list)
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			**A B C D * + * E +**

A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

(iii) Example 2:

A * B + C becomes A B * C +

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	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 rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

(iv) Example 3:

A + B * C becomes A B C * +

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when they are both popped off in lines 6, their order will be reversed.

(v) Example 4:

A * B ^ C + D becomes A B C ^ * D +

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

(vi) Example 5:

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

Resultant Postfix Expression: ABC*DEF^/G*-H*+

(vii) Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

c) Infix to Prefix using two stacks

The idea is to use one operator stack for storing operators and other operand stack to store operands.

(i) Algorithm

1. Traverse the infix expression from left to right and check if given character is an operator or an operand.
    a) If it is an operand, then push it into operand stack.
    b) If it is an operator, then check if priority of current operator is greater than or less than or equal to the operator at top of the stack. If priority is greater, then push operator into operator stack. Otherwise pop two operands from operand stack, pop operator from operator stack and push string **operator + operand1 + operand 2** into operand stack. Keep popping from both stacks and pushing result into operand stack until priority of current operator is less than or equal to operator at top of the operator stack.
    c) If current character is '(', then push it into operator stack.
    d) If current character is ')', then check if top of operator stack is opening bracket or not. If not pop two operands from operand stack, pop operator from operator stack and push string **operator + operand1 + operand 2** into operand stack. Keep popping from both stacks and pushing result into operand stack until top of operator stack is an opening bracket.
2. The final prefix expression is present at top of operand stack.

(ii) Conversion from infix to prefix using C++

```
#include <bits/stdc++.h>
using namespace std;

// Function to check if given character is
// an operator or not.
bool isOperator(char c)
{
    return (!isalpha(c) && !isdigit(c));
}

// Function to find priority of given
// operator.
int getPriority(char C)
{
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}

// Function that converts infix
// expression to prefix expression.
string infixToPrefix(string infix)
{
    // stack for operators.
    stack<char> operators;

    // stack for operands.
    stack<string> operands;
```

```
for (int i = 0; i < infix.length(); i++) {

    // If current character is an
    // opening bracket, then
    // push into the operators stack.
    if (infix[i] == '(') {
        operators.push(infix[i]);
    }

    // If current character is a
    // closing bracket, then pop from
    // both stacks and push result
    // in operands stack until
    // matching opening bracket is
    // not found.
    else if (infix[i] == ')') {
        while (!operators.empty() &&
               operators.top() != '(') {

            // operand 1
            string op1 = operands.top();
            operands.pop();

            // operand 2
            string op2 = operands.top();
            operands.pop();

            // operator
            char op = operators.top();
            operators.pop();

            // Add operands and operator
            // in form operator +
            // operand1 + operand2.
            string tmp = op + op2 + op1;
            operands.push(tmp);
        }

        // Pop opening bracket from
        // stack.
        operators.pop();
    }

    // If current character is an
    // operand then push it into
    // operands stack.
    else if (!isOperator(infix[i])) {
        operands.push(string(1, infix[i]));
    }

    // If current character is an
    // operator, then push it into
    // operators stack after popping
    // high priority operators from
    // operators stack and pushing
    // result in operands stack.
    else {
```

```
            while (!operators.empty() &&
                   getPriority(infix[i]) <=
                     getPriority(operators.top())) {

                string op1 = operands.top();
                operands.pop();

                string op2 = operands.top();
                operands.pop();

                char op = operators.top();
                operators.pop();

                string tmp = op + op2 + op1;
                operands.push(tmp);
            }

            operators.push(infix[i]);
        }
    }

    // Pop operators from operators stack
    // until it is empty and add result
    // of each pop operation in
    // operands stack.
    while (!operators.empty()) {
        string op1 = operands.top();
        operands.pop();

        string op2 = operands.top();
        operands.pop();

        char op = operators.top();
        operators.pop();

        string tmp = op + op2 + op1;
        operands.push(tmp);
    }

    // Final prefix expression is
    // present in operands stack.
    return operands.top();
}

// Driver code
int main()
{
    string s = "(A-B/C)*(A/K-L)";
    cout << infixToPrefix(s);
    return 0;
}
```

Output:

```
*-A/BC-/AKL
```

d) Postfix to Infix

The idea is to use one operand stack to store operands.

(i) Algorithm

1. Traverse the postfix expression from left to right until the end of the expression and check if given character is an operator or an operand
2. If the character is an operand
   2.1 Push it onto the stack.
3. Otherwise,
   3.1 The character is an operator.
   3.2 Pop the top 2 values from the stack. The operand that is popped at first is second operand (say operand1). The operand that is popped at second is first operand (say operand2).
   3.3 Form string as: '(' + **operand**2 + operator + **operand**1 + ')'
   3.4 Push the resulted string back to stack.
4. If there is only one value in the stack
   4.1 The value in the stack is the desired infix string.

(ii) Conversion of Postfix into Infix using C++

```
#include <bits/stdc++.h>
using namespace std;

bool isOperand(char x)
{
   return (x >= 'a' && x <= 'z') ||
          (x >= 'A' && x <= 'Z');
}

// Get Infix for a given postfix
// expression
string getInfix(string exp)
{
    stack<string> s;

    for (int i=0; exp[i]!='\0'; i++)
    {
        // Push operands
        if (isOperand(exp[i]))
        {
```

```
            string op(1, exp[i]);
            s.push(op);
        }

        // We assume that input is
        // a valid postfix and expect
        // an operator.
        else
        {
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push("(" + op2 + exp[i] +
                    op1 + ")");
        }
    }

    // There must be a single element
    // in stack now which is the required
    // infix.
    return s.top();
}

// Driver code
int main()
{
    string exp = "ab*c+";
    cout << getInfix(exp);
    return 0;
}
```

Output:

```
((a*b)+c)
```

e) Prefix to Infix

Computers usually does the computation in either prefix or postfix (usually postfix). But for humans, its easier to understand an Infix expression rather than a prefix. Hence conversion is need for human understanding.

(iii)Algorithm

1. Read the Prefix expression in reverse order (from right to left)
2. If the symbol is an operand, then push it onto the Stack
3. If the symbol is an operator, then pop two operands from the Stack
   a. Create a string by concatenating the two operands and the operator between them. The operand that is popped at first is first operand (say operand1). The operand that is popped at second is second operand (say operand2)
   b. String = '(' + operand1 + operator + operand2 + ')'
   c. And push the resultant string back to Stack
4. Repeat the above steps until end of Prefix expression.

(iv) Conversion form prefix to Infix using C++

```
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
  switch (x) {
  case '+':
  case '-':
  case '/':
  case '*':
    return true;
  }
  return false;
}

// Convert prefix to Infix expression
string preToInfix(string pre_exp) {
  stack<string> s;

  // length of expression
  int length = pre_exp.size();

  // reading from right to left
  for (int i = length - 1; i >= 0; i--) {

    // check if symbol is operator
    if (isOperator(pre_exp[i])) {

      // pop two operands from stack
```

```
        string op1 = s.top();    s.pop();
        string op2 = s.top();    s.pop();

        // concat the operands and operator
        string temp = "(" + op1 + pre_exp[i] + op2 + ")";

        // Push string temp back to stack
        s.push(temp);
      }
      // if symbol is an operand
      else {
        // push the operand to the stack
        s.push(string(1, pre_exp[i]));
      }
    }
    // Stack now contains the Infix expression
    return s.top();
}
int main() {
    string pre_exp = "*-A/BC-/AKL";
    cout << "Infix : " << preToInfix(pre_exp);
    return 0;
}
```

Output:

```
Infix : ((A-(B/C))*((A/K)-L))
```

f) Prefix to Postfix

Conversion of Prefix expression directly to Postfix without going through the process of converting them first to Infix and then to Postfix is much better in terms of computation and better understanding the expression.

(i) Algorithm

1. Read the Prefix expression in reverse order (from right to left)
2. If the symbol is an operand, then push it onto the Stack
3. If the symbol is an operator, then pop two operands from the Stack
   a. Create a string by concatenating the two operands and the operator after them.
   b. **string = operand1 + operand2 + operator**
   c. push the resultant string back to Stack
4. Repeat the above steps until end of Prefix expression.

(ii) Conversion from prefix to postfix

```
#include <iostream>
#include <stack>
using namespace std;

// funtion to check if character is operator or not
bool isOperator(char x) {
  switch (x) {
  case '+':
  case '-':
  case '/':
  case '*':
    return true;
  }
  return false;
}

// Convert prefix to Postfix expression
string preToPost(string pre_exp) {

  stack<string> s;

  // length of expression
  int length = pre_exp.size();

  // reading from right to left
  for (int i = length - 1; i >= 0; i--) {

    // check if symbol is operator
    if (isOperator(pre_exp[i])) {

      // pop two operands from stack
      string op1 = s.top(); s.pop();
      string op2 = s.top(); s.pop();

      // concat the operands and operator
      string temp = op1 + op2 + pre_exp[i];

      // Push string temp back to stack
      s.push(temp);
    }

    // if symbol is an operand
    else {
```

```
            // push the operand to the stack
            s.push(string(1, pre_exp[i]));
        }
    }

    // stack contains only the Postfix expression
    return s.top();
}
int main() {
    string pre_exp = "*-A/BC-/AKL";
    cout << "Postfix : " << preToPost(pre_exp);
    return 0;
}
```

Output:

```
Postfix : ABC/-AK/L-*
```

g) Postfix to Prefix

Conversion of Postfix expression directly to Prefix without going through the process of converting them first to Infix and then to Prefix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

(i) Algorithm

1. Read the Postfix expression from left to right
2. If the symbol is an operand, then push it onto the Stack
3. If the symbol is an operator, then pop two operands from the Stack
    a. Create a string by concatenating the two operands and the operator before them.
    b. **string = operator + operand2 + operand1**
    c. Push the resultant string back to Stack
4. Repeat the above steps until end of Prefix expression.

(ii) Conversion from postfix to prefix

```
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x)
{
    switch (x) {
    case '+':
    case '-':
```

```
    case '/':
    case '*':
        return true;
    }
    return false;
}

// Convert postfix to Prefix expression
string postToPre(string post_exp)
{
    stack<string> s;

    // length of expression
    int length = post_exp.size();

    // reading from right to left
    for (int i = 0; i < length; i++) {

        // check if symbol is operator
        if (isOperator(post_exp[i])) {

            // pop two operands from stack
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();

            // concat the operands and operator
            string temp = post_exp[i] + op2 + op1;

            // Push string temp back to stack
            s.push(temp);
        }

        // if symbol is an operand
        else {

            // push the operand to the stack
            s.push(string(1, post_exp[i]));
        }
    }

    // stack[0] contains the Prefix expression
    return s.top();
}
```

```
int main()
{
    string post_exp = "ABC/-AK/L-*";
    cout << "Prefix : " << postToPre(post_exp);
    return 0;
}
```

Output:

```
Prefix : *-A/BC-/AKL
```