**Assignment No. –7**

**Title:-** Expression Evaluation and Conversion.

**Problem statement:-**Implement C++ program for expression conversion.

1. Infix to Postfix.
2. Infix to Prefix.
3. Prefix to Infix.
4. Postfix to Infix.
5. Postfix to Prefix.

**Learning Objectives:-**To implement concept of stack and application of stack using expression conversion.

**Pre-requisites:-** Concept of stack and all operations performed on stack.

**Theory:-**

**Stack**

A stack is defined as, a restricted list where all insertion and deletion operation are made only at one end i.e. top.

**Stack Operations**

1. **Create:-**

intnum[100];

int top=-1;

1. **Empty:-**Empty is a operation that takes stack as argument it checks whether it is empty or not and return a Boolean value T and F.

If(top = = -1)

{

Return 1;

}

Else

{

Return 0;

}

1. Gettop:-Gettop operation check status of stack if the stack is empty it gives stack underflow else its returns a copy of element that present at top of stack. Here top is not updated as the element is not deleted form the stack. Rather the element is still at top location.

If(top = = -1)

{

Cout<<”stack underflow/empty”;

}

Else

{

Return(stack [top]);

}

1. Push:- The push operation insert an element size(maximum capacity). Elementinsertion is possible only if when the stack is not full.

If(top = = maxcapacity-1)

{

Cout<<”stack full/overflow”;

}

Else

{

Top++;

Stack[top]=element;

}

1. Pop:- The pop operation delete the element at top of the stack and return the same this is done only if stack is not empty. This can be checked by empty function.If the stack is not empty then the element at the top of stack is returned and is decreased by one.

If(top = = -1)

{

Cout<<”stack underflow”;

}

Else

{

Return(stack[top--]);

}

Application of Stack

* Converting expression.
* Evaluate expression.
* Checking well formed parenthesis.
* Reversing a string.
* Processing function calls.
* Parcing of computer program.
* Simulating recursion.
* In computation such as decimal to binary conversion.
* In back tracking algorithm.

Expression Evaluation and Conversion

|  |  |
| --- | --- |
| Operation | Priority |
| Exponential ^,+,-,~ | 1 |
| Multiplication, Division(\*,/) | 2 |
| Addition, subtraction (+,-) | 3 |
| Relational Operator<,>,<=,>= | 4 |
| Logic AND | 5 |
| Logic OR | 6 |

Infix to Postfix conversion:-

For this conversion initially fully parenthesis is given infix expression need to convert into postfix. Use operator priority and associativity rules move all operators so that they can replace corresponding right parenthesis. Finally delete all parathesis and we get postfix expression.

Example:- A^B\*C-C+D/A/(E+F)

|  |  |  |
| --- | --- | --- |
| Character scan | Stack content | Postfix expression |
| A |  | A |
| ^ | ^ | A |
| B | ^ | AB |
| \* | \* | AB^ |
| C | \* | AB^C |
|  | - | AB^C\* |
| C | - | AB^C\*C |
| + | -+ | AB^C\*C |
| D | -+ | AB^C\*CD |
| / | -+/ | AB^C\*CD |
| A | -+/ | AB^C\*CDA |
| / | -+// | AB^C\*CDA |
| ( | -+//( | AB^C\*CDA |
| E | -+//( | AB^C\*CDAE |
| + | -+//(+ | AB^C\*CDAE |
| F | -+//(+ | AB^C\*CDAEF |
| ) | -+// | AB^C\*CDAEF+ |
|  |  | AB^C\*CDAEF+//+- |

**Algorithm:-**

Step-1) START.

Step-2) From left to right character by character by ‘#’ or ‘$’.

Ch=get\_token(E);

Step-3)while(ch!=’#’)

Intch=’)’ then pop( )

While (ch!=’(’)

Display ch

Ch=pop

End=pop

End while

While(ch if ch=operand)

Display operand

If ch=operator then

Pop operand and display it

End while

Step-4) If ch=’#’ then pop contents of stack and display.

Step-5) STOP.

Summary Rules for Conversion

* Print operand as arrived.
* If the stack is empty or the content left parenthesis on top, push the incoming operation on to the stack.
* If the incoming symbol is left parenthesis push it on the stack.
* If the incoming symbol is right parenthesis the pop the stack. Print the operator until you see a left parenthesis.
* If incoming symbol has higher priority, then the top of the stack push it on to the stack.
* If incoming symbol has equal priority use association. If the association is left to right pop and print top of the stack. Then push incoming operation. If association is right to left then push incoming operator.
* If the incoming symbol has lower priority pop the stack and print the top operator and test the incoming operator against new top of the stack.
* At the end of expression pop and print all operators.

Infix to Prefix conversion

**Algorithm:-**

Step-1) START.

Step-2) Scan expression E character from right to left.

Ch=get\_next\_token(E);

Step-3) while(ch!=’#’)

do

if (ch=operand)

then push(ch) in display stack

if(ch==’)’) then hc =pop( ) from operator stack

while(ch!=’(’)

push(ch) inn display stack

ch=pop( )

end while

if(ch=operator)

push(ch) in operator stack

elsech=pop( ) from display stack

end while

Step-4) if(ch-‘#’)

Ch=pop(operator)

While(! Empty stack(display))

Ch=pop(operator) and display character

End while

Step-5) STOP.

Example:- A^B\*C-C+D/A/(E+F)

|  |  |  |
| --- | --- | --- |
| Char Scan | Stack Content | Prefix expression |
| ) | ) |  |
| F | ) | F |
| + | )+ | F |
| E | )+ | EF |
| ( |  | +EF |
| / | / | +EF |
| A | / | A+EF |
| / | // | A+EF |
| D | // | DA+EF |
| + | + | //DA+EF |
| C | + | C//DA+EF |
| - | +- | C//DA+EF |
| C | +- | CC//DA+EF |
| \* | +-\* | CC//DA+EF |
| B | +-\* | BCC//DA+EF |
| ^ | +-\*^ | BCC//DA+EF |
| A | +-\*^ | ABCC//DA+EF |
|  |  | +-\*^ABCC//DA+EF |

Postfix to Infix

**Algorithm:-**

Step-1) START.

Step-2) Scan expression E character from left to right.

Ch=get\_next\_token(E);

Step-3) while(ch!=’#’)

do

if (ch=operand)

then push ch

if(ch=operator)

then begin

t2=pop( ) and t1=pop( )

push(strcat[‘(’,t1 ch,t2,’)’])

end

ch=get.next\_token(E)

end while

Step-4) if(ch=‘#’)

Ch=pop(operator)

While(! Empty stack(display))

Ch=pop(operator) and display character

End while

Step-5) STOP.

Example:- AB^C\*C-DA/EF+/+

|  |  |
| --- | --- |
| Character Scan | Infix Expression |
| A | A |
| B | AB |
| ^ | A^B |
| C | A^BC |
| \* | A^B\*C |
| C | A^B\*CC |
| - | A^B\*C-C |
| D | A^B\*C-CD |
| A | A^B\*C-CDA |
| / | A^B\*C-CD/A |
| E | A^B\*C-CD/AE |
| F | A^B\*C-CD/AEF |
| + | A^B\*C-CD/AE+F |
| / | A^B\*C-CD/A/(E+F) |
| + | A^B\*C-C+D/A/(E+F) |

Postfix to Prefix

If the scan character is digital then push it in the stack.

If the scan character is operator then pop element from the string containing scanned operator and two pop the element. Push the resultant string into the stack.

**Algorithm:-**

Step-1) START.

Step-2) Scan expression E character from left to right.

Ch=get\_next\_token(E);

Step-3) while(ch!=’#’)

do

if (ch=operand)

then push ch

if(ch=operator)

then begin

t2=pop( ) and t1=pop( )

push(strcat[‘(’,ch, t1,t2,’)’])

end

ch=get.next\_token(E)

end while

Step-4) if(ch-‘#’)

Ch=pop(operator)

While(! Empty stack(display))

Ch=pop(operator) and display character

End while

Step-5) STOP.

Example:- AB^C\*C-DA/EF+/+

|  |  |
| --- | --- |
| Character Scan | Prefix Expression |
| A | A |
| B | AB |
| ^ | ^AB |
| C | ^ABC |
| \* | \*^ABC |
| C | \*^ABCC |
| - | -\*^ABCC |
| D | -\*^ABCCD |
| A | -\*^ABCCDA |
| / | -\*^ABCC/DA |
| E | -\*^ABCC/DAE |
| F | -\*^ABCC/DAEF |
| + | -\*^ABCC/DA+EF |
| / | -\*^ABCC//DA+EF |
| + | +-\*^ABCC//DA+EF |

Prefix to Infix

**Algorithm:-**

Step-1) START.

Step-2) Scan expression E character from right to left.

Ch=get\_next\_token(E);

Step-3) while(ch!=’#’)

do

if (ch=operand)

then push ch

if(ch=operator)

then begin

t2=pop( ) and t1=pop( )

push(strcat[‘(’,t1, ch,t2,’)’])

end

ch=get.next\_token(E)

end while

Step-4) if(ch-‘#’)

Ch=pop(operator)

While(! Empty stack(display))

Ch=pop(operator) and display character

End while

Step-5) STOP.

Example:- +-\*^ABCC//DA+EF

|  |  |
| --- | --- |
| Character Scan | Infix Expression |
| F | F |
| E | EF |
| + | E+F |
| A | AE+F |
| D | DAE+F |
| / | DA/(E+F) |
| / | D/A/(E+F) |
| C | CD/A/(E+F) |
| C | CCD/A/(E+F) |
| B | BCCD/A/(E+F) |
| A | ABCCD/A/(E+F) |
| ^ | A^BCCD/A/(E+F) |
| \* | A^B\*CCD/A/(E+F) |
| - | A^B\*C-CD/A/(E+F) |
| + | A^B\*C-C+D/A/(E+F) |

Prefix to Postfix

**Algorithm:-**

Step-1) START.

Step-2) Scan expression E character from left to right.

Ch=get\_next\_token(E);

Step-3) while(ch!=’#’)

do

if (ch=operand)

then push ch

if(ch=operator)

then begin

t2=pop( ) and t1=pop( )

push(strcat[‘(’, t1,t2,ch’)’])

end

ch=get.next\_token(E)

end while

Step-4) if(ch-‘#’)

Ch=pop(operator)

While(! Empty stack(display))

Ch=pop(operator) and display character

End while

Step-5) STOP.

Example:- \*+A-BC/-DE+-FGH

|  |  |
| --- | --- |
| Character Scan | Postfix Expression |
| H | H |
| G | GH |
| F | FGH |
| - | FG-H |
| + | FG-H+ |
| E | EFG-H+ |
| D | DEFG-H+ |
| - | DE-FG-H+ |
| / | DE-FG-H+/ |
| C | CDE-FG-H+/ |
| B | BCDE-FG-H+/ |
| - | BC-DE-FG-H+/ |
| A | ABC-DE-FG-H+/ |
| + | ABC-+DE-FG-H+/ |
| \* | ABC-+DE-FG-H+/\* |

**\*Advantages of merge sort:-**

1) As its worst case complexity is 0 (n log n), it is very fast.

2) As compared to other sorting algorithms, its complexity is less.

**\*Disadvantages of merge sort:-**

1) Merge sort uses a lot of memory so it has high space complexity. As it divides and conquers, the memory management becomes tedious.

2) It does not go well when attempting to sort large data.

**\*Applications:-**

1) To calculate number of inversions.

2) E-commerce.

**\*Conclusion:-**Thus we, implemented merge sort to merge two link lists one containing positive numbers and other containing negative into one which was sorted.