|  |  |  |  |
| --- | --- | --- | --- |
|  | **Hope Foundation’s**  **Finolex Academy of Management and Technology, Ratnagiri** | | |
| **Department of CSE (AI&ML)** | | |
| Subject name: Data Structures Lab Subject Code: CSL301 | | | |
| Class | SECSE(AI&ML) | | Semester –III (CBCGS) Academic year: 2021-22 |
| Name of Student | PARAB AKSHAY GURUNATH **QUIZ Score:4 / 6** | | |
| Roll No | 21 | Experiment No. 02 | |
| Title: **Convert an Infix expression to Postfix expression using stack ADT.** | | | |

|  |
| --- |
| **1. Lab objectives applicable:**  **LOB3** - To develop application using data structure algorithms  **LOB4** – Ccompute the complexity of various algorithms.  **LOB5** - To introduce various techniques for representation of the data in the real world. **2. Lab outcomes applicable:**  **1. LO3 -** Students will be able to choose appropriate data structure and apply it in various problems **2. LOB4**- Students will be able to Compute the complexity of various algorithms  **3. LOB6**- Students will be able to demonstrate the ability to analyse, design, apply and use data structures to solve engineering problems and evaluate their solutions.  **3. Learning Objectives:**  1. To understand implementation of application for conversion of an infix expression to postfix expression using Stack ADT  2. To study application of appropriate data structure in particular problem.  **4. Practical applications of the experiment:** Stack ADT can be used in memory management, expression evaluation etc. |
| **5. Prerequisites**:  1. Understanding of C programming. |
| **6. Minimum Hardware Requirements**:  1. P-IV PC with 2GB RAM, 500GB HDD, NIC  **7. Software Requirements:**  1. Windows / Linux operating systems, C IDE. |

**8. Understanding Questions :**

1. What is Infix, prefix and postfix expression? How to convert infix expression into postfix expression using stack ADT.

2. Write a note on evaluation of postfix expression using Stack ADT.

**Quiz Questions :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Q.No** | **Question** | **Correct Answer** | **Given Answer** |
| **1** | If the input sequence is 1, 2, 3, 4, 5. If you popped the stack identify correct sequence of output possible after different combination of PUSH and POP instructions (permutations) | 4, 5, 3, 2, 1  3, 2, 1, 5, 4  2, 3, 5, 4, 1  4, 3, 2, 5, 1 | **5, 4, 3, 1 , 2, 4, 5, 3, 2, 1** |
| **2** | Infix to Postfix conversion of 2+5\*3^2+9 is | 2532^\*+9+ | **2532^\*+9+** |
| **3** | Convert abc\*+de/f\*- into infix expression | a+b\*c-d/e\*f | **a+b\*c-d/e\*f** |

Department of CSE (AI&ML)

|  |  |  |  |
| --- | --- | --- | --- |
| **4** | What data structure would you mostly likely see in non recursive implementation of a recursive algorithm? | Stack | **Stack** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **9. Experiment/Assignment Evaluation:** | | | | |
| **Sr. No.** | **Parameters Marks obtained** | | | **Out of** |
| **1** | Technical Understanding (Assessment may be done based on Q & A **or** any other relevant  **4 / 6**  method.) Teacher should mention the other method used - | | | 6 |
| **2** | Lab Performance | | | 2 |
| **3** | Punctuality | | | 2 |
| **Date of performance (DOP)** | |  | **Total marks obtained** | **10** |

**Signature of Faculty**

Department of CSE (AI&ML)

**10. Theory:**

**Infix, Prefix and Postfix Expressions:**

When we write an arithmetic expression such as B \* C, the form of the expression provides us with information so that we can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator \* appears between them in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.

Consider another infix example, A + B \* C. The operators + and \* still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the \* take B and C? The expression seems ambiguous.

In fact, we have been reading and writing these types of expressions for a long time and they do not cause us any problem. The reason for this is that we know something about the operators + and \*. Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

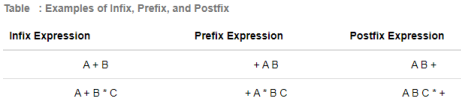
Let’s interpret the troublesome expression A + B \* C using operator precedence. B and C are multiplied first, and A is then added to that result. (A + B) \* C would force the addition of A and B to be done first before the multiplication. In expression A + B + C, by precedence (via associativity), the leftmost + would be done first.

Although all this may be obvious to us, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a **fully parenthesized** expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression A + B \* C + D can be rewritten as ((A + (B \* C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.

There are two other very important expression formats that may not seem obvious to us at first. Consider the infix expression A + B. What would happen if we moved the operator before the two operands? The resulting expression would be + A B. Likewise, we could move the operator to the end. We would get A B +. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands.



A + B \* C would be written as + A \* B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that \* has precedence over +. The addition operator then appears before the A and the result of the multiplication.

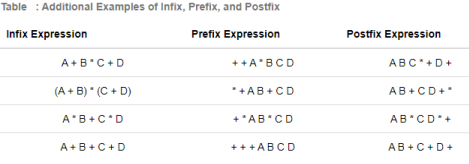
Department of CSE (AI&ML)

In postfix, the expression would be A B C \* +. Again, the order of operations is preserved since the \* appears immediately after the B and the C, denoting that \* has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Now consider the infix expression (A + B) \* C. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when A + B was written in prefix, the addition operator was simply moved before the operands, + A B. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us \* + A B C. Likewise, in postfix A B + forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then A B + C \*.

Consider these three expressions again. Something very important has happened. Where did the parentheses go? Why don’t we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

Table below shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.



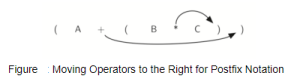
**Conversion of Infix Expressions to Prefix and Postfix:**

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As we might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

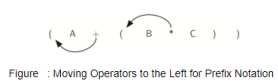
The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that A + B \* C can be written as (A + (B \* C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that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

Department of CSE (AI&ML)

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.

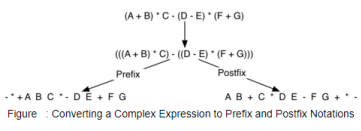


If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation. The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

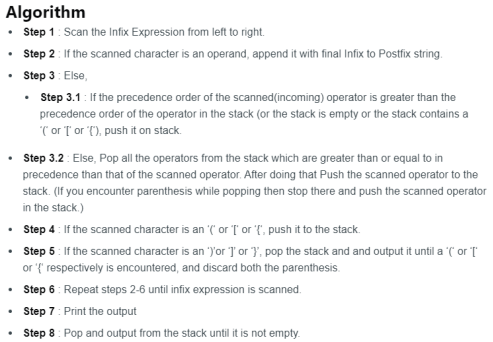


So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix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

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



Department of CSE (AI&ML)



**11. Installation Steps / Performance Steps and Results –**

|  |  |
| --- | --- |
| Q | Implement C program to Convert an Infix expression to Postfix expression using stack ADT. |

Source Code :

//

// Program Statement: Write a program to convert an infix expression into its equivalent postfix notation.

//

#include <stdio.h>

#include <conio.h>

#include <ctype.h>

#include <string.h>

#define MAX 100

char st[MAX];

int top=-1;

void push(char st[], char);

char pop(char st[]);

void InfixtoPostfix(char source[], char target[]);

int getPriority(char);

int main()

{

char infix[100], postfix[100];

printf("\n Enter any infix expression : ");

gets(infix);

strcpy(postfix, "");

InfixtoPostfix(infix, postfix);

printf("\n The corresponding postfix expression is : ");

puts(postfix);

getch();

return 0;

}

void InfixtoPostfix(char source[], char target[])

{

int i=0, j=0;

char temp;

strcpy(target, "");

while(source[i]!='\0')

{

if(source[i]=='(')

{

push(st, source[i]);

i++;

}

else if(source[i] == ')')

{

while((top!=-1) && (st[top]!='('))

{

target[j] = pop(st);

j++;

}

if(top==-1)

{

printf("\n INCORRECT EXPRESSION");

exit(1);

}

temp = pop(st);//remove left parenthesis

i++;

}

else if(isdigit(source[i]) || isalpha(source[i]))

{

target[j] = source[i];

j++;

i++;

}

else if (source[i] == '+' || source[i] == '-' || source[i] == '\*' ||

source[i] == '/' || source[i] == '%')

{

while( (top!=-1) && (st[top]!= '(') && (getPriority(st[top]) > getPriority(source[i])))

{

target[j] = pop(st);

j++;

}

push(st, source[i]);

i++;

}

else

{

printf("\n INCORRECT ELEMENT IN EXPRESSION");

exit(1);

}

}

while((top!=-1) && (st[top]!='('))

{

target[j] = pop(st);

j++;

}

target[j]='\0';

}

int getPriority(char op)

{

if(op=='/' || op == '\*' || op=='%')

return 1;

else if(op=='+' || op=='-')

return 0;

}

void push(char st[], char val)

{

if(top==MAX-1)

printf("\n STACK OVERFLOW");

else

{

top++;

st[top]=val;

}

}

char pop(char st[]) {

char val = ' ';

if (top ==-1)

printf("\n STACK UNDERFLOW");

else

{

val = st[top];

top--;

}

return val;

}

Infix\_to\_Postfix.c

Open with ZIP Extractor

Department of CSE (AI&ML)