**Batch: A1 Roll No. 1911004**

**Experiment No. 2**

**Grade: AA / AB / BB / BC / CC / CD /DD**

|  |
| --- |
| **Title:**  Implementation of infix to postfix transformation and its evaluation using stack |

**Objective:** To understand the application of Last In First Out Data Structure “Stack” in practical applications such as converting the infix expression to postfix and postfix evaluation using either arrays or linked list.

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **CO** | **Outcome** |
| 1 | Explain the different structures used in problem solving. |

**Books/ Journals/ Websites referred:**

1. Data Structures A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A. Forouzan, second edition, CENGAGE learning.
2. Data Structures Using C & C++, Rajesh K. Shukla, Wiley- india.

**Abstract**:-

Different type of input expressions can be given as input for evaluation in some real mathematical models. General-Purpose programming languages like C, C++ do not have built-in data types or functions to convert the infix expressions and their evaluation directly. So, there by comes an important data structure Stack which is Last- In-First-Out (LIFO) concept, which is used for converting an infix expression to postfix or prefix expressions. Stacks are implemented using Arrays or Linked Lists. Computer also uses its own stack called System Stack.

**Related Theory: -**

Infix expression is the user inputted expression which may consist brackets or can be bracket-free. Infix is the one where operators are placed in between operands. 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.

➢ Postfix Expression is the one where all the operators are placed after the operands and it is a bracket-free expression. This makes the evaluation of expression easy and less complicated as we don’t have to follow operator precedence rules while evaluating it.

**Related Theory :**

Writing a mathematical expression in its normal form is called infix expression

🡺 7-((4+3)/(6-1))\*2

The infix expression can be converted to postfix by removing the parentheses-brackets [‘(’ & ‘)’] & write up operands and operators in a way that the operator is placed after the operands.

🡺 743+6/1—2\*

Similarly, the infix expression can be converted to prefix by removing the parentheses and writing the operands & operators in such a way that the operator is placed before the operands. By transforming the expression to post fix or prefix, and putting up a code in place for evaluating it we make it very efficient for the computer to process the expression, thus the expression is processed faster manner, thus it is efficient too. This saves much computational time.

**Diagram for Push, Pop operations on the stack:**



**WE PUSH ELEMENT ONTO STACK ON THE TOP & ALSO POP ELEMETS FROM TOP OF STACK**

**Algorithm:**

* **For Infix to Postfix Conversion**

1. **Start**
2. **Add ‘)’ to the end of the infix expression**
3. **we push ‘(‘ on to the stack**
4. **Now until each character in the infix expression is scanned we will repeat**
5. **IF we encounter a ‘(‘, push it on the stack**
6. **IF we encounter an operand ( a digit / a character), we add it postfix expression.**
7. **IF we encounter a ‘)’, then**

* **we repeatedly pop out elements from stack & add it to the postfix expression until we encounter a ‘(‘ .**
* **we would discard the ( ; [ remove ( from stack & wont add it to the postfix expression]**

1. **IF we encounter an operator , then**

* **we repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression that has the same precedence / a higher precedence than that.**
* **we push the operator on to the stack**

1. **we repeatedly pop out elements from the stack & add them to our postfix expression until the stack left is empty**
2. **End**

* **Evaluation of Postfix Expression:**

1. **Start**
2. **Add a ‘)’ to the end of the postfix expression ,**
3. **we scan up every element and repeat steps 3 and 4 until we encounter ‘)’**
4. **IF we encounter an operand, it is pushed on to the stack**

**IF we encounter an operator O , then**

* **Pop the top two elements from the stack as x & y**
* **b. Evaluate y O x i.e. x is the topmost element & y is the element before x.**
* **c. Push the result of evaluation on to the stack**

1. **the top most element is set the result of evaluation**
2. **End**

**Program Source Code**:

**#include <ctype.h>**

**#include <stdio.h>**

**#define SIZE 50**

**char s[SIZE];**

**int top = -1;**

**//Function to remove spaces from exp**

**void RemoveSpaces(char\* source) {**

**char\* i = source;**

**char\* j = source;**

**while(\*j != 0) {**

**\*i = \*j++;**

**if(\*i != ' ')**

**i++;**

**}**

**\*i = 0;**

**}**

**char pop() {**

**return (s[top--]);**

**}**

**void push(char element) {**

**s[++top] = element;**

**}**

**//precedence**

**int pr(char element) {**

**switch (element) {**

**case '#':**

**return 0;**

**case '(':**

**return 1;**

**case '+':**

**case '-':**

**return 2;**

**case '\*':**

**case '/':**

**return 4;**

**}**

**return -1;**

**}**

**void infix\_to\_postfix(char \*infix, char \*postfix) {**

**char ch, e;**

**int i = 0, k = 0;**

**RemoveSpaces(infix);**

**push('#');**

**while ((ch = infix[i++]) != '\n') {**

**if (ch == '(')**

**push(ch);**

**else if (isalnum(ch))**

**postfix[k++] = ch;**

**else if (ch == ')') {**

**while (s[top] != '(')**

**postfix[k++] = pop();**

**e = pop(); // Remove (**

**} else {**

**while (pr(s[top]) >= pr(ch))**

**postfix[k++] = pop();**

**push(ch);**

**}**

**}**

**while (s[top] != '#') /\* Pop from stack till empty \*/**

**postfix[k++] = pop();**

**postfix[k] = 0;**

**}**

**int eval\_postfix(char \*postfix) {**

**char ch;**

**int i = 0, op1, op2;**

**while((ch = postfix[i++]) != 0) {**

**if(isdigit(ch))**

**push(ch-'0');**

**else {**

**op2 = pop();**

**op1 = pop();**

**switch(ch) {**

**case '+' : push(op1+op2);**

**break;**

**case '-' : push(op1-op2);**

**break;**

**case '\*' : push(op1\*op2);**

**break;**

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

**break;**

**}**

**}**

**}**

**return s[top];**

**}**

**int main() {**

**char infix[50], postfix[50];**

**printf("\nEnter the infix expression: ");**

**fgets(infix, 50, stdin);**

**infix\_to\_postfix(infix, postfix);**

**printf("\nGiven Infix Expression: %s Postfix Expression: %s", infix, postfix);**

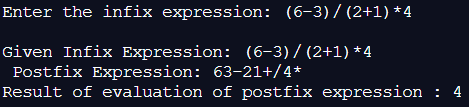
**top = -1;**

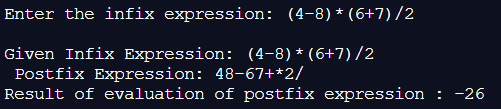
**printf("\nResult of evaluation of postfix expression : %d\n\n", eval\_postfix(postfix));**

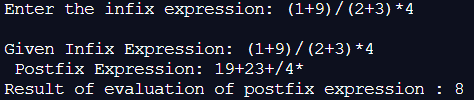
**return 0;**

**}//main() ends**

**Output screenshots:**







**Implementation Details:**

1. **Enlist all the Steps followed and various options explored**

* **To convert an infix expression to postfix,**

1. **we scan every element of the expression ,to find if an operand is encountered which it is directly moved into the post fix expression.**
2. **If an operator is encountered then the all the elements in the stack are popped one by one unless a ‘(‘ is encountered or an operator of lower priority is encountered.**
3. **If a “(“ is encountered we moved it into the stack.**
4. **If a ‘)’ is encountered then we pop all the elements of the stack one by one unless a ‘(‘ is encountered.**
5. **Then ‘(‘ is removed from the stack.**
6. **Thus the final expression obtained is the postfix expression.**
7. **To evaluate the expression we scan all the elements of the post fix expression , if an operator is encountered we move it into the stack.**
8. **If an operator is encountered then the operation is performed between the first 2 operands and the result is stored into the stack.**
9. **Lastly after scanning through all the elements & performing the corresponding operations left out in the stack is the needed result of the evaluation of the post expression.**
10. **Explain your program logic, classes and methods used.**

* **Functions created are**

void push(char element)

//This function pushes a character element into the stack thus increments the top of stack after performing the pushing operation

char pop()

//This function pops a character element at top of stack and stores the popped value in the postfix expression.

int pr(char element)

//This function finds the precedence of the symbols used

void infix\_to\_postfix(char \*infix, char \*postfix)

//This function converts the infix to postfix expression using

void RemoveSpaces(char\* source)

//This function removes all the extra spaces entered for better results of final evaluation

int eval\_postfix(char \*postfix)

//This method takes postfix expression and finds answer thus evaluates the expression

int main()

//way to execute the program and call all function

1. **Explain the Importance of the approach followed by you**

* **Converting the expression into postfix makes it easier for the computer to understand, thus the expression is processed at faster pace and this is done in more speedy, efficient manner. The computer can execute the code & evaluate expression of any length much faster than the normal conventional way, which makes it efficient , thus saves on computational time .Thus we can increase efficiency of the computer by converting the infix expression to postfix expression. Thus infix to postfix plays important role in arithmetic processes of computer .**

**Conclusion:-**

**We understood the concept of infix to postfix expression & implemented it in c thus got correct necessary output .**