**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**



**A Report on**

**COMP 202: Data Structures and Algorithms**

**Mini Project**

**Submitted by:**

Sajan Mahat (30)

Nripesh Karmacharya (21)

**Submitted to:**

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

**Submission Date:**

January 15, 2019

**Task:**  To write a program to convert infix expressions to postfix and to find the time complexity of the program

To convert infix expression to postfix expression, we used the stack data structure.

Note: Here we considered only {+, −,∗,/, () } operators.

**Implementation:**

Algorithm

Input: an infix expression

Output: an equivalent postfix expression

Steps:  
1. Scan the infix expression from left to right.  
2. If the scanned character is an operand, output it.  
3. Else,  
…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a ‘(‘, push it.  
…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)  
4. If the scanned character is an ‘(‘, push it to the stack.  
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.  
6. Repeat steps 2-6 until infix expression is scanned completely.  
7. Pop and output from the stack until it is not empty  
8. Print output

**C++ Source Codes**

The C++ source codes are submitted with the labels stack.h, stack.cpp and infixtopostfix.cpp.

The stack.h and stack.cpp handle the stack operations such as push, pop, peak, etc. The infixtopostfix.cpp includes the stack implementation and converts the infix expression input from the user to its equivalent postfix expression in the same manner as the algorithm above.

**Test Program** (Manually done according to the algorithm)

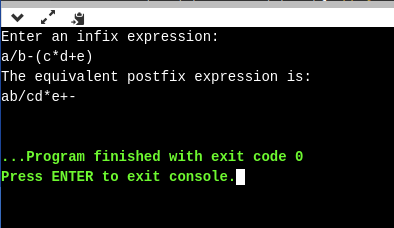
**Input**: a/b-(c\*d+e) : an infix expression

|  |  |  |
| --- | --- | --- |
| **Token** | **Stack** | **Postfix Expression** |
| a | { } | a |
| / | { / } | a |
| b | { / } | ab |
| - | { - } | ab/ |
| ( | { -, ( } | ab/ |
| c | {-, (} | ab/c |
| \* | { - , ( , \* } | ab/c |
| d | { - , ( , \* } | ab/cd |
| + | { - , ( , + } | ab/cd\* |
| e | { - , ( , +} | ab/cd\*e |
| ) | { - } | ab/cd\*e+ |
|  | { } | ab/cd\*e+- |

Note: The top of the stack is the rightmost operator.

**Expected Output**: ab/cd\*e+- : an equivalent postfix expression

**Output from the program:**



Hence, the implementation works correctly.

**Time Complexity of the Program**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Program** | **Steps** |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | int main(){  string infixexp,postfixexp;  cout<<"Enter an infix expression:"<<endl;  cin>>infixexp;  Stack s;  char c;  for(int i=0;i<infixexp.length();i++){ //let infixexp.length()=n  c=infixexp[i];  if(c=='('){  s.push(c);  }  else if(c==')') {    char p=s.pop();  while(p!='('){  postfixexp+=p;  p=s.pop();  }  }  else if(int(c)>41&&int(c)<48){  if(!s.isEmpty()){  char top=s.peak();  while(!s.isEmpty()&&(priority(c)<=priority(top))){  char token=s.pop();  postfixexp+=token;  top=s.peak();  }  }  s.push(c);  }  else{  postfixexp+=c;  }  }  while(!s.isEmpty()){  char token=s.pop();  postfixexp+=token;  }  cout<<"The equivalent postfix expression is:"<<endl<<postfixexp<<endl;  } | 0  0  1  1  0  0  n+1  n  n  c1  0  c1  0  c1  3c1  c1  2c1  0  0  c2  c2  c2  3c2  2c2  c2  2c2  0  0  c2  0  c3  c3  0  0  3  2  2  0  1  0 |
|  | TOTAL | Kn+r |

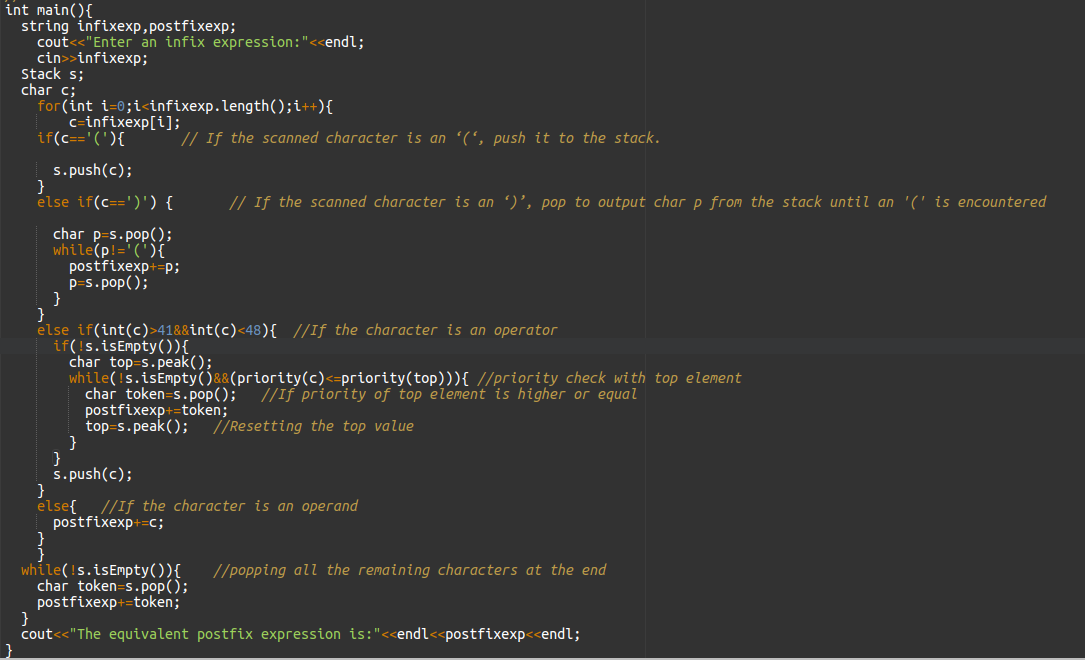
Hence, Time complexity or the total runtime of the program = O(n).

Here, c1 = number of steps for loop of ‘(‘ and ‘)’ , c2 = number of steps for loop of operators , c3 = number of steps for loop of operands, k = number of operators in the stack at the end and K and r are some constants. The number of times the program will run completely depends on the input infix expression, the numbers and positions of the operands and operators and so on. So, the value of the constants will vary according to the expression. However, the constants do not play a significant role here.

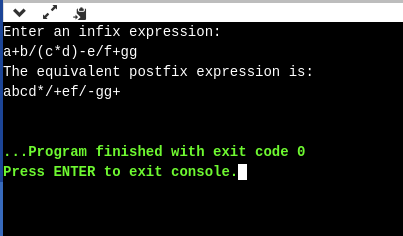
Simply, if we assume the length of the infix expression input from the user to be ‘n’, then there will be a single loop in the program accompanied by several other if and else statements. So, as the other constants would not matter when we have a loop for n times, the time complexity of the program would be O(n).

**Screenshots:**

The main() block;



**Output Sample**

****