Stack applications

Expression

An application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation: A+B

2. Prefix notation: +AB

3. Postfix notation: AB+

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as: A + B
Note that the operator '+' is written in between the operands A and B.

The prefix notation is a notation in which the operator(s) is written before the operands, it is also called **polish notation**. The same expression when written in prefix notation looks like: + A B
As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as **suffix notation** or **reverse polish notation**.
The above expression if written in postfix expression looks like:
A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add(A, B)
Note that the operator add (name of the function) precedes the operands A and B.

Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. In the preceding sections we will study the conversion of the expression from one notation to other.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in infix notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS (Order of Operations), and associativity.
Using infix notation, one cannot tell the order in which operators should be applied.
Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated

first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

Converting infix to postfix expression

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Problem 1. Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE +) F *] / G}
A + { [(BC +) (DE + F * +] / G} .

A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Problem 2. Give postfix form for (A + B) * C / D + E ^ A / B
Solution. Evaluation order is
[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]
[(AB +) C * D /] + [(EA ^) B /]
(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm
Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
2. If an operand is encountered, add it to Q.
3. If a left parenthesis is encountered, push it onto stack.
4. If an operator $\otimes$ is encountered, then:
   (a) Repeatedly pop from stack and add to Q each operator (on the top of stack), which has the same precedence as, or higher precedence than $\otimes$.
   (b) Add $\otimes$ to stack.
5. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from stack and add to Q (on the top of stack until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to stack.]
6. Exit.

Note. Special character $\otimes$ is used to symbolize any operator in P.
Consider the following arithmetic infix expression P

P = A + (B / C - (D * E ^ F) + G) * H)

Fig. 1 shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character scanned	Stack	Postfix Expression (Q)
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
/	(+ (/	A B
C	(+ (/	A B C
–	(+ (-	A B C /
(	(+ (- (	A B C /
D	(+ (- (	A B C / D
*	(+ (- (*	A B C / D
E	(+ (- (*	A B C / D E
^	(+ (- (* ^	A B C / D E
F	(+ (- (* ^	A B C / D E F
)	(+ (-	A B C / D E F ^ *
+	(+ (+	A B C / D E F ^ * -
G	(+ (+	A B C / D E F ^ * - G
)	(+	A B C / D E F ^ * - G +
*	(+ *	A B C / D E F ^ * - G +
H	(+ *	A B C / D E F ^ * - G + H
)		A B C / D E F ^ * - G + H * +

Fig. 1

H.W. Write a C++ program to implement the algorithm above (converting infix to postfix algorithm).

Evaluating postfix expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Scan P from left to right and repeat Steps 3 and 4 for each element of P.
2. If an operand is encountered, put it on STACK.
3. If an operator $\otimes$ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B $\otimes$ A.
   (c) Place the result on to the STACK.
4. Result equal to the top element on STACK.
5. Exit.

H.W. Write a C++ program to implement the algorithm above (Evaluating postfix expression).