An Application of Stacks and Queues

Algebraic Expressions (infix, postfix, and prefix):

This week we discuss a major application of stacks. Though it is one of the well-known applications, it is by no means the only one.

To proceed with the presentation of our problem and the development of algorithms and programs to solve the problem, it is necessary to provide some background.

Consider the sum of A and B. We think of applying the operator "+" to the operands A and B and express the sum as A+B. This particular representation is called infix. There are two alternative notations for expressing the sum of A and B using the symbols A, B, and +. These are

+AB prefix representation
AB+ postfix representation

The prefixes pre, post and in refer to the relative position of the operator with respect to the two operands. In prefix notation the operator precedes the two operands, in postfix notation the operator follows the two operands, and in infix notation the operator is between the two operands.

Consider the expression A+B*C, as written in standard infix notation. To evaluate A+B*C, one requires knowledge of which of the two operations, + or *, is to be performed first. In the case of + and * we know, from our knowledge of evaluating algebraic expressions, that multiplication is to be done before addition. Thus A+B*C is to be interpreted as A+(B*C) unless otherwise specified. We say that multiplication takes precedence over addition. Suppose that we would now like to rewrite A+B*C in postfix. Applying the rules of precedence, we first convert the portion of the expression that is evaluated first, namely the multiplication B*C. We do this conversion in stages. Converting B*C first to postfix, the expression A+B*C becomes, underline for emphasis, A+BC*. Next we convert the addition to obtain the final expression ABC*+ that is the postfix form of our original infix expression A+B*C.

The only rules to remember during the conversion process is that the operations with highest precedence are converted first and that after a portion of the expression has been converted to postfix it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses.

Consider the expression (A+B)*C in infix form. First convert the addition in (A+B)*C to obtain AB+*C. Then convert the multiplication in the expression AB+*C to get the final expression AB+C* in postfix form. In the example above, the addition was converted before the multiplication because of the parentheses. In going from (A+B)*C to AB+*C, A and B are the operands and + is the operator. In going from AB+*C to AB+C*, AB+ and C are the operands and * is the operator. The rules for converting from infix to postfix are simple provided that you know the order of precedence.

We consider four binary operations: addition(+), subtraction(-), multiplication(*), and division(/). For these binary operators the following is the order of precedence (highest to lowest):

multiplication /division
addition /subtraction

By using parentheses we can override the default precedence.

We consider some additional examples of converting from infix to postfix. Be sure that you understand each of these examples. We follow the convention that when unparenthesized operators of the same precedence are scanned, the order is assumed to be from left to right. Thus A+B+C means (A+B)+C.

infix	postfix
A+B	AB+
A-B+C	AB-C+
(A-B)+C	AB-C+
A-(B+C)	ABC+-
(A+B)*C/D	AB+C*D/
A+B*C/D	ABC*D/+

An important point, that you may have noticed, about the postfix expression is that it requires no parentheses. Let us consider the two expressions A+(B*C) and (A+B)*C. The parentheses in the expression A+(B*C) is superfluous; because by our convention A+B*C = A+(B*C). However, the parenthesis in (A+B)*C is necessary. The postfix forms of these expressions are

infix	postfix
A+(B*C)	ABC*+
(A+B)*C	AB+C*

There are no parentheses in either of the two transformed expressions. A careful look tells us that the order of the operators in the postfix expression determines the actual order of operations in evaluating the expression making the use of parentheses unnecessary. Indeed, an expression that is in the postfix form is much easier to evaluate as compared to an expression in the infix form.

Evaluating a Postfix Expression

Consider the following expression:

infix	postfix
(A+B)*C	AB+C*

Suppose that A=2, B=3, and C=4. Then from the infix form we know that (2+3)*4 = 20. How do we know for the postfix expression 2 3 + 4 * = 20?

We answer this question by developing a simple algorithm that evaluates expressions in postfix. Each operator in a postfix expression refers to the previous two operands in the expression. Suppose that each time we read an operand, we push it onto a stack. When we reach an operator, its operands will then be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them and push the result on the stack so that it will be available for use as operand of the next operator. The following algorithm evaluates an expression in postfix using this method.
Suppose that the postfix expression to be evaluated is available to us in a Queue object called postQ. The stack we described above is an object, called eval, of type Stack.

```
Stack<int> eval;
Queue<char> postQ;
// postQ contains the postfix expression to be evaluated.
char t;
int topNum, nextNum, answer;

while (postQ is not empty) {
     t = postQ.Front();
     postQ.DeQueue();
     if (t is a number token)
            eval.Push(t);
     else {                          // t is an operator token
           topNum = eval.Top();      // Get operand from stack
           eval.Pop();               // Remove operand from stack
           nextNum = eval.Top();     // Get operand from stack
           eval.Pop();               // Remove operand from stack

     switch (t) {
            case '+': answer = nextNum + topNum; break;
            case '-': answer = nextNum - topNum; break;
            case '*': answer = nextNum * topNum; break;
            case '/': answer = nextNum / topNum; break;
     }

     eval.Push(answer);
}

return eval.Top();
```

Let us now consider an example. Suppose that we are asked to evaluate the following expression in postfix:

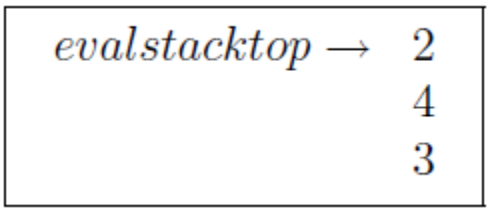
infix	postfix
(3+4/2)*(5*3-6)-8	3 4 2 / + 5 3 * 6 - * 8 -

From the infix we know (3+4/2)*(5*3-6)-8 = 37. Let us verify whether our algorithm on the

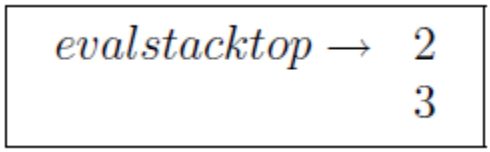
post fix expression 3 4 2 / + 5 3 * 6 - * 8 - yields the result 37.

The following stack contents show the sequence of operations:

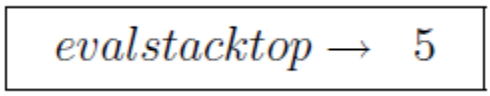
First we push the numbers 3, 4, and 2 on to the eval stack as shown below.



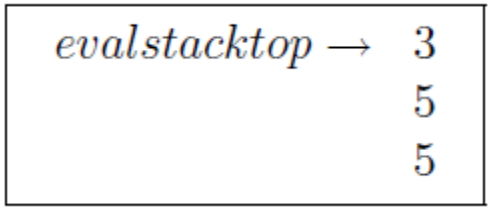
The next symbol we dequeue from the postQ is the division operator(/). So, we pop the first two operands from the stack and do the intended operation; that is (4/2). We then push the result 2 back on to the stack, so that the stack contents now are as shown below.



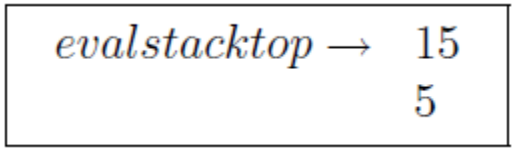
Our next symbol dequeued from postQ is the addition(+) operator. So we pop the first two operands from the stack and do the intended operation; that is (3+2). We then push the result 5 back on to the stack, so that the stack contents now are as shown below.



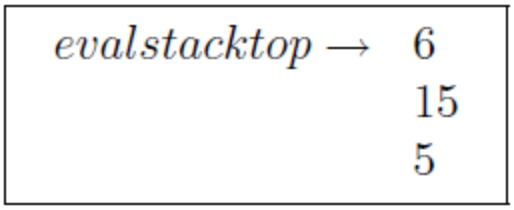
Next we push 5 and 3 to the stack.



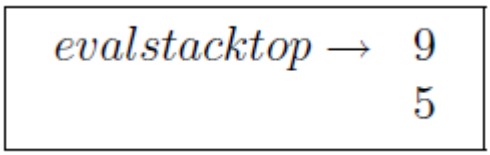
The next symbol dequeued from postQ is the multiplication(*) operator. So we pop the first two operands from the stack and do the operation (5*3). We then push the result 15 back on to the stack, so that the stack contents now are as shown below.



Next we push 6 on the stack.

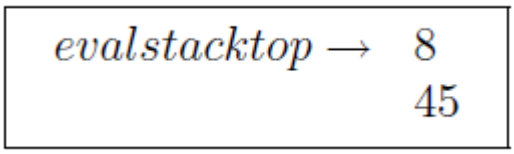


Our next operator is (-). We pop the operands as before and push the result (15-6) back on to the stack.



Our next operator is (*). We pop the first two operands from the stack and push the result (5*9) back on to the stack.

Next we dequeuer 8 and pust it on to the stack



Finally we dequeue the (-) operator from postQ and pop the two operands from the stack. The result is (45-8) = 37. This value is pushed on to the stack.

At this stage the postQ is empty and the eval stack has exactly one value which is 37.

Converting Infix Expression to Postfix

So far discussed an algorithm to evaluate a postfix expression. Though we discussed a method for converting an infix expression to postfix informally, we have not considered an algorithm for doing so. If we develop such an algorithm then we can take any infix expression and evaluate it by first converting it to the postfix form and then evaluating the postfix expression.

Consider the two infix expressions A+B*C and (A+B)*C and their respective postfix versions ABC*+ and AB+C*. Note that in each case the order of the operands is the same as the order of the operands in the original infix expressions. in scanning the first expression, A+B*C, the first operand A can be immediately inserted into the postfix expression. However the + symbol cannot be inserted until after its second operand, which has not yet been scanned, is inserted. Therefore, we must store this operator until it

can be inserted in its proper position. When the operand B is scanned, it is inserted immediately after A. Now, however, two operands have been scanned. What prevents the symbol + from being retrieved and inserted? The answer is, of course, the * symbol, which follows and has precedence over +. In the case of the second expression the closing parenthesis indicates that the + operation should be performed first. Remember that in postfix, unlike infix, the operator that appears earlier in the expression is the one that is applied first.

Now we are ready to present an algorithm to convert an infix expression into a postfix expression.

Conversion from Infix to Postfix

```
Stack<char> opStack;
Queue<char> postQ, infixQ;
// infixQ contains the infix expression to be converted.
// postQ contains the converted postfix expression.
char t;
while (infixQ is not empty)
     t = infixQ.Front(); infix.DeQueue();

     if (t is a number token)             // (1)
            postQ.EnQueue(t);
     else if (opStack.IsEmpty())          // (2)
            opStack.Push(t);
     else if (t is a left paren token)    // (3)
            opStack.Push(t);
     else if (t is a right paren token) { // (4)
            while (opStack.Top() is not a left paren) {
                  postQ.EnQueue(opStack.Top());
                  opStack.Pop();
            }
            opStack.Pop(); // discard a left paren from stack
     }
     else {                               // (5)
            while ( opStack is not empty and opStack.Top() != '('
                  and precedence of t <= precedence of opStack.Top()) {
            postQ.EnQueue(opStack.Top());
            opStack.Pop();
       }
     opStack.Push(t);
     }
}
// Now there are no tokens left in infixQ, so transfer remaining
operators.
while (!opStack.IsEmpty()) {              // (6)
     postQ.EnQueue(opStack.Top());
     opStack.Pop();
}
```

The basis behind the conversion algorithm is that we must output the operators in the

order in which they are to be executed. For this purpose we make use of the stack opStack to keep track of the operators encountered previously. If the operator under consideration is of greater precedence than the one on top of the stack this new operator is pushed onto the stack. This means that when all the elements in the stack are finally popped, this new operator will precede the former top in the postfix expression. If, on the other hand, when the precedence of the current operator is less than that of the top of the opStack, the operator at the top of the stack should be applied first. Therefore, the top of the opStack is popped and the current operator is compared with the new top, and so on (see line 5 of the algorithm). If our infix expression has parentheses, we have to override the order of operations. Thus when a left parenthesis is scanned, it is pushed on the opStack. When the corresponding right parenthesis is found, all the operators between the two parentheses are placed on the postQ, because they are to be applied before any operators appearing after the parentheses (see line 4 of the algorithm).