**Data Structures**

**Topics Covered: Expression Conversions and Evaluations using stack**

There are three types of notations for expressions. The standard form is known as the

infix form. The other two are postfix and prefix forms.

**Infix:** operator is between operands A + B

**Postfix:** operator follows operands A B +

**Prefix:** operator precedes operands + A B

Note that all infix expressions cannot be evaluated by using the left to right order of the

operators inside the expression. However, the operators in a postfix expression are ALWAYS

in the correct evaluation order. Thus, evaluation of an infix expression is done in two steps.

The first step is to convert it into its equivalent postfix expression. The second step involves

evaluation of the postfix expression. We shall see in this section, how stacks are useful in

carrying out both the steps. Let us first examine the basic process of infix to postfix

conversion.

**Infix to postfix conversion:**

a + b \* c Infix form

(Precedence of \* is higher than of +)

a + (b \* c) convert the multiplication

a + ( b c \* ) convert the addition

a (b c \* ) + Remove parentheses

a b c \* + Postfix form

***Note that there is no need of parentheses in postfix forms.***

**Example 2:**

( A + B ) \* C Infix form

( A B + ) \* C Convert the addition

(A B + ) C \* Convert multiplication

A B + C \* Postfix form

No need of parenthesis anywhere

**• More examples**

**Infix Postfix**

(a + b) \* (c – d) a b + c d - \*

a – b / (c + d \* e) a b c d e \* + / -

((a + b) \* c – (d – e))/ (f + g) a b + c \* d e - - f g + /

**Order of precedence for operators:**

multiplication (\*) and division (/)

addition (+) and subtraction (-)

The association is assumed to be left to right.

i.e., a + b + c = (a+b) + c = ab+c+

**Evaluating a Postfix Expression**

We can evaluate a postfix expression using a stack. Each operator in a postfix string

corresponds to the previous two operands. Each time we read an operand we push it onto a

stack.

When we reach an operator its associated operands (the top two elements on the stack) are

popped out from the stack.

We then perform the indicated operation on them and push the result on top of the stack so

that it will be available for use as one of the operands for the next operator.

The following example shows how a postfix expression can be evaluated using a stack.

**Algorithm:**

**While (there are more characters in the input)  
 {**

**Read next symbol ch in the given postfix expression.**

**If ch is an operand put it in the stack.**

**If ch is an operator i.e \*, - , + , /**

**{**

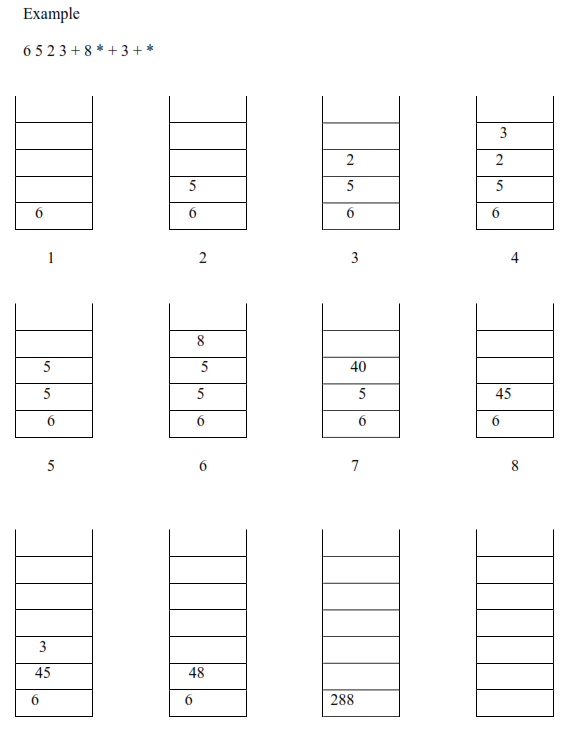
**Pop the top two operands and apply the operation on them**

**based on the symbol read and store the result back on the top   
of stack to be use in the next operation.**

**}**

**}**

**Pop remaining items in the stack to output which is the final result of the   
 postfix expression.**



The process stops when there is no more operator left in the string. The result of

evaluating the expression is obtained just by popping off the single element. More

examples will be done in the lecture and recitation labs.

**Converting an Infix Expression to Postfix**

A stack can also be used to convert an infix expression in standard form into postfix form.

We shall assume that the expression is a legal one (i.e. it is possible to evaluate it). When an

operand is read, it will be placed on output list (printed out straight away). The operators are

pushed on a stack. However, if the priority of the top operator in the stack is higher than the

operator being read, then it will be put on output list, and the new operator pushed on to the

stack. The priority is assigned as follows.

Priority

1. ( Left parenthesis in the expression

2. \* /

3. + –

4. ( Left parenthesis inside the stack

The left parenthesis has the highest priority when it is read from the expression, but once it is

on the stack, it assumes the lowest priority.

To start with, the stack is empty. The infix expression is read from left to right. If the

character is an OPERAND; it is not put on the stack. It is simply printed out as part of the

post fix expression.

The stack stores only the OPERATORS. The first operator is pushed on the stack. For all

subsequent operators, priority of the incoming operator will be compared with the priority of

the operator at the top of the stack.

If the priority of the incoming-operator is higher than the priority of topmost operator-on-the stack, it will be pushed on the stack.

if the priority of the incoming-operator is same or lower than the priority of the operator at

the top of the stack, then the operator at top of the stack will be popped and printed on the

output expression.

The process is repeated if the priority of the incoming-operator is still same or lower than the

next operator-in-the stack.

When a left parenthesis is encountered in the expression it is immediately pushed on the

stack, as it has the highest priority. However, once it is inside the stack, all other operators

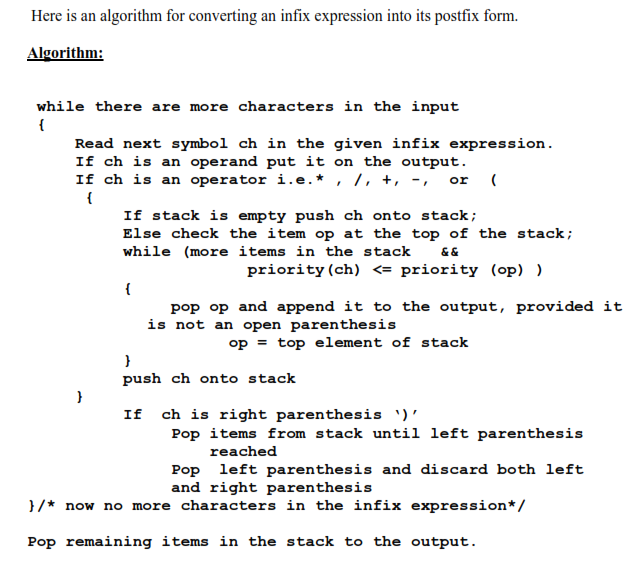
are pushed on top of it, as its inside-stack priority is lowest.

When a right parenthesis is encountered, all operators up to the left parenthesis are popped

from the stack and printed out. The left and right parentheses will be discarded. When all

characters from the input infix expression have been read, the operators remaining inside the

stack, are printed out in the order in which they are popped.



**Sample Run:**

