Application of Stack

Unit 2.2

Infix, Post, Prefix: Introduction

- **Expression**: a number of operands or data items combined using several operators

  Example: A+B. Here + is an **operator** and A and B are called **operands**

- There are three types of notation to represent expression
  - Infix Notation
    - Operator is written in between of the operands. Eg. A+B
  - Prefix Notation
    - Operator is written before the operands. Eg. +AB
  - Postfix Notation
    - Operator is written after the operands. Eg AB+

What is BODMAS??

- Operator Precedence
  - Exponential operator ^ Highest Precedence
  - Multiplication/Division *, / Next Precedence
  - Addition / Substraction +,- Latest Precedence

Converting Infix to Postfix

• A+B*C	Infix Form
• A+(B*C)	Parenthesized Expression
• A+(BC*)	Convert the multiplication
• A(BC*)+	Convert the addition
• ABC*+	Postfix Form

Rules for infix to postfix

- Parenthesize the expression starting from left to right.
- The operands associated with operator having higher precedence are first parenthesized.
- The sub-expression which has been converted into postfix is to be treated as single operand
- Once the expression is converted to postfix form remove the parenthesis.

Ex: Give postfix form for A+[(B+C)+(D+E)*F]/G

- Solution:

Ex: Give postfix form for A+[(B+C)+(D+E)*F]/G

- Solution:
- A+[(B+C)+(D+E)*F]/G
- A+[(BC+)+(DE+)*F]/G
- A+[(BC+)+(DE+F*)]/G
- A+[BC+DE+F*+]/G
- A+[BC+DE+F*+G/]
- ABC+DE+F*+G/+

EX. Convert to Postfix A*B+C/D

- If two operators with equal precedence, the operator that is encountered first while evaluating from left to right is parenthesized first.

(A*B)+C/D

(AB*)+C/D

(AB*)+(C/D)

(AB*)+(CD/)

AB*CD/+ Postfix Expression

(A*B)+C/D
(AB*)+C/D
(T)+C/D :substitute AB* with T
(T)+(C/D)
(T)+(CD/) : substitute CD/ with P
(T)+(P)
TP+
AB*CD/+ :Postfix Expression

Some Examples

1. A*B+C
2. A+B/C-D
3. (A+B) / (C-D)
4. (A+B)*C/D
5. (A+B)*C/D+E^F/G

Algorithm to convert infix to postfix notation

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack top and poststack top respectively.

1. Scan one character at a time of an infix expression from left to right
2. opstack=the empty stack
3. Repeat till there is data in infix expression
   - 3.1 if scanned character is '(' then push it to opstack
   - 3.2 if scanned character is operand then push it to poststack
   - 3.3 if scanned character is operator then
     - if(opstack!=-1)
     - while(precedence (opstack[otos])>precedence(scan character)) then
       - pop and push it into poststack
     - otherwise
       - push into opstack
   - 3.4 if scanned character is ')' then
     - pop and push into poststack until '(' is not found and ignore both symbols
4. pop and push into poststack until opstack is not empty.
5. return

((A-(B+C))*D)$(E+F)

Trace of Conversion Algorithm

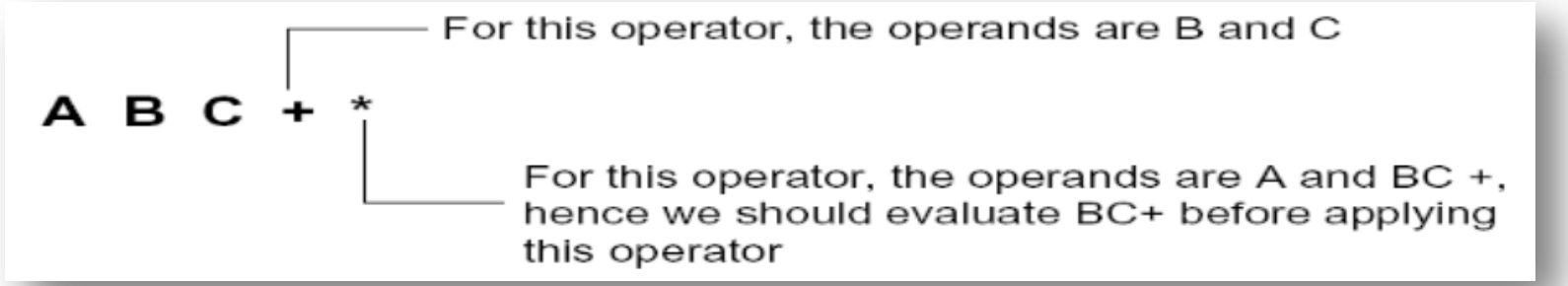
((A-(B+C))*D)$(E+F)

Scan character	Poststack	opstack
(	……	(
(	……	((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	…….
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
……	ABC+-D*EF+$ (postfix)	…….

Evaluating the Postfix expression

Consider an example
3 4 5 * +
=3 20 +
=23 (answer)

- Each operator in a postfix expression refers to the previous two operands in the expression.



- procedure:
  - Each time we read an operand we push it onto a stack.
  - When we reach an operator, its operands will 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 an operand of the next operator.

Evaluate following postfix algorithm

- 6 2 3 + - 3 8 2 / + * 2 $ 3 +

6 2 3 + - 3 8 2 / + * 2 $ 3 +

symb	*opnd1*	*opnd2*	*value*	*opndstk*
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
−	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Algorithm to evaluate the postfix expression

Use **only one stack** called vstack(value stack)

- 1. Scan one character at a time from left to right of given postfix expression
  - 1.1 if scanned symbol is operand then read its corresponding value and push it into vstack
  - 1.2 if scanned symbol is operator then
    - pop and place into op2
    - pop and place into op1
    - compute result according to given operator and push result into vstack
- 2. pop and display which is required value of the given postfix expression
- 3. return

Converting an Infix expression to Prefix expression

- The precedence rule for converting from an expression from **infix to prefix are identical.**
- Only changes from postfix conversion is that the **operator is placed before the operands** rather than after them.
- The prefix of A+B-C is …….

Find Prefix of: A $ B * C – D + E / F / (G + H)

Find Prefix of: A $ B * C – D + E / F / (G + H)

- A $ B * C – D + E / F / (G + H)

A $ B * C – D + E / F /(+GH)

$AB * C – D + E / F /(+GH)

*$ABC – D + E / F / (+GH)

*$ABC – D + (/EF) / (+GH)

*$ABC – D + //EF+GH

(-*$ABCD) + (//EF+GH)

+-*$ABCD//EF+GH which is in prefix form.

Algorithm: Infix to Prefix

Steps to convert infix to prefix expression:

1. Scan the valid infix expression from right to left.
2. Initialize the empty character stack and empty prefix string.
3. If the scanned character is an operand, add it to prefix string.
4. If the scanned character is closing parenthesis, push it to the stack.
5. If the scanned character is opening parenthesis, pop all the characters of the stack and concatenate to the end of the prefix string until closing parenthesis occurs.
6. Pop the closing parenthesis.
7. If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack. If top element of the Stack has higher precedence over the scanned character, pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.
8. Repeat steps 3 to 7 until all characters are scanned.
9. If the stack is empty, reverse and return the prefix string.
10. Else, pop the elements of the stack and add it to prefix string, and then reverse and return the prefix string.

Example:

Suppose if we want to convert the following infix expression to prefix expression then,

(a + (b * c) / (d – e)) = +a/*bc-de

NOTE: scan the infix string in reverse order.

SYMBOL	PREFIX	OPSTACK
)	Empty	)
)	Empty	))
e	e	))
-	e	))-
d	de	))-
(	-de	)
/	-de	)/
)	-de	)/)
c	c-de	)/)
*	c-de	)/)*
b	bc-de	)/)*
(	*bc-de	)/
+	/*bc-de	)+
a	a/*bc-de	)+
(	+a/*bc-de	Empty

Evaluating the Prefix Expression

- To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms.
- One stack is used to store operators and another is used to store the operands.

- Consider an example

+ 5 *3 2 prefix expression
= +5 6
=11

Evaluating the Prefix Expression

- Let ‘operandstk’ be an empty stack then the algorithm to evaluate prefix expression is:

1. Start
2. Input a prefix expression in the ‘prefix’ string.
3. for each last character ‘ch’ of ‘prefix’ string

```
        if(‘ch’ is an operand)
                push ‘ch’ in ‘operandstk’
        else
        {
                opnd2= pop an operand from ‘operandstk’
                opnd1= pop an operand from ‘operandstk’
                value= result of applying ‘ch’ in ‘opnd2’ and ‘opnd1’
                push ‘value’ to ‘operandstk’
        }
```

4. pop the top element of the stack ‘operandstk’ which is the required result.
5. Exit

TRACING THE ALGORITHM:

prefix string: + + 2 * 3 2 / 10 2

ch	opnd1	opnd2	value	operandstk
2				2
10				2, 10
/	2	10	5	5
2	2	10	5	5,2
3	2	10	5	5, 2, 3
*	2	3	6	5,6
2	2	3	6	5, 6, 2
+	6	2	8	5, 8
+	5	8	13	13