Data Structures

IT 205

Dr. Manish Khare

Lecture – 10
19-Jan-2018

Infix to Postfix Conversion

- Procedure for Postfix Conversion
  - Scan the Infix string from left to right.
  - Initialize an empty stack.
  - If the scanned character is an operand, add it to the Postfix string.
  - If the scanned character is an operator and if the stack is empty push the character to stack.
  - 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 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.
  - Repeat 4 and 5 steps till all the characters are scanned.
  - After all characters are scanned, we have to add any character that the stack may have to the Postfix string.
  - If stack is not empty add top Stack to Postfix string and Pop the stack.
  - Repeat this step as long as stack is not empty.

Algorithm for Postfix Conversion

```
S:stack
while(more tokens)
  x<=next token
    if(x == operand)
      print x
    else
      while(precedence(x)<=precedence(top(s)))
        print(pop(s))
    push(s,x)
while(! empty (s))
  print(pop(s))
```

Conversion To Postfix

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

Input	Stack	Output
A+(B*C-(D/E-F)*G)*H	Empty	-
+(B*C-(D/E-F)*G)*H	Empty	A
(B*C-(D/E-F)*G)*H	+	A
B*C-(D/E-F)*G)*H	+(	A
*C-(D/E-F)*G)*H	+(	AB
C-(D/E-F)*G)*H	+(*	AB
-(D/E-F)*G)*H	+(*	ABC
(D/E-F)*G)*H	+(-	ABC*
D/E-F)*G)*H	+(-(	ABC*

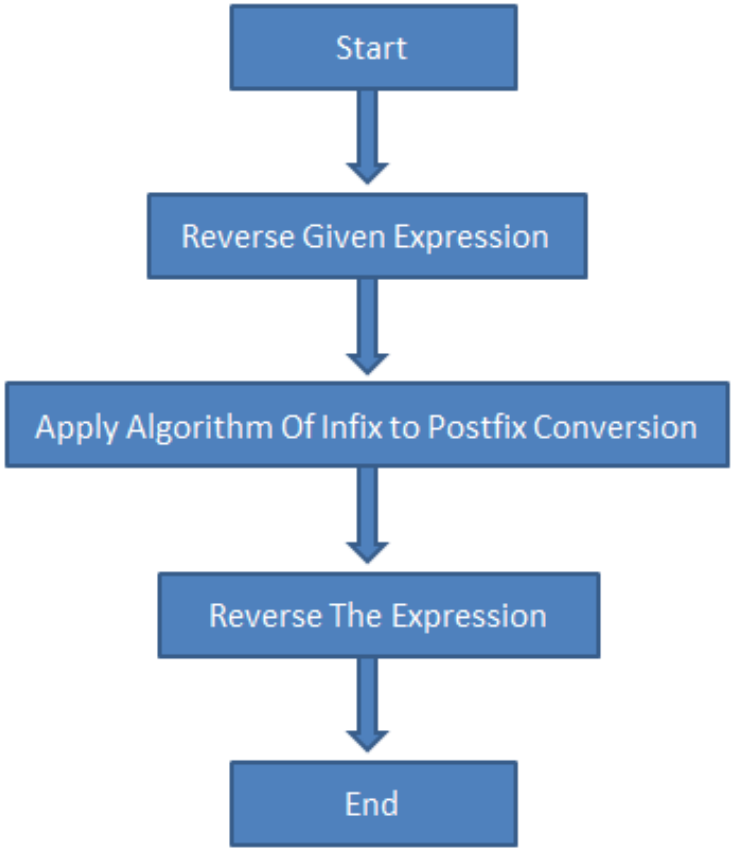
/E-F)*G)*H	+(-(	ABC*D
E-F)*G)*H	+(-(/	ABC*D
-F)*G)*H	+(-(/	ABC*DE
F)*G)*H	+(-(-	ABC*DE/
F)*G)*H	+(-(-	ABC*DE/
)*G)*H	+(-(-	ABC*DE/F
*G)*H	+(-	ABC*DE/F-
G)*H	+(-*	ABC*DE/F-
)*H	+(-*	ABC*DE/F-G
*H	+	ABC*DE/F-G*-
H	+*	ABC*DE/F-G*-
End	+*	ABC*DE/F-G*-H
End	Empty	ABC*DE/F-G*-H*+

(A – (B / C + (D % E * F) / G)* H)

Infix Character Scanned	Stack	Postfix Expression
	(	
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 * –

Infix to Prefix Conversion

- Algorithm of Infix to Prefix



Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
Step 2: Obtain the postfix expression of the infix expression obtained in Step 1.
Step 3: Reverse the postfix expression to get the prefix expression

- Infix to prefix conversion

- Expression = **(A+B^C)*D+E^5**

-

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form.

Expression	Stack	Output
5^E+D*(C^B+A)	Empty	-
^E+D*(C^B+A)	Empty	5
E+D*(C^B+A)	^	5
+D*(C^B+A)	^	5E
D*(C^B+A)	+	5E^
*(C^B+A)	+	5E^D
(C^B+A)	+*	5E^D
C^B+A)	+*(	5E^D
^B+A)	+*(	5E^DC

B+A)	+*(^	5E^DC
+A)	+*(^	5E^DCB
A)	+*(+	5E^DCB^
)	+*(+	5E^DCB^A
End	+*	5E^DCB^A+
End	Empty	5E^DCB^A+*+

Step 4. Reverse the expression.

+*+A^BCD^E5

Postfix to Infix Conversion

Algorithm of Postfix to Infix

- 1.While there are input symbol left
- 2. Read the next symbol from input.
- 3. If the symbol is an operand
- Push it onto the stack.
- 4. Otherwise, the symbol is an operator.
- 5. If there are fewer than 2 values on the stack
- Show Error /* input not sufficient values in the expression */
- 6. Else
- Pop the top 2 values from the stack.
- Put the operator, with the values as arguments and form a string.
- Encapsulate the resulted string with parenthesis.
- Push the resulted string back to stack.
- 7. If there is only one value in the stack
- That value in the stack is the desired infix string.
- 8. If there are more values in the stack
- Show Error /* The user input has too many values */

abc-+de-fg-h+/*

Expression	Stack
abc-+de-fg-h+/*	NuLL
bc-+de-fg-h+/*	"a”
c-+de-fg-h+/*	"b” "a"
-+de-fg-h+/*	"c" "b" "a"
+de-fg-h+/*	"b - c" "a"
de-fg-h+/*	"a+b-c"

e-fg-h+/*	"d" "a+b-c"
-fg-h+/*	"e" "d" "a+b-c"
fg-h+/*	"d - e" "a+b-c"
g-h+/*	"f" "d - e" "a+b-c"
-h+/*	"g" "f" "d - e" "a+b-c"
h+/*	"f-g" "d - e" "a+b-c"

+/*	"h" "f-g" "d - e" "a+b-c"
/*	"f-g+h" "d - e" "a+b-c"
*	"(d-e)/(f-g-h)" "a+b-c"
Null	"(a+b-c)*(d-e)/(f-g+h)"

Prefix to Infix Conversion

Algorithm of Prefix to Infix

- This algorithm is a non-tail recursive method.
- 1. The reversed input string is completely pushed into a stack.
- prefixToInfix(stack)
- 2. IF stack is not empty
- a. Temp -->pop the stack
- b. IF temp is a operator
- Write a opening parenthesis to output
- prefixToInfix(stack)
- Write temp to output
- prefixToInfix(stack)
- Write a closing parenthesis to output
- c. ELSE IF temp is a space -->prefixToInfix(stack)
- d. ELSE
- Write temp to output
- IF stack.top NOT EQUAL to space -->prefixToInfix(stack)

*+a-bc/-de+-fgh

Expression	Stack
*+a-bc/-de+-fgh	NuLL
*+a-bc/-de+-fg	"h"
*+a-bc/-de+-f	"g" "h"
*+a-bc/-de+-	"f" "g" "h"
*+a-bc/-de+	"f - g" "h"
*+a-bc/-de	"f-g+h"

*+a-bc/-d	"e" "f-g+h"
*+a-bc/-	"d" "e" "f-g+h"
*+a-bc/	"d - e" "f-g+h"
*+a-bc	"(d-e)/(f-g+h)"
*+a-b	"c" "(d-e)/(f-g+h)"

*+a-	"b" "c" "(d-e)/(f-g+h)"
*+a	"b-c" "(d-e)/(f-g+h)"
*+	"a" "b-c" "(d-e)/(f-g+h)"
*	"a+b-c" "(d-e)/(f-g+h)"
End	"(a+b-c)*(d-e)/(f-g+h)"

Postfix Expression Evaluation

- A postfix expression can be evaluated using the Stack data structure.
- To evaluate a postfix expression using Stack data structure we can use the following steps...
  - Read all the symbols one by one from left to right in the given Postfix Expression
  - If the reading symbol is operand, then push it on to the Stack.
  - If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
  - Finally! perform a pop operation and display the popped value as final result.

- Consider the infix expression given as **9 – ((3 * 4) + 8) / 4**.

- The infix expression **9 – ((3 * 4) + 8) / 4** can be written as **9 3 4 * 8 + 4 / –** using postfix notation.

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

Prefix Expression Evaluation

- The algorithm for evaluating a prefix expression is as follows:
  - Accept a prefix string from the user
  - Repeat until all the characters in the prefix expression have been scanned
    - Scan the prefix expression from right, one character at a time.
    - If the scanned character is an operand, push it on the operand stack
    - If the scanned character is an operator, then
      - (i). Pop two values from the operand stack
      - (ii). Apply the operator on the popped operands
      - (iii). Push the result on the operand stack
  - End

- **For example, consider the prefix expression + – 9 2 7 * 8 / 4 12.**

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

Reversing a List

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Implementing Parentheses Checker

- Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid. Look at the program below which traverses an algebraic expression to check for its validity.

Algorithm:

1) Declare a character stack S.

2) Now traverse the expression string exp.

a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.

b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

Recursion

- A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases. They are
  - ***Base case***, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - ***Recursive case***, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

- To understand recursive functions, let us take an example of calculating factorial of a number.
- To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.
- In other words, n! = n x (n–1)!
- Let us say we need to find the value of 5!
- 5! = 5 x 4 x 3 x 2 x 1 = 120
- This can be written as 5! = 5 x 4!, where 4!= 4 x 3!
- Therefore, 5! = 5 x 4 x 3!
- Similarly, we can also write 5! = 5 x 4 x 3 x 2!
- Expanding further 5! = 5 x 4 x 3 x 2 x 1!
- We know, 1! = 1

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

- every recursive function must have a base case and a recursive case. For the factorial function

  - **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.

  - **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)

- From the example, let us analyze the steps of a recursive program.
  - *Step 1:* Specify the base case which will stop the function from making a call to itself.
  - *Step 2:* Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.
  - *Step 3:* Divide the problem into smaller or simpler sub-problems.
  - *Step 4:* Call the function from each sub-problem.
  - *Step 5:* Combine the results of the sub-problems.
  - *Step 6:* Return the result of the entire problem.

Examples of Recursion

- Greatest Common Divisor
- Finding Exponents
- The Fibonacci Series

Types of Recursion

- Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on
  - whether the function calls itself directly or indirectly (*direct or indirect recursion*),

Direct Recursion

- A function is said to be *directly* recursive if it explicitly calls itself. For example, consider the code shown in following figure. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Indirect Recursion

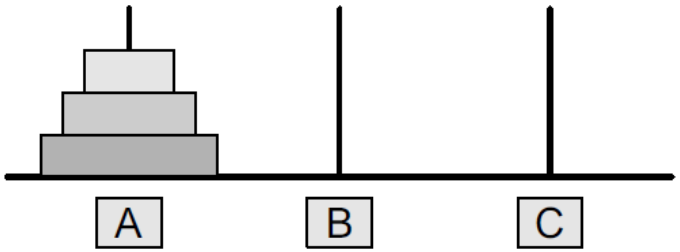
- A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

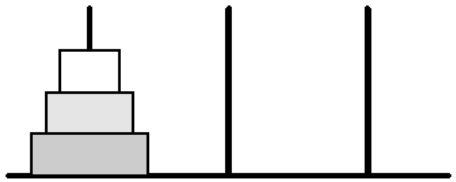
```
int Funcl (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
      return Func1(x-1);
}
```

Tower of Hanoi

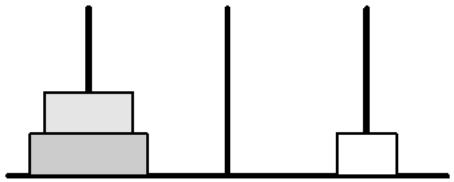
- The tower of Hanoi is one of the main applications of recursion. It says, ‘if you can solve n–1 cases, then you can easily solve the nth case’.

Tower of Hanoi

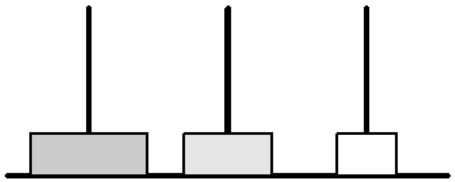




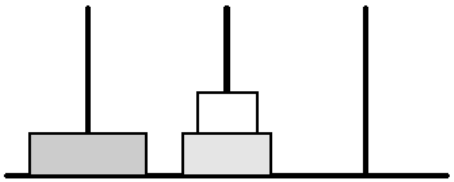
(Step 1)



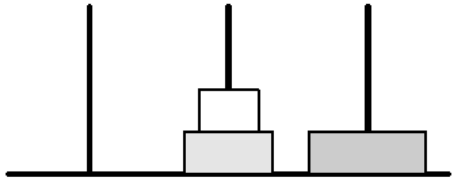
(Step 2)



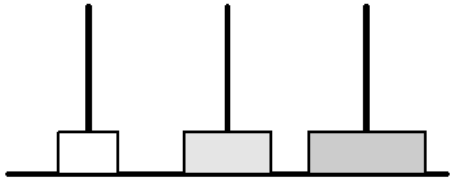
(Step 3)



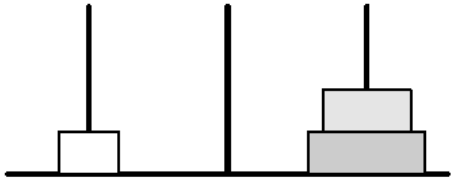
(Step 4)



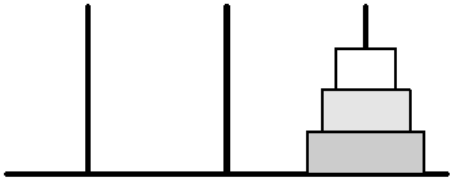
(Step 5)



(Step 6)



(Step 7)



(Step 8)

Advantage of Recursion

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Recursion follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Advantages/Drawback of recursion

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive program in midstream can be a very slow process.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly while using global variables.