Applications of Stack

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching: Algorithm

Algorithm PARAN_MATCH(X, n)
Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in X match
[Initialize Stack S]
TOP← -1
[Scan through array X and check]
for i=0 ← n−1 **do**
    if $X[i]$ is an opening grouping symbol **then**
        push(S, $X[i]$)
    else if $X[i]$ is a closing grouping symbol **then**
        if isEmpty(S) **then**
            return false {nothing to match with}
        if pop(S) does not match the type of $X[i]$ **then**
            return false {wrong type}
[Check if every symbol matched or not]
if isEmpty(S) **then**
    return true {every symbol matched}
else
    return false {some symbols were never matched}

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

HTML Tag Matching: Algorithm

Algorithm TAG_MATCH(FILE)
Input: An html file FILE containing valid tags and related contents. NEXTTAG is a function that returns next tag from the input file. EOF is the function that returns true if end of file is reached.
Output: **true** if and only if all the tags in FILE match
[Initialize Stack S]
TOP← -1
[Scan through FILE and check]
Repeat while NOT(EOF(FILE))
    TAG← NEXTTAG(FILE)
    if *TAG* is an opening tag **then**
        push(S, TAG)
    else if *TAG* is a closing tag **then**
        if isEmpty(S) **then**
            return false {nothing to match with}
        if pop(S) does not match the type TAG **then**
            return false {wrong type}
[Check if every tags matched or not]
if isEmpty(S) **then**
    return true {every symbol matched}
else
    return false {some tags were never matched}

Recognizing strings in a language

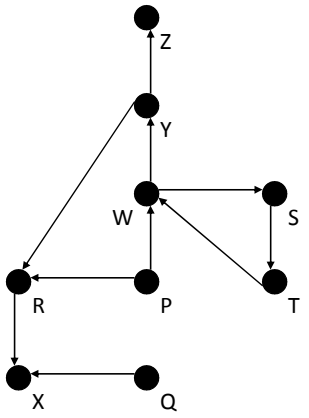
- Consider
  $L = \{ s\$s' : s$ is possibly empty string of characters other than $\$$, s' = reverse(s)$\}$
- Can you write an algorithm that recognizes all strings accepted by L ?

Recognizing strings in a language L: Algorithm

```
Algorithm RECOGNIZE(STRING)
Input: An input string STRING containing a NULL in its rightmost character position.
Output: true if and only if STRING matches with rules specified in L
[Initialize Stack S by placing '$' on top ]
TOP← -1
push(S,'$')
i←0
[Scan through STRING and stack symbols until '$' is encountered]
Repeat while STRING[i] != '$'
        if STRING[i] == NULL then
                return false {Invalid string}
        else
                push(S, STRING[i])
                i←i+1
[Scan characters following '$', compare them to characters in stack]
i←i+1
Repeat while S[TOP] != '$'
        if S[TOP]!= STRING[i] then
                        return false {not accepted}
        else
                pop(S)
                i← i+1
 [Check if all characters matched or not]
if STRING[i] = NULL then
        return true {string accepted}
else
        return false {not accepted}
```

Finding a Path

- Consider the following graph of flights



: city (represented as C)

$C_1 \rightarrow C_2$: flight from city C_1 to city C_2

W → S flight goes from W to S

Finding a Path

- If it exists, we can find a path from any city C_1 to another city C_2 using a stack
  - place the starting city on the bottom of the stack
    - mark it as visited
    - pick any arbitrary arrow out of the city
      - city cannot be marked as visited
    - place that city on the stack
      - also mark it as visited
    - if that's the destination, we're done
    - otherwise, pick an arrow out of the city currently at
      - next city must not have been visited before
      - if there are no legitimate arrows out, pop it off the stack and go back to the previous city
    - repeat this process until the destination is found or all the cities have been visited

Example

- Want to go from P to Y
  - push P on the stack and mark it as visited
  - pick R as the next city to visit (random select)
    - push it on the stack and mark it as visited
  - pick X as the next city to visit (only choice)
    - push it on the stack and mark it as visited
  - no available arrows out of X – pop it
  - no more available arrows from R – pop it
  - pick W as next city to visit (only choice left)
    - push it on the stack and mark it as visited
  - pick Y as next city to visit (random select)
    - this is the destination – all done

Recursion

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

When you turn this into a program, you end up with functions that call themselves (*recursive functions*)

```
int f(int x)
{
    int y;

    if(x==0)
        return 1;
    else
    {
        y = 2 * f(x-1);
        return y+1;
    }
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *factorial of n*

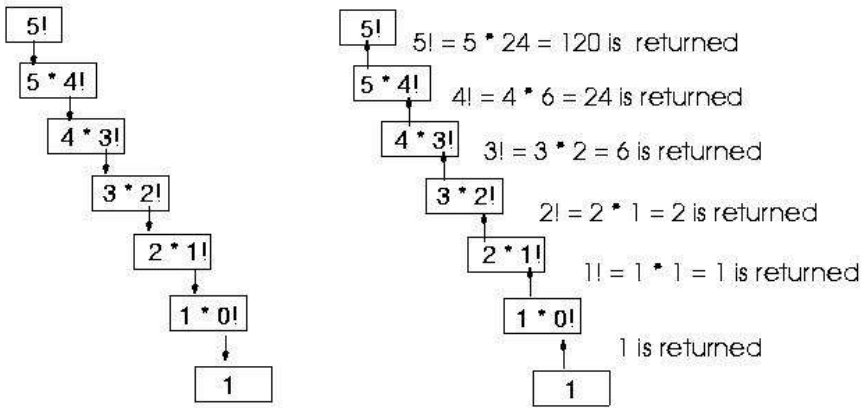
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n\text{-}1)!*n & \text{if } n > 0 \end{cases} \quad \text{(}\textit{recursive}\text{ solution)}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n\text{-}1)*n & \text{if } n > 0 \end{cases} \quad \text{(}\textit{closed form}\text{ solution)}$$

Coding the factorial function

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```



Coding the factorial function…

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Another example:
n choose *k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \qquad \textit{(recursive solution)}$$

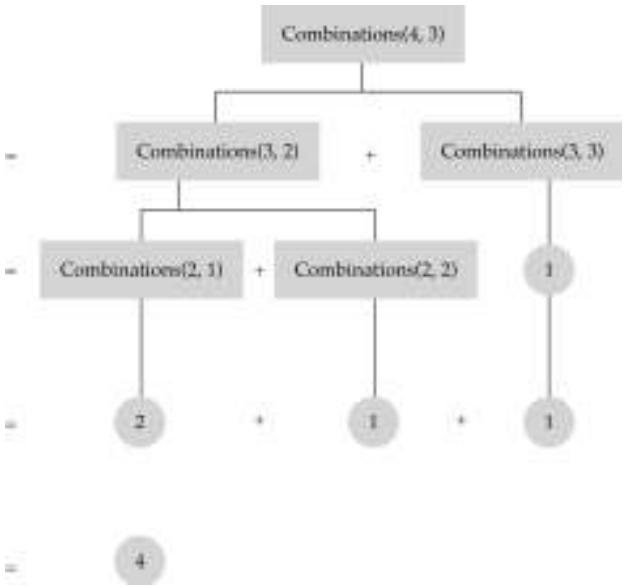
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \ (k = 1), \quad \binom{n}{n} = 1 \ (k = n)$$

n choose *k* (combinations)…

```
int Combinations(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```



Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How is recursion implemented?

- What happens when a function gets called?

```
int a(int w)
{
 return w+w;
}

int b(int x)
{
 int z,y;
 ..... // other statements
 z = a(x) + y;
 return z;
}
```

What happens when a function is called?...

- An **activation** record is stored into a stack **(run-time stack)**
  1) The computer has to stop executing function ***b*** and starts executing function **a**
  2) Since it needs to come back to function ***b*** later, it needs to store everything about function **b** that is going to need (**x, y, z**, and the place to start executing upon return)
  3) Then, **x** from **a** is bounded to **w** from **b**
  4) Control is transferred to function **a**

What happens when a function is called? …

- After function **a** is executed, the activation record is popped out of the run-time stack
- All the old values of the parameters and variables in function **b** are restored and the return value of function **a** replaces **a(x)** in the assignment statement

What happens when a recursive function is called?...

- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
 int y;
 if(x==0)
  return 1;
 else {
   y = 2 * f(x-1);
   return y+1;
}
}
```

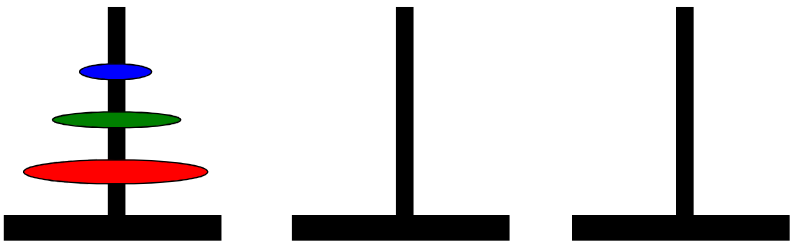
Tower of Hanoi

- There are three towers
- 64 gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

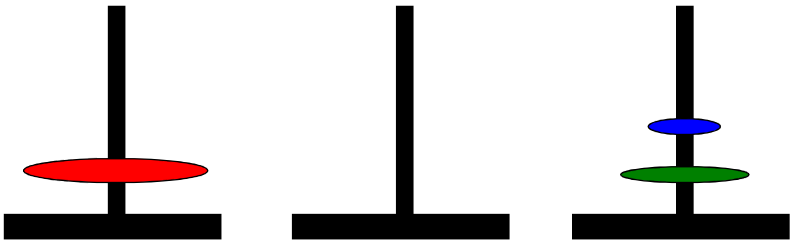
Tower of Hanoi…

- The disks must be moved within one week. Assume one disk can be moved in 1 second. Is this possible?
- To create an algorithm to solve this problem, it is convenient to generalize the problem to the "N-disk" problem, where in our case N = 64.

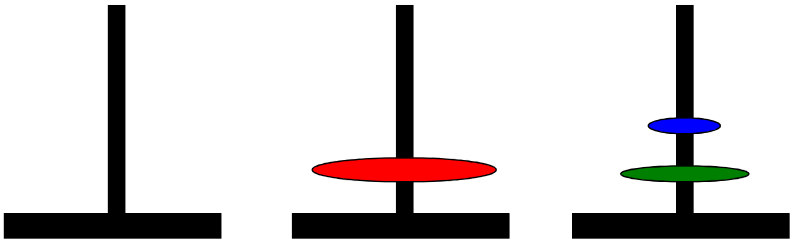
Recursive Solution



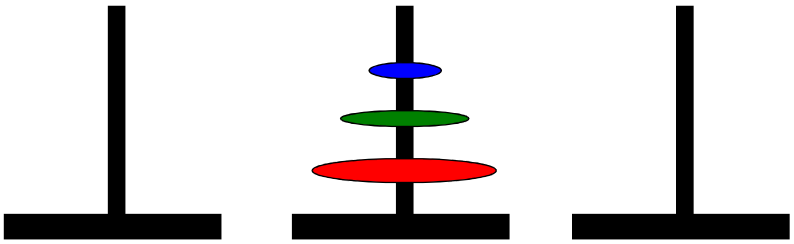
Recursive Solution



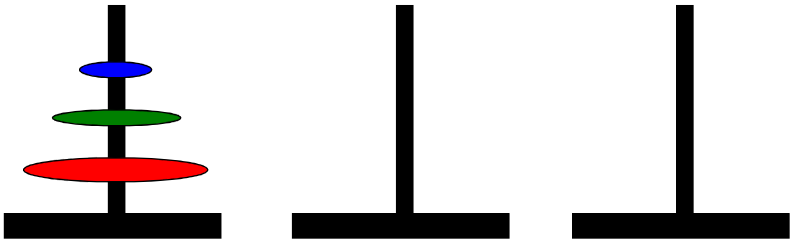
Recursive Solution



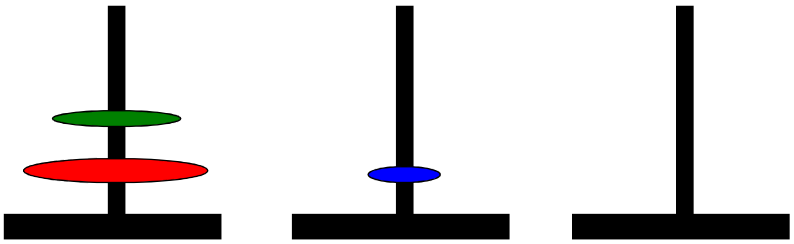
Recursive Solution



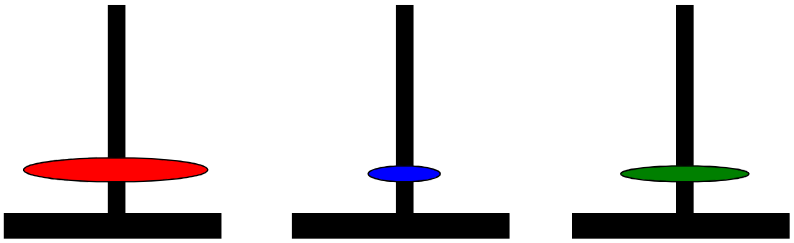
Tower of Hanoi



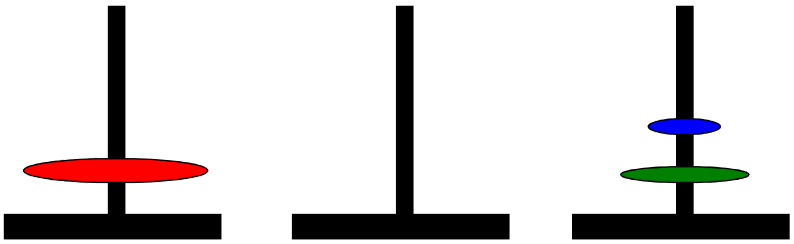
Tower of Hanoi



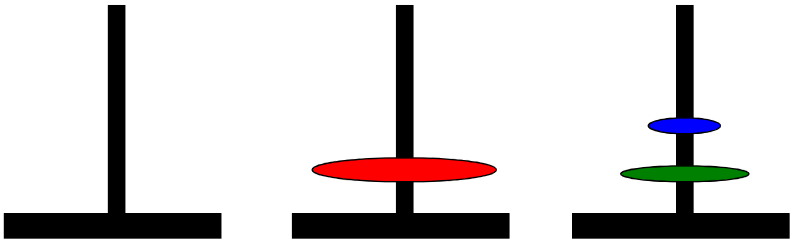
Tower of Hanoi



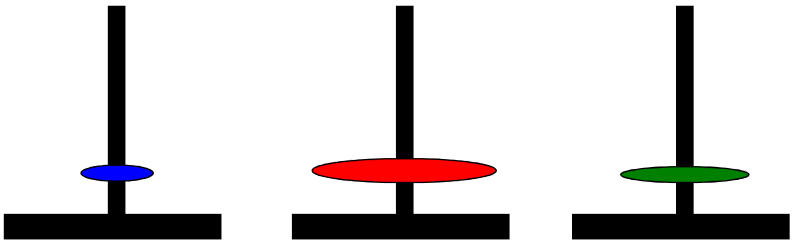
Tower of Hanoi



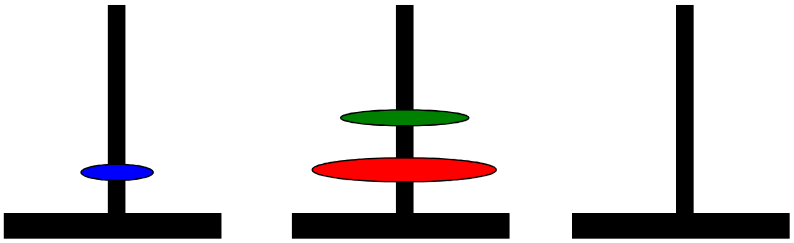
Tower of Hanoi



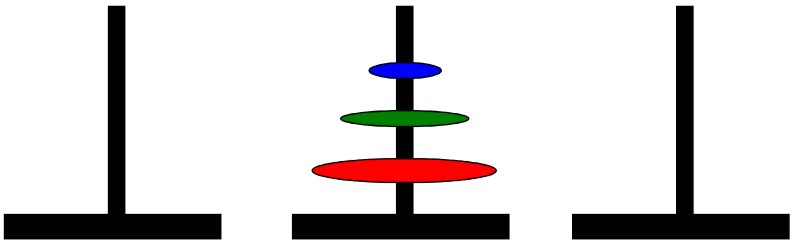
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Recursive function

```
void Hanoi(int n, string a, string b, string c)
  {
    if (n == 1)  /* base case */
      Move(a,b);
    else { /* recursion */
      Hanoi(n-1,a,c,b);
      Move(a,b);
      Hanoi(n-1,c,b,a);
    }
  }
```

Infix, Postfix and Prefix Expressions

Algebraic Expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands. Example of familiar operators include +,-,*, /, ^
- Considering these definitions of operands and operators now we can write an example of expression as x+y*z.

Infix, Postfix and Prefix Expressions

- **INFIX**
  - The expressions in which operands surround the operator, e.g. x+y, 6*3
  - This way of writing the Expressions is called infix notation.
- **POSTFIX**
  - Postfix notation are also Known as Reverse Polish Notation (RPN).
  - The expressions in which operator comes after the operands, e.g. xy+, xyz+* etc.
- **PREFIX**
  - Prefix notation are also Known as Polish notation.
  - The expression in which operator comes before the operands, e.g. +xy, *+xyz etc.

Operator Priorities

- How do you figure out the operands of an operator?
  - a + b * c
  - a * b + c / d
- This is done by assigning operator priorities.
  - priority(*) = priority(/) > priority(+) = priority(-)
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
  - a + b - c
  - a * b / c / d

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.

  (a + b) * (c – d) / (e – f)

WHY Prefix and Postfix?

- Why to use these weird looking PREFIX and POSTFIX notations when we have simple INFIX notation?
- To our surprise INFIX notations are not as simple as they seem specially while evaluating them
- To evaluate an infix expression we need to consider operators' Priority and Associative property
  - For example expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).
- To solve this problem Precedence or Priority of the operators were defined.
- Operator precedence governs evaluation order.
  - An operator with higher precedence is applied before an operator with lower precedence.

Infix Expression Is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and Prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier to evaluate expressions that are in these forms.

Examples of Infix to Prefix and Postfix

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

  infix: 2+5 *postfix*: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Suppose that we would like to rewrite A+B*C in postfix

- Applying the rules of precedence,we obtained

A+B*C

A+(B*C) Parentheses for emphasis

A+(BC*) Convert the multiplication,Let D=BC*

A+D Convert the addition

A(D)+

ABC*+ Postfix Form

Postfix Examples

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

When do we need to use them

- Expression is scanned from user in infix form;
- It is converted into prefix or postfix form and,
- Then evaluated without considering the parenthesis and priority of the operators.

Algorithm for Infix to Postfix

1) Examine the next element in the input.
2) If it is operand, output it.
3) If it is opening parenthesis, push it on stack.
4) If it is an operator, then
   i) If stack is empty, push operator on stack.
   ii) If the top of stack is opening parenthesis, push operator on stack
   iii) If it has higher priority than the top of stack, push operator on stack.
   iv) Else pop the operator from the stack and output it, repeat step 4
5) If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6) If there is more input go to step 1
7) If there is no more input, pop the remaining operators to output.

Algorithm for Infix to Postfix…

Algorithm INFIX_TO_POSTFIX(EXPR)
Input: An input string EXPR containing a NULL in its rightmost character position.
Output: Postfix equivalent POSTFIX of EXPR
[Initialize an empty Stack S]
1. TOP← -1
2. i← 0
[Examine the next element in the EXPR]
3. Repeat while EXPR[i] != NULL
4. **if** EXPR[i] is operand **then**
5. POSTFIX← POSTFIX O EXPR[i]
6. **if** EXPR[i] is opening parenthesis **then**
7. push(S, EXPR[i])
8. **endif**
9. **if** EXPR[i] is operator
10. **if** isEmpty(S) **then** push(S,EXPR[i]);
11. **else if** S[TOP] is opening parenthesis **then** push(S,EXPR[i])
12. **else if** EXPR[i] has higher priority **then** S[TOP] **then** push(S,EXPR[i])
13. **else**
14. **Repeat while** S(TOP) has higher priority then EXPR[i]
15. POSTFIX← POSTFIX O pop[S]
16. **Goto** step 10

Algorithm for Infix to Postfix…

Algorithm INFIX_TO_POSTFIX(EXPR) Cont…

17. **if** EXPR[i] is closing parenthesis **then**
18. **Repeat while** S[TOP] is not equal to opening parenthesis
19. POSTFIX← POSTFIX O pop[S]
20. *[Pop opening parenthesis and discard it]*
21. pop[S]
[Increment the loop counter and proceed with next iteration]
22. i← i+1
23. endwhile

[Pop remaining elements and output]
24. **Repeat while** isEmpty(S) is true
25. POSTFIX← POSTFIX O pop[S]
[Display and return POSTFIX]
26. Output POSTFIX
27. return POSTFIX

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Example

- (5 + 6) * 9 +10

will be

- 5 6 + 9 * 10 +

Evaluation a postfix expression

- Each operator in a postfix string refers to the previous two operands in the string.
- Suppose that each time we read an operand we **push** it into a stack. When we reach an operator, its operands will then be 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.

Evaluating a postfix expression

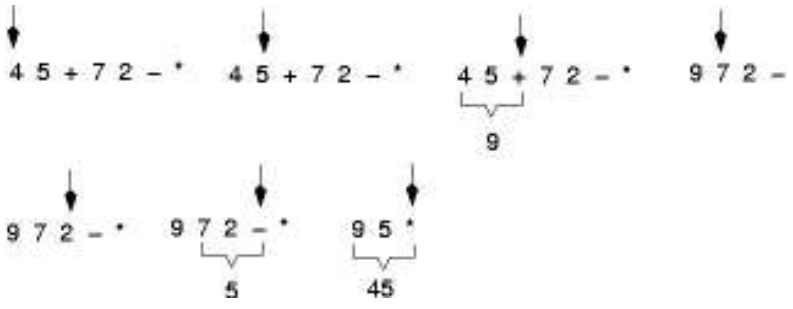
- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack,apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Evaluating a postfix expression: Algorithm

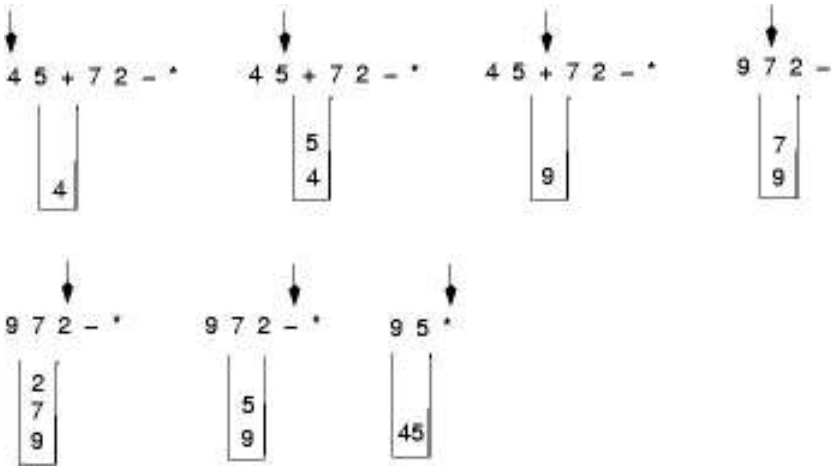
Algorithm POSTFIX_EVAL(EXPR)
Input: A postfix expression string EXPR containing a NULL in its rightmost character position, TEMP is a temporary variable
Output: Result of evaluation of EXPR
[Initialize an empty Stack S]

```
1.  TOP← -1
2.  i← 0
3.  [Examine the next element in the EXPR]
4.  Repeat while EXPR[i] != NULL
5.  [If it is number then push it on the stack]
6.      if EXPR[i] is number  then
7.          push(S, EXPR[i])
8.      endif
9.  [If it is operator then pop two elements from stack and apply EXPR[i] on them]
10.     if EXPR[i] is an operator then
11.         operand2=pop(S)
12.         operand1=pop(S)
13.         Apply EXPR[i] on operand1 and operand2 and store result in TEMP
14.         push(S, TEMP)
15.     endif
16.     [Pop off the result]
17.     return pop(S)
```

Example: Evaluation of postfix expressions



Postfix expressions: Algorithm using stacks



Question : Evaluate the following expression in postfix : 623+-382/+*2^3+

Final answer is

- 49
- 51
- 52
- 7
- None of these

Evaluate- 623+-382/+*2^3+

Symbol	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

Evaluate- 623+-382/+*2^3+

Symbol	**opnd1**	**opnd2**	**value**	**opndstk**
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**