Stack

Marwadi University

Department of CE/IT

Unit no 2
Linear Data Structures and their representation
Data Structure (01CE1301)

Stack

A ***stack*** is a linear data structure which can be implemented using an array or a linked list.

The elements in a stack are added and removed only from one end, which is called the ***top***.

Hence, stack is called a ***LIFO*** (Last In First Out)

Array representation of Stacks

Every ***stack*** has a variable **TOP** which is used to store the address of the topmost element of the stack.

Another variable **MAX** which is used to store the maximum number of elements that stack can hold.

If TOP = -1 then stack is Empty.

If TOP = MAX – 1 then stack is Full.

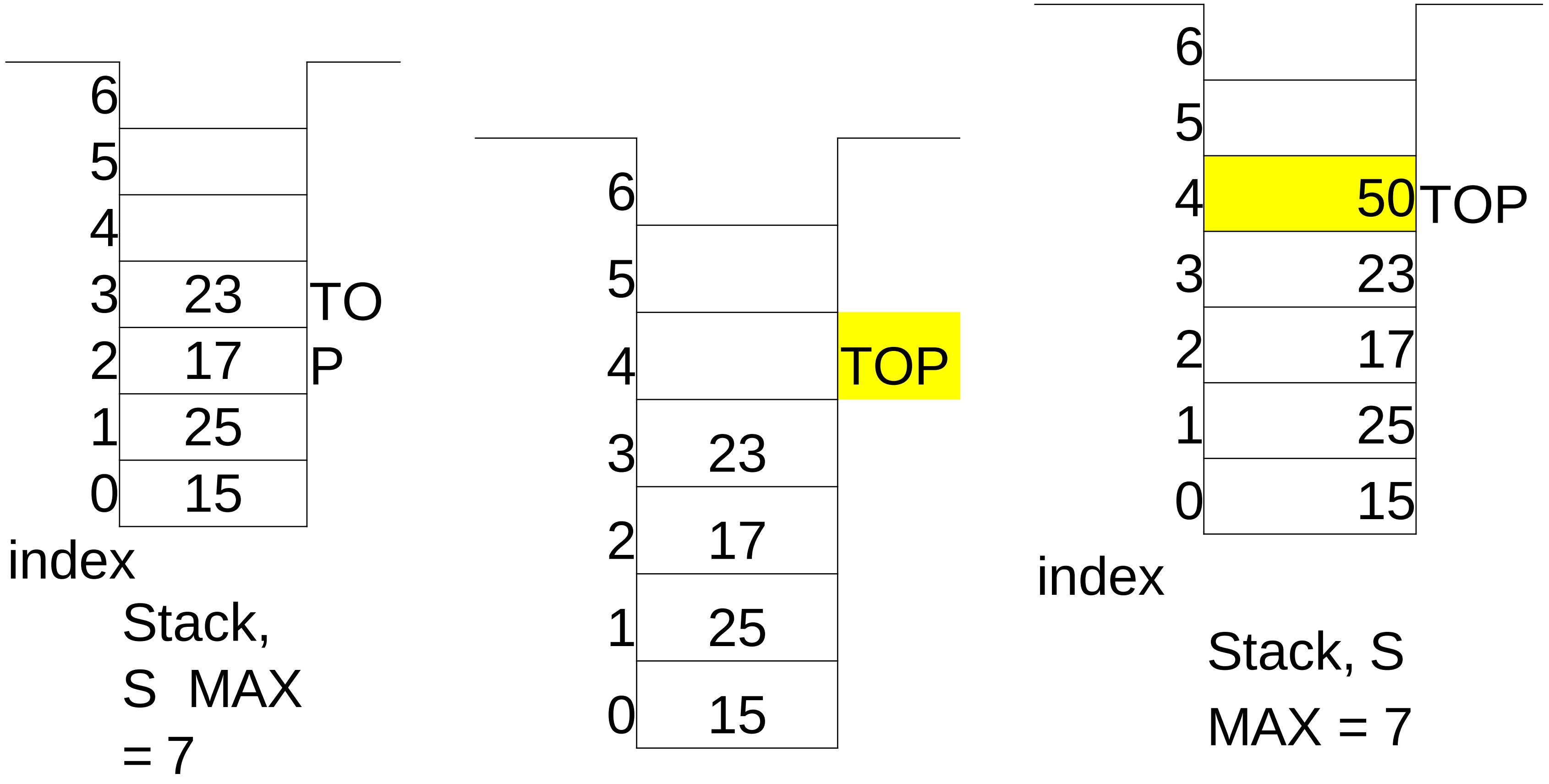
Operations on a Stack

PUSH Operation

S, Stack

MAX, maximum number of elements in a Stack

TOP, index of topmost element in a Stack



Operations on a Stack

PUSH Operation

S,	Stack
MAX,	maximum number of elements in a Stack
TOP,	index of topmost element in a Stack

```
Step 1: [check for stack overflow]
        if TOP = MAX – 1 then
            Write “Stack Overflow”
            return
      [End of If]
Step 2: [increment TOP]
        Set TOP = TOP + 1
Step 3: [insert element]
        Set S[TOP] = Value
Step 4: [finished] return
```

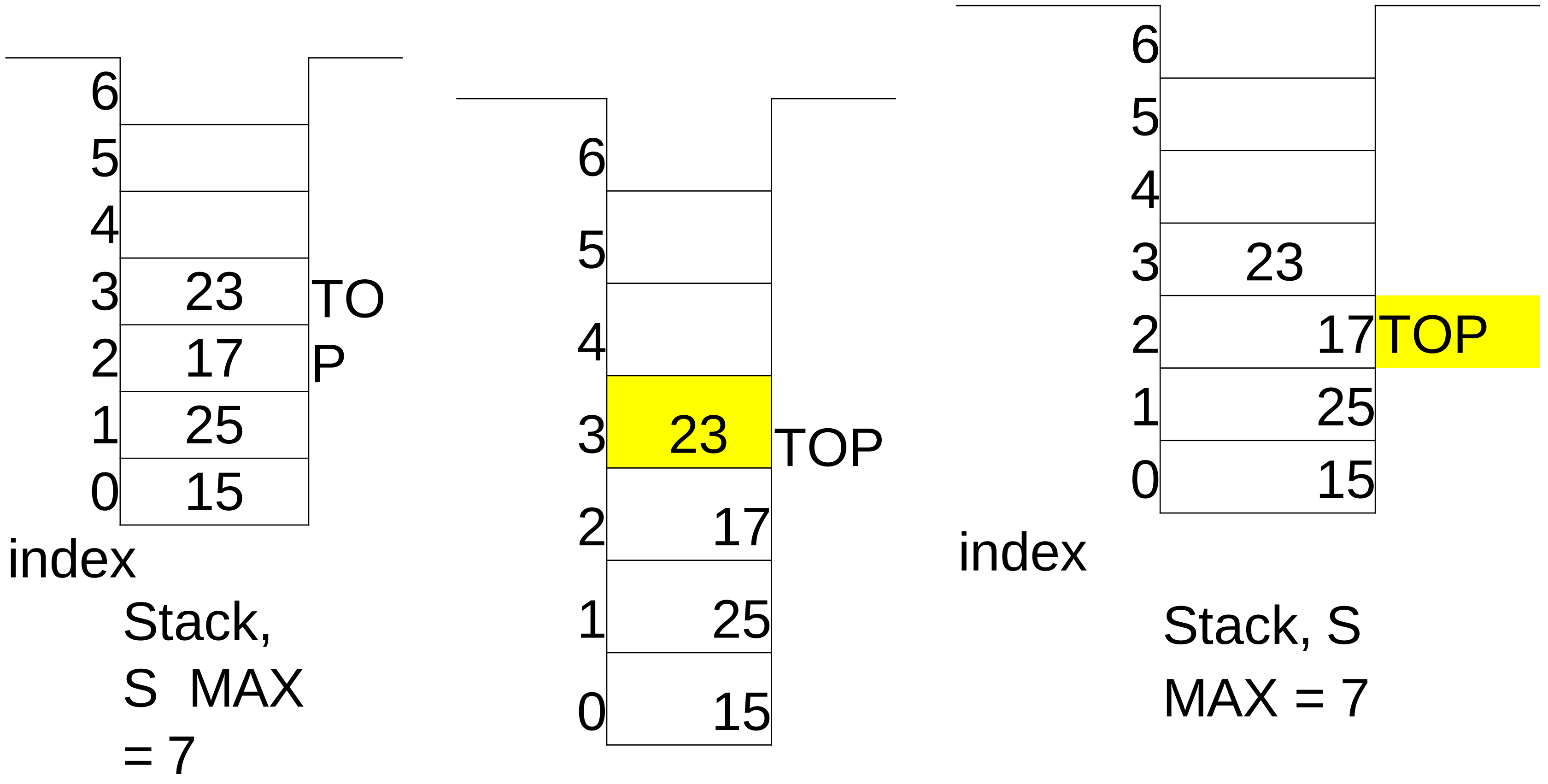
Operations on a Stack

- **POP Operation**

  S, Stack

  MAX, maximum number of elements in a Stack

  TOP, index of topmost element in a Stack



Operations on a Stack

POP Operation

```
        S,   Stack
      MAX,   maximum number of elements in a Stack
      TOP,   index of topmost element in a Stack
Step 1: [check for Underflow of the Stack]
          if TOP = – 1 then
              Write “Stack Underflow”
              return
        [End of If]
Step 2: [print/return topmost element]
          Set Value = S[TOP]
          Write Value
Step 3: [decrement TOP]
          Set TOP = TOP - 1
Step 4: [finished] return
```

Operations on a Stack

Peek / Peep Operation

```
        S,    Stack
      MAX,    maximum number of elements in a Stack
      TOP,    index of topmost element in a Stack
Step 1: [check for Underflow of the Stack]
        if TOP = – 1 then
            Write “Stack Underflow”
            return
      [End of If]
Step 2: [return top most element]
        return ( S[TOP] )
```

Operations on a Stack

Change Operation

```
 S,  Stack
MAX,  maximum number of elements in a Stack
TOP,  index of topmost element in a Stack
 I, index from the top of element which will change
Step 1: [check for stack underflow]
            If TOP – I + 1 < 0 then
                write “Stack underflow on Change”
                return
            [End of If]
Step 2: [Change Ith element from the top of stack]
            S[TOP-I+1] = New_Value
Step 3: [Finished]
        Return
```

Applications of Stack

1) Stacks are primarily used to convert an arithmetic expression from one form to another.

   Arithmetic Expression Forms: infix, prefix and postfix

2) Used in compiler for parsing the syntax of expressions, program, etc.

3) Certain programming languages are stack-oriented.

   i.e. Adding two numbers is done by Stack.

4) Stacks are used in function calls.

5) Stacks are implicitly used by recursive functions.

6) Some programming languages like c allows to store the local data on a stack.

Infix, Polish and Reverse Polish Notations

Notation	Example
Infix	A + B
Prefix / Polish Notation	+ A B
Postfix / Reverse Polish Notation	A B +

The order of evaluation of a postfix and prefix expression is always from left to right.

Convert infix to postfix expression

Direct Method

(1) (A–B) * (C+D)

Postfix Exp: A B – CD + *

(2) A + B * C

Postfix Exp: A B C * +

(3) (A + B) * C

Postfix Exp: A B + C *

(4) (A + B) / (C + D) – (D * E)

Postfix Exp: A B + CD + / D E * -

(5) A – (B / C + (D % E * F) / G) * H

Postfix Exp: A B C / D E % F * G / + H * -

Operator precedence

operator	precedence	evaluation
^ (Power), $, ↑	Higher	Right to left
* /	.	Left to right
+ -	Lower	Left to right

Convert infix to postfix expression

Stack Method usingAlgorithm:

(1) **A + B * C**

(A + B * C)

Infix Character Scanned	Stack	Postfix Expression
(	(	
A	(	A
+	(+	A
B	(+	A B
*	(+ *	A B
C	(+ *	A B C
)		**A B C* +**

Postfix Exp: A B C* +

Convert infix to postfix expression

Algorithm INFIX to POSTFIX Notation:

Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix expression is scanned
    (i) If operand, add it to the postfix expression
    (ii) If "(" , push it on the stack
    (iii) If ")" then:
        (a) Repeatedly pop from stack and add each operator to postfix expression until "(" is encountered.
        (b) Discard the "(". Remove "(" from stack and don't add it to the postfix expression.
    (iv) If an operator 'O' is encountered.then:
        (a) If precedence of operator at top of stack is higher or same as compare to 'O' then repeatedly it is popped and add to postfix expression.
        (b) Push the operator O to the stack.
Step 4: Exit.

Convert infix to postfix expression

Algorithm INFIX to POSTFIX Notation:

Note:

(1) If any operator has Right to Left associativity And if the precedence of operator at top of stack is SAME as compare to coming infix operator then push it on the Stack, don’t pop it.

(2) Exponentiation (power fun) is the highest level of precedence and Right to Left associativity.
Symbol: ^ $ ↑
(Different different books use different symbol for the exponentiation.)

Convert infix to postfix expression

Examples:

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

(2) **((A – (B + C)) * D) / (E + F)**

(3) **A / B $ C + D * E / F – G + H**

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

(5) **(A + B ^ D) / (E – F) + G**

(6) **A + B ^ C ^ D – E * F / G**

(7) **A + B – C * D * E $ F $ G**

(8) **A $ B – C * D + E $ F / G**

Infix to Postfix table

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

POSTFIX: AB+CD*E-*F*G/

Sr No	Expression	Stack	Postfix
0	(	(	
1	A	(	A
2	+	(+	A
3	B	(+	AB
4	)		AB+
5	*	*	AB+
6	(	*(	AB+
7	C	*(	AB+C
8	*	*(*	AB+C
9	D	*(*	AB+CD
10	-	*(-	AB+CD*
11	E	*(-	AB+CD*E
12	)	*	AB+CD*E-
13	*	*	AB+CD*E-*
14	F	*	AB+CD*E-*F
15	/	/	AB+CD*E-*F*
16	G	/	AB+CD*E-*F*G
17			AB+CD*E-*F*G/

2. ((A – (B + C)) * D) / (E + F)

POSTFIX: A –B + C * D / E + F

Sr No	Expression	Stack	Postfix
0	(	(	
1		(	
2	(	((	
3		((	
4	A	((	A
5		((	A
6	–	((–	A
7		((	A –
8	(	(((	A –
9		(((	A –
10	B	(((	A –B
11		(((	A –B
12	+	(((+	A –B
13		(((	A –B +
14	C	(((	A –B + C
15		(((	A –B + C
16	)	((	A –B + C
17		((	A –B + C
18	)	(	A –B + C
19		(	A –B + C
20	*	(*	A –B + C
21		(	A –B + C *
22	D	(	A –B + C * D
23		(	A –B + C * D
24	)		A –B + C * D
25			A –B + C * D
26	/	/	A –B + C * D
27			A –B + C * D /
28	(	(	A –B + C * D /
29		(	A –B + C * D /
30	E	(	A –B + C * D / E
31		(	A –B + C * D / E
32	+	(+	A –B + C * D / E
33		(	A –B + C * D / E +
34	F	(	A –B + C * D / E + F
35		(	A –B + C * D / E + F
36	)		A –B + C * D / E + F
37			A –B + C * D / E + F

Computation / Evaluation of Postfix (Reverse Polish) Expression

Algorithm Evaluation of Postfix Expression:

```
Step 1: Add ")" at the end of postfix expression
Step 2: Scan every character of postfix expression and repeat
        step 3 and step 4 until ")" is encountered.
Step 3: If an operand is encountered, push it on the stack.
            If an operator 'O' is encountered, then
                (a) pop two elements
                        $A \leftarrow$ Top element
                        $B \leftarrow$ Next to Top element
                (b) Evaluate B 'O' A
            (c)     Push the result of evaluation on the stack
            [End of If]
Step 4: Set Result = topmost element of the stack
Step 5: Exit.
```

Computation / Evaluation of Postfix (Reverse Polish) Expression

Example: 9 – ((3 * 4) + 8) / 4

Solution:

First Convert it into Postfix expression

Postfix expression is: 9 3 4 * 8 + 4 / -

Character Scanned	Stack	Operation
9	9	
3	9, 3	
4	9, 3, 4	
*	9, 12	3*4 = 12
8	9, 12, 8	
+	9, 20	12+8 = 20
4	9, 20, 4	
/	9, 5	20/4 = 5
-	**4**	9-5 = 4

Result = 4.

Computation / Evaluation of Postfix (Reverse Polish) Expression

Examples:

(1) **14 / 7 * 3 – 4 + 9 / 3**

(2) **10 + ((7 – 5) + 10) / 2**

(3) **5, 4, 6, +, *, 4, 9, 3, /, + , ***

(4) **A B + C D / * G H * +**

where A = 2, B = 4, C = 6, D = 3, G = 8, H = 7.

(5) **2 $ 3 + 5 * 2 $ 2 – 6 / 6**

Convert infix to prefix (Polish) expression

Direct Method

(1) (A–B) * (C+D)

Prefix Exp: * – A B + C D

(2) A + B * C

Prefix Exp: + A * B C

(3) (A + B) * C

Prefix Exp: * + A B C

(4) (A + B) / (C + D) – (D * E)

Prefix Exp: - / +A B +C D *D E

(5) A – (B / C + (D % E * F) / G) * H

Prefix Exp: -A*+/BC/*%DEFGH

Convert Infix Expression to Prefix (Polish) Expression

Algorithm Infix to Prefix Expression:

Step 1: Reverse the infix string (Interchange parenthesis)

Step 2: Convert infix exp. to postfix exp.

Step 3: Reverse the postfix exp.

Step 4: Exit

Note:

When Pushing operator, follow the rules of Right to Left associativity because we started from Right side (reverse the string)

Convert Infix Expression to Prefix (Polish) Expression

Example 1: A / B ^ C – D

Reverse the string: D – C ^ B / A

D – C ^ B / A)

Character Scanned	Stack	Postfix
	(	
D	(	D
-	(-	D
C	(-	D C
^	(- ^	D C
B	(- ^	D CB
/	(- /	D CB^
A	(- /	D CB^A
)		DCB^A/ -

Reverse Postfix Exp.And

Ans: Prefix Expression is: - / A ^ B C D

Convert Infix Expression to Prefix (Polish) Expression

Examples:

1) **2 * 3 / (2 – 1) + 5 * (4 – 1)**

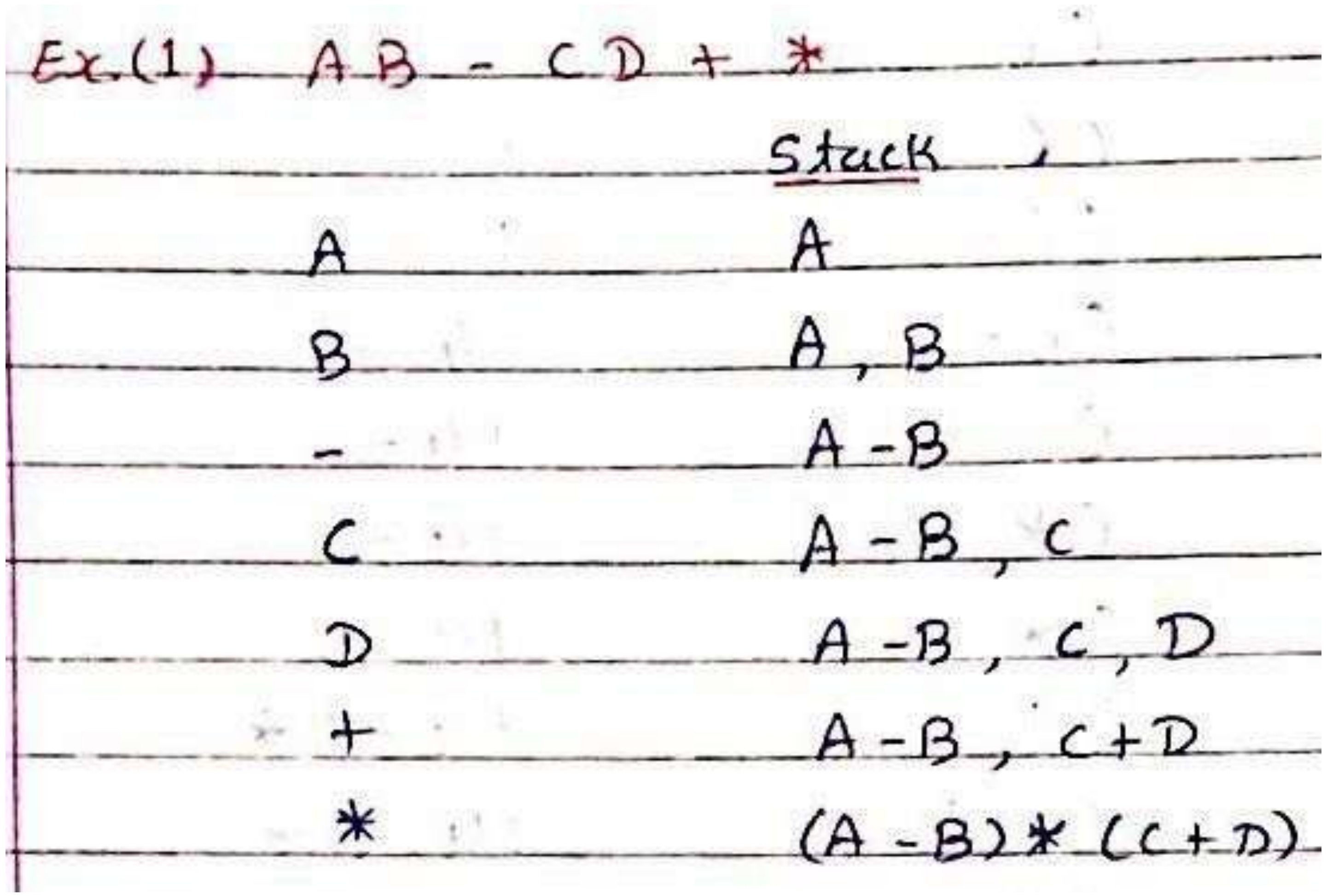
   Prefix: + / * 2 3 – 2 1 * 5 - 4 1

2) **A ^ B * C * D / E ^ F**

   Prefix: / * * ^ A B C D ^ E F

Convert postfix Expression to Infix Expression

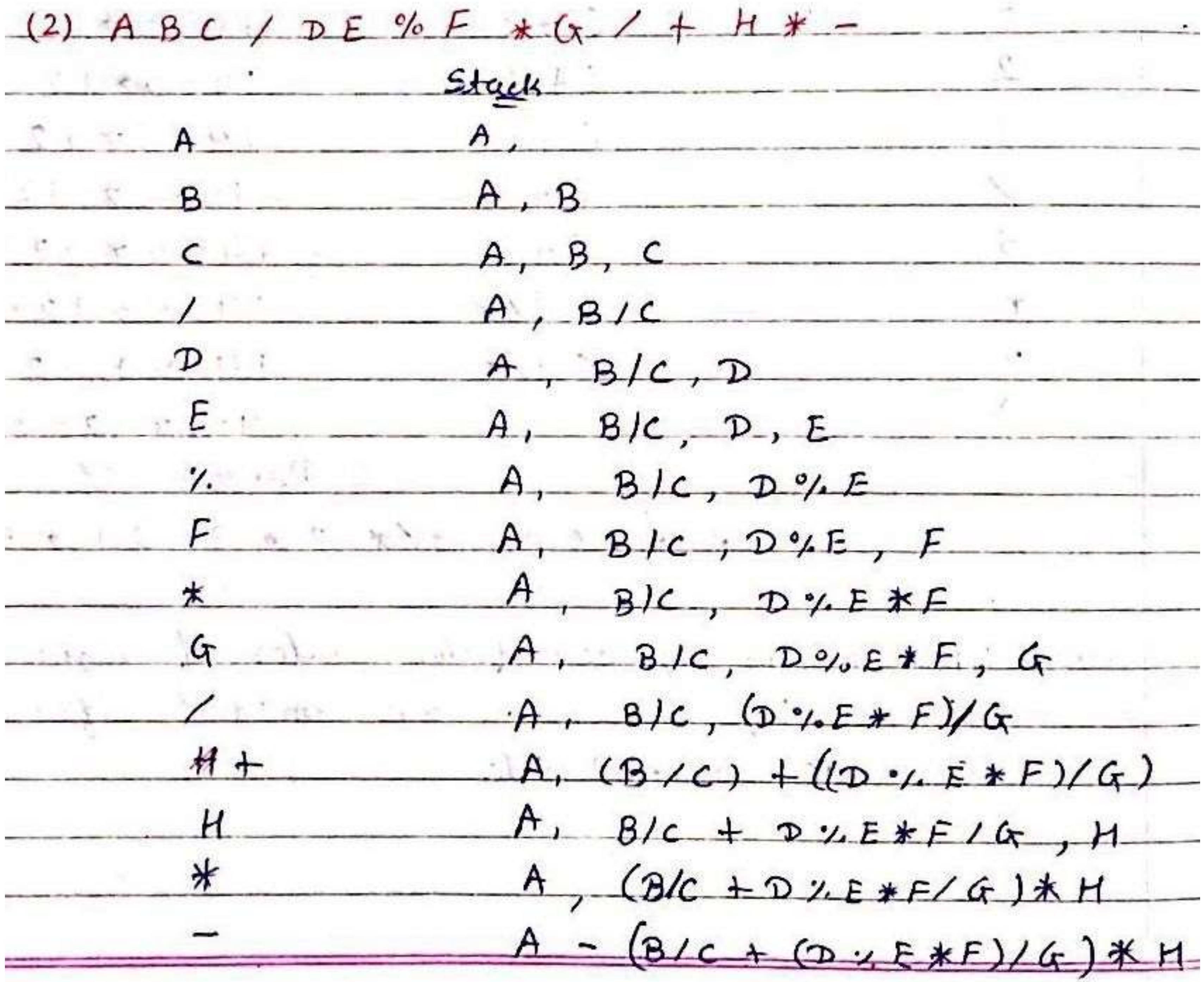
Same Rules or Algorithm as Computation / Evaluation of Postfix Expression



Ex.(1) AB - CD + *

	Stack
A	A
B	A, B
-	A - B
C	A - B, C
D	A - B, C, D
+	A - B, C + D
*	(A - B) * (C + D)

Convert Postfix to Infix



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

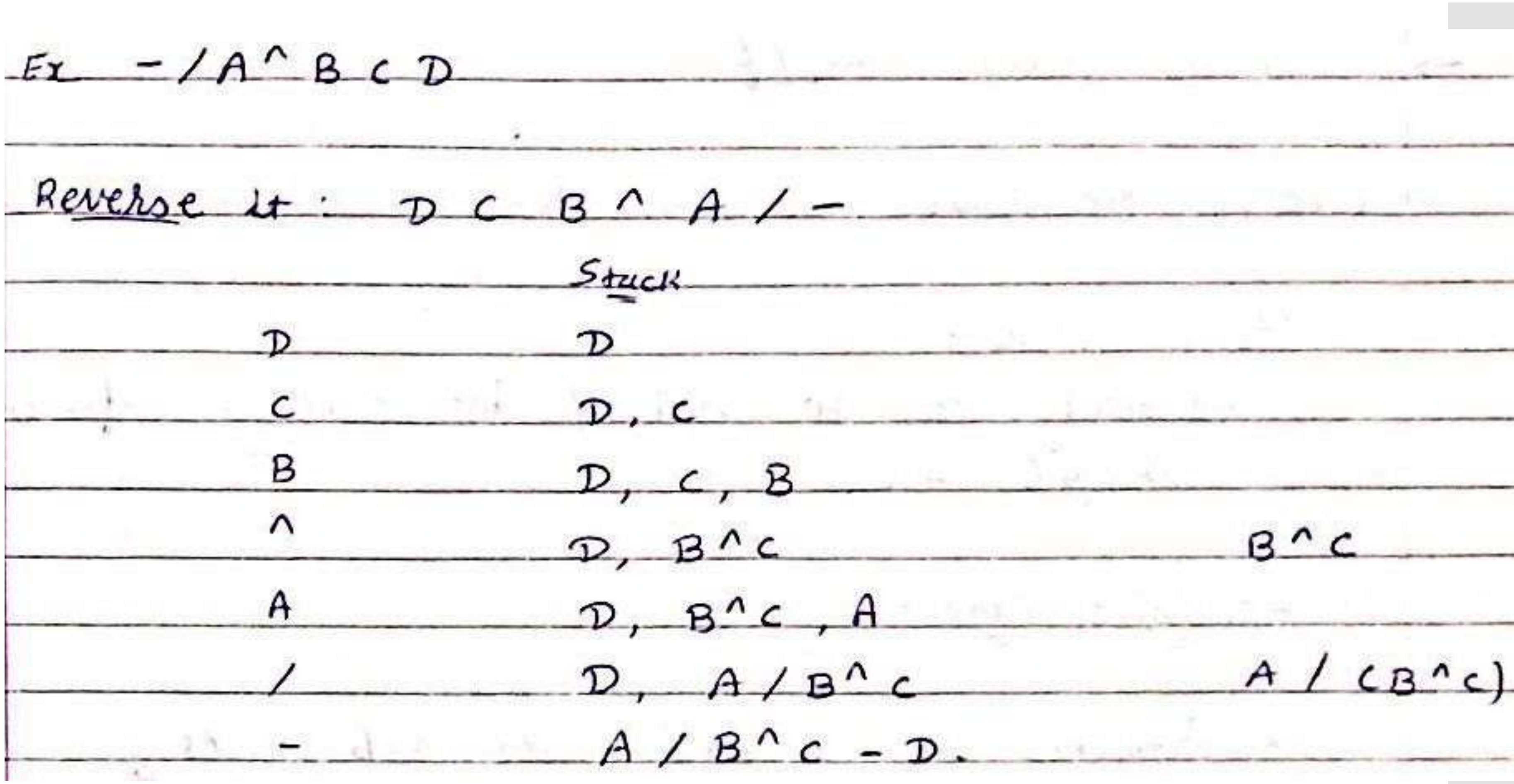
	Stack
A	A,
B	A, B
C	A, B, C
/	A, B/C
D	A, B/C, D
E	A, B/C, D, E
%	A, B/C, D%E
F	A, B/C, D%E, F
*	A, B/C, D%E*F
G	A, B/C, D%E*F, G
/	A, B/C, (D%E*F)/G
~~H~~ +	A, (B/C)+((D%E*F)/G)
H	A, B/C + D%E*F/G, H
*	A, (B/C + D%E*F/G)*H
-	A - (B/C + (D%E*F)/G)*H

Conversion Prefix to Infix Expression

Prefix to Infix

- Step 1: Reverse the string
- Step 2: Apply Algorithm for Evaluation of Postfix String
- Step 3: Again Reverse the string

Conversion Prefix to Infix Expression



Ex − / A ^ B C D

Reverse it : D C B ^ A / −

	Stack	
D	D	
C	D, C	
B	D, C, B	
^	D, B ^ C	B ^ C
A	D, B ^ C , A	
/	D, A / B ^ C	A / (B ^ C)
−	A / B ^ C − D.	

Stack Application in Recursion

```
int factorial(int n)
{
   if (n==1)                                             //termination
   condition
          return 1;
   else
          return (n*factorial(n-1));
          //Recursive call
}
void main()
{  int num,result;
   clrscr();
   cout<<"\nEnter number:";
    cin>>num;
   result = factorial(num);
   cout<<"\nFactorial of a number "<<result;
   getch();
}
```

Recursion

```
factorial(1) = 1      //termination condition

factorial(n) = n * factorial(n-1)

factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 *  (3 * factorial(2)))
= 5 * (4 *  (3 * (2 * factorial(1))))
= 5 * (4 *  (3 * (2 * 1)))
= 5 * (4 *  (3 * 2))
= 5 * (4 *  6)
= 5 * 24
= 120
```

```
int factorial(5)
{

5*factorial (4)

return address to 5

}
```

```
int factorial(4)
{

4*factorial (3)

return address to 4

}
```

```
int factorial(3)
{

3*factorial (2)

return address to 3

}
```

```
int factorial(2)
{

2*factorial (1)

return address to 2

}
```

```
int factorial(1)
{

return 1

}
```

1
return address to 2
2
return address to 3
3
return address to 4
4
return address to 5
5
return address to main

1*1=1

2*1=2

3*2=6

4*6=24

5*24=120

Recursion

In recursive function,

1. Save parameters, local variables, and return address on stack.
2. If the base criteria(termination condition) has been reached, perform computation and go to step 3. Otherwise perform partial computation and go to step 1.
3. Restore most recently saved parameters, local variables, and return address from stack. Go to this return address.

Recursive Functions

Basic 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 the current value is matches with value of base case. If yes, process and return the value.

Step 3: Divide the problem into a smaller sub-problems

Step 4: Call the function on the sub-problems

Step 5: Combine the results of sub-problems

Step 6: Return the result of entire problem.

Recursive Functions

Recursion Vs Iteration

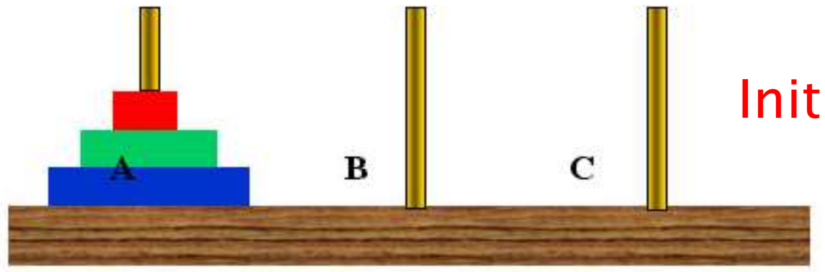
Advantages of Recursive Functions

- Recursive solution is tend to be shorter
- Divide-n-Conquer formula
- Useful for complex problems

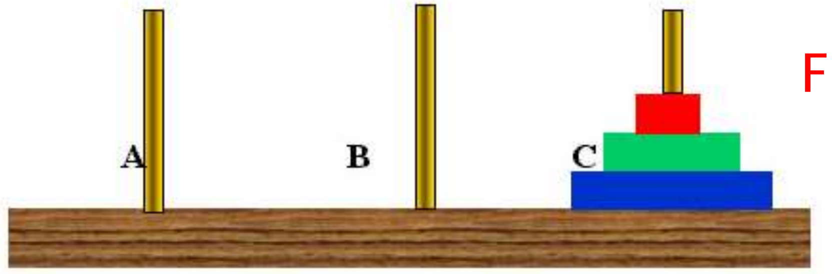
Disadvantages of Recursive Functions

- Recursion is difficult concept
- Recursive function takes more memory and time to execute, with compare to non-recursive (Iterative)
- Difficult to find bugs when using global variables.

Tower of Hanoi



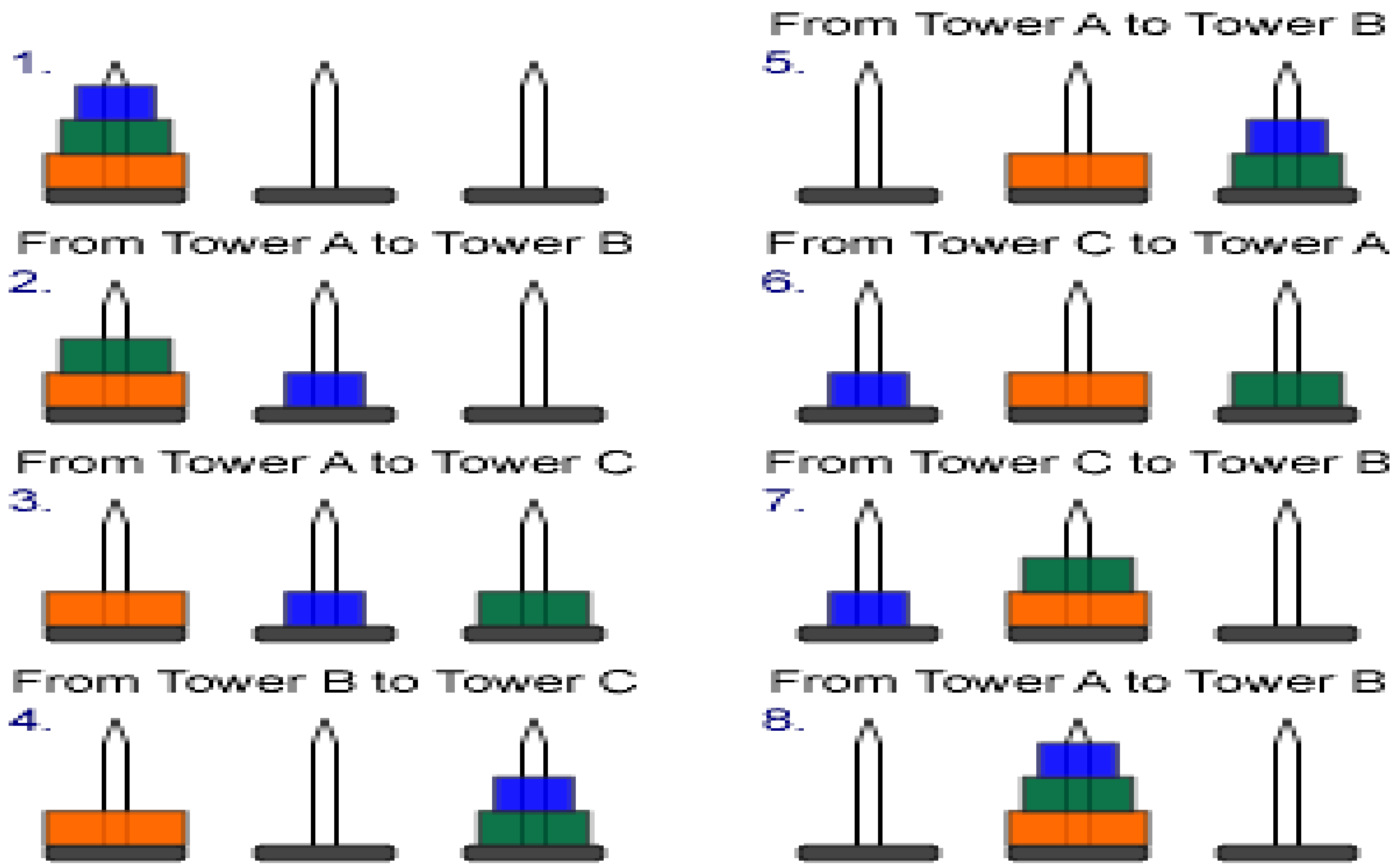
Initial State



Final State

Only one disk can be moved at a time, from one peg to another, such that it is never placed on a disk of smaller diameter.

Solution for 3 disks

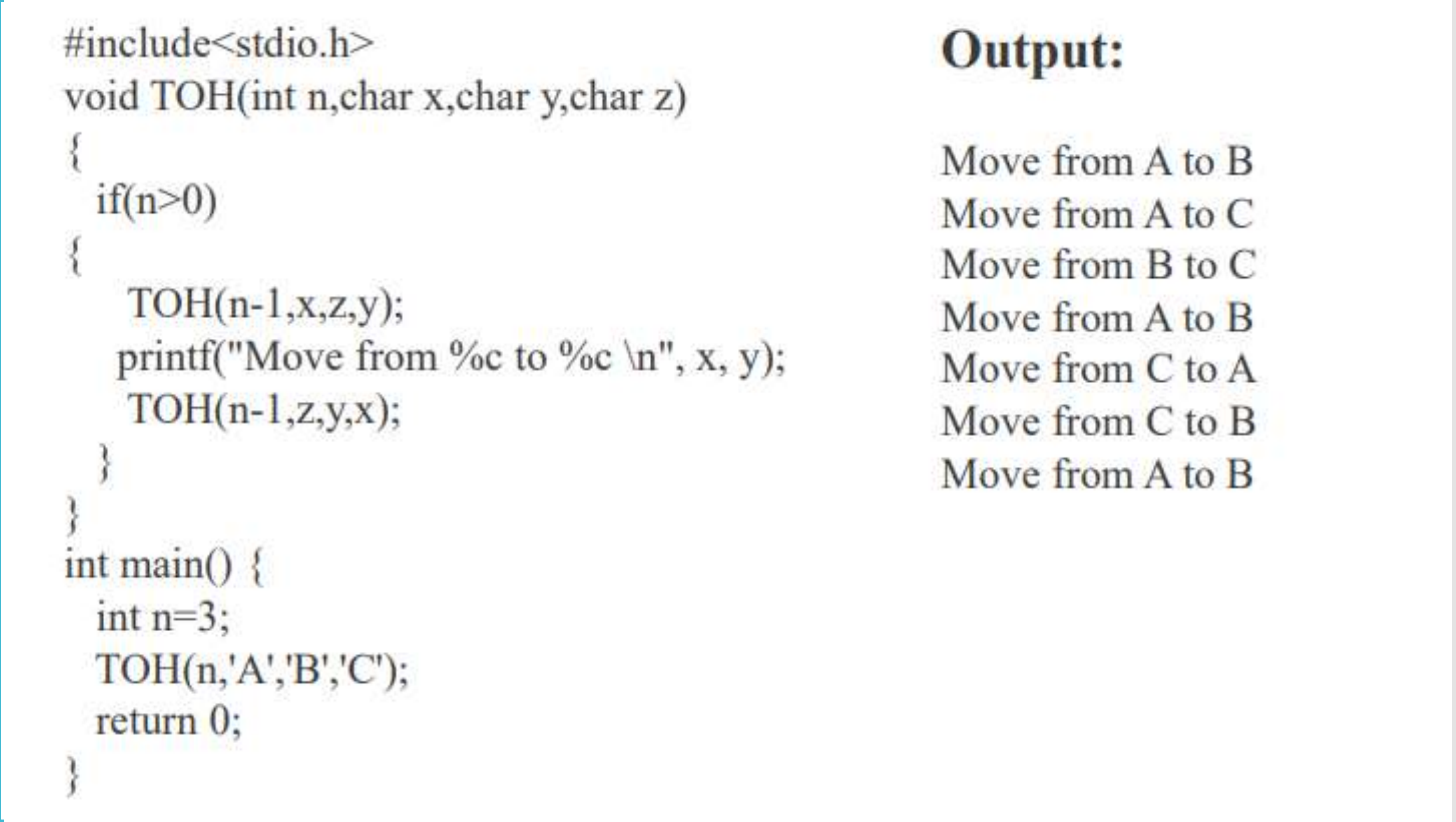


Algorithm

- Move the top N-1 disks from Src to Aux (using Dst as an intermediary peg)
- Move the bottom disk from Src to Dst
- Move N-1 disks from Aux to Dst (using Src as an intermediary peg)

```
1. if (n=1)
    1. writeln('Move the plate from ', source, ' to ', dest)
2. else
    1. Hanoi(n-1, source, aux, dest)
    2. Hanoi(1, source, dest, aux)
    3. Hanoi(n-1, aux, dest, source)
```

Implementation of Tower of Hanoi



```c
#include<stdio.h>
void TOH(int n,char x,char y,char z)
{
  if(n>0)
{
    TOH(n-1,x,z,y);
   printf("Move from %c to %c \n", x, y);
    TOH(n-1,z,y,x);
  }
}
int main() {
  int n=3;
  TOH(n,'A','B','C');
  return 0;
}
```

Output:

```
Move from A to B
Move from A to C
Move from B to C
Move from A to B
Move from C to A
Move from C to B
Move from A to B
```

THANK YOU