Data Structure

- a data structure is a particular way of storing and organizing data so that it can be used efficiently.
- Different kinds of data structures are suited to different kinds of applications

For example:

- An array data structure stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed-length or expandable.

- Record: Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.

Data Structure

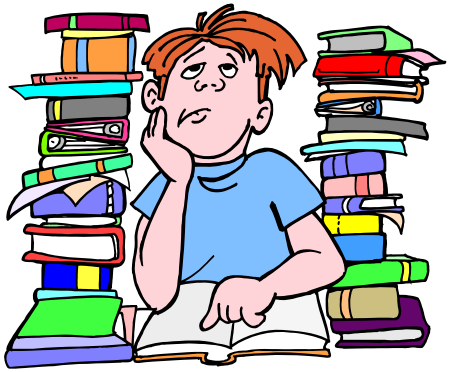
Data Structure describes,

- how to store a collection of objects in memory,
- what operations we can perform on that data,
- the algorithms for those operations, and
- how time and space efficient those algorithms are.

- A logical relationship among data elements that support specific data manipulation functions.

- Some data structures are linear in fashion. some are non-linear.
- In linear data structure processing of data items is done in linear fashion. Linear adjacency is exist among the data.

What is Stack



- It is an ordered group(List) of homogeneous items of elements.

Characteristics

- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).

- It means: the last element inserted is the first one to be removed

- The last element to be added is the first to be removed called LIFO: (Last In, First Out) data structure.

What is Stack

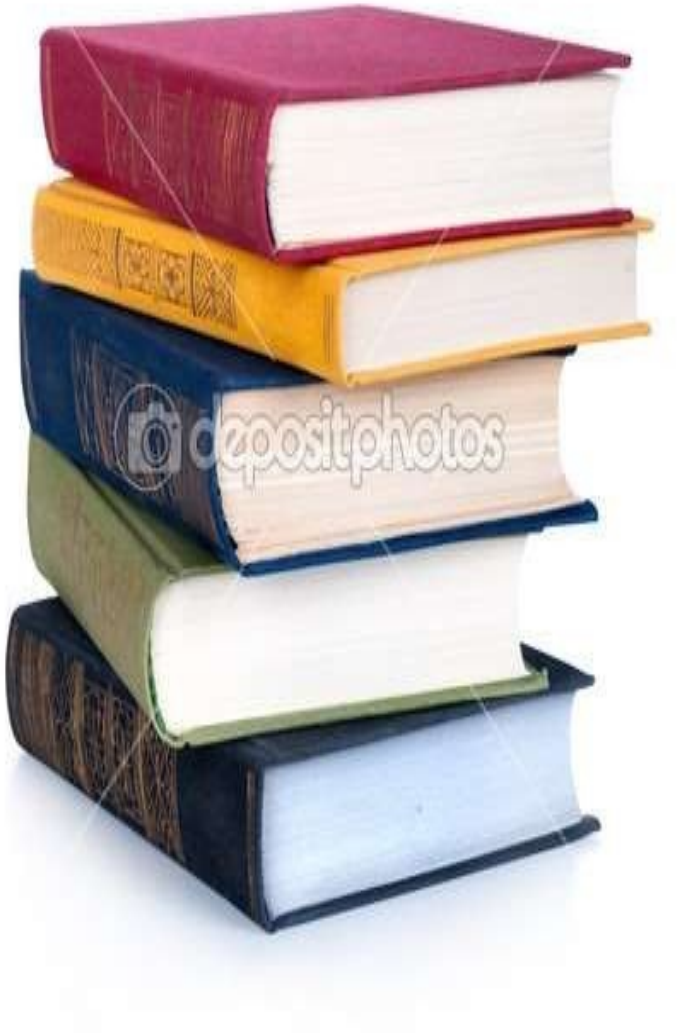
So,

- New item is added on top of the stack
- Item which is at the top of the stack will be removed from the stack.

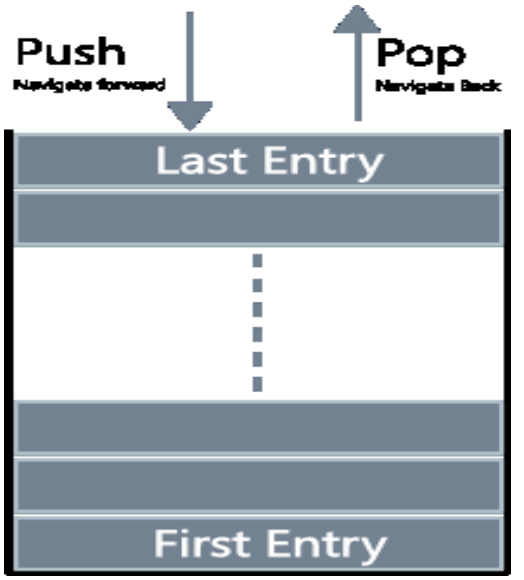
- So stack is either proceed upward or downward .
- Upward and downward depends on which side is designed as a top of the stack.

- Example: Making tower of books

EXAMPLES OF STACK:



What is Stack



- Two Main operation on stack:

Push Operation and Pop Operation

- Push operation insert an element to the Stack.
- Pop operation remove (delete) an element from the stack.

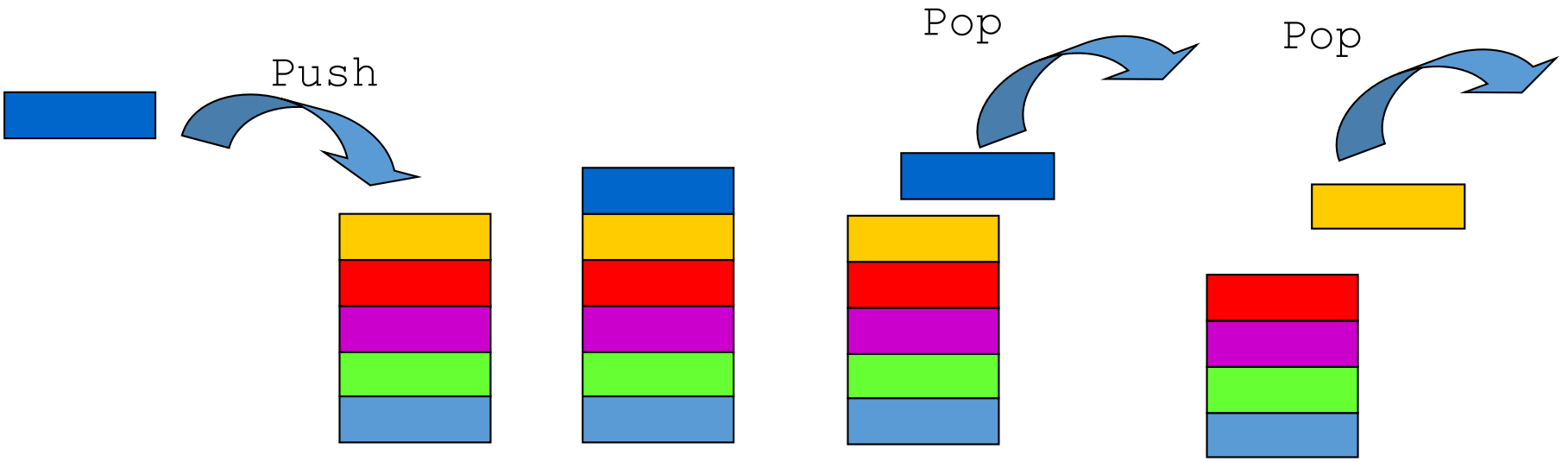
How do we keep track which element to be removed and where the new element to be added to stack???????

Since insertion and deletion is performed at only one end so one end is designed as a top of stack.

Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

- LIFO – Last in, First out



Basic Stack Operations

- Creating and Initializing the Stack.
- Inserting an element into Stack.
- Deleting an element from the Stack.
- Retrieving and changing specific element from the Stack.
- Is the Stack empty?
- Is the Stack full?
- Clear the Stack
- Determine Stack Size

Creating Stack

- There are Two way to implement the Stack Data structure

- Static Implementation of stack
- Dynamic Creation of Stack.

- The first approach is implementing Stack using ARRAY.

- The Second approach is implementing Stack using Linked List(pointer).

Array Implementation of Stack

Array implementation of stack require followings:

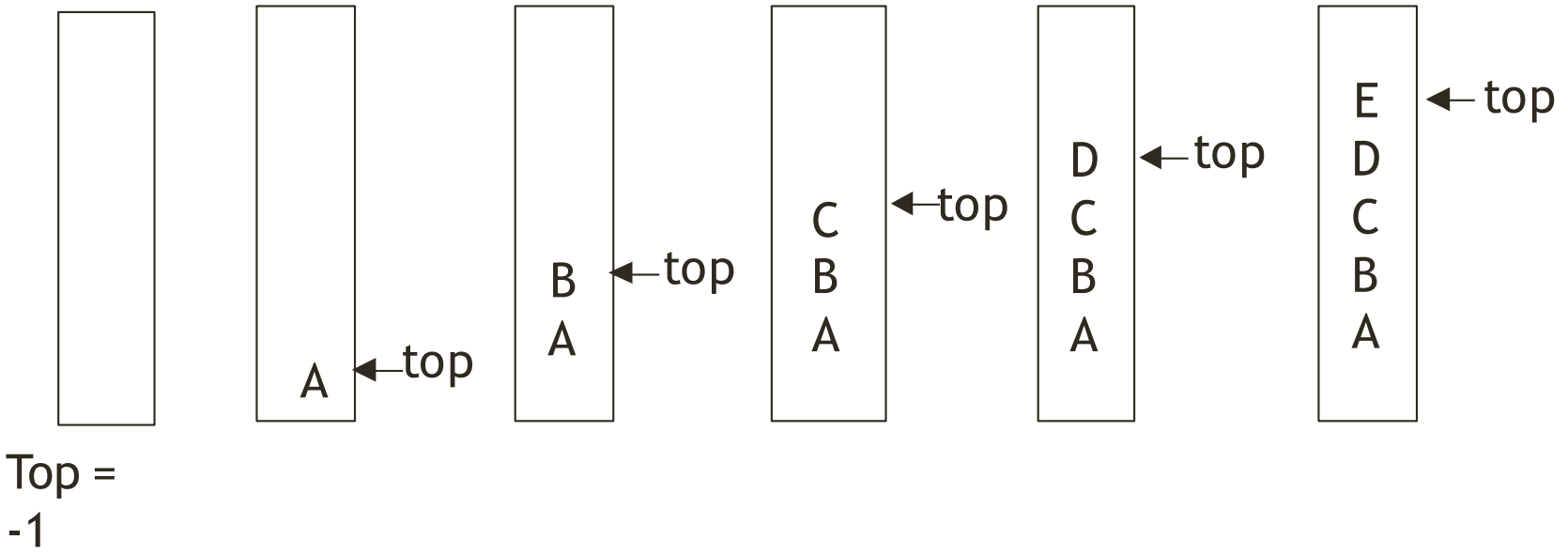
- Declare an array of appropriate size.
- Create one index variable which will work as a top of the stack.

- During stack operations stack may grow or shrink within the space reserved for it.

- When stack is empty ,index variable has value -1 . when stack has one element ,index variable has value 0,when stack has 2 elements then index variable has value 1 ,so on called stack is growing.

Array Implementation of Stack

http://www.csanimated.com/animation.php?t=Stack



Array Implementation of Stack

- When stack has Size-1 elements then we can say stack is full. Next push operation put the stack in overflow condition.
- In short, every push operation increment the value of index variable by 1.
- Similarly every pop operation remove top most element from the stack and decrement the index variable by 1.

Exercise: Stacks

- Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)

Stack example

- https://yongdanielliang.github.io/animation/web/Stack.html

Push operation

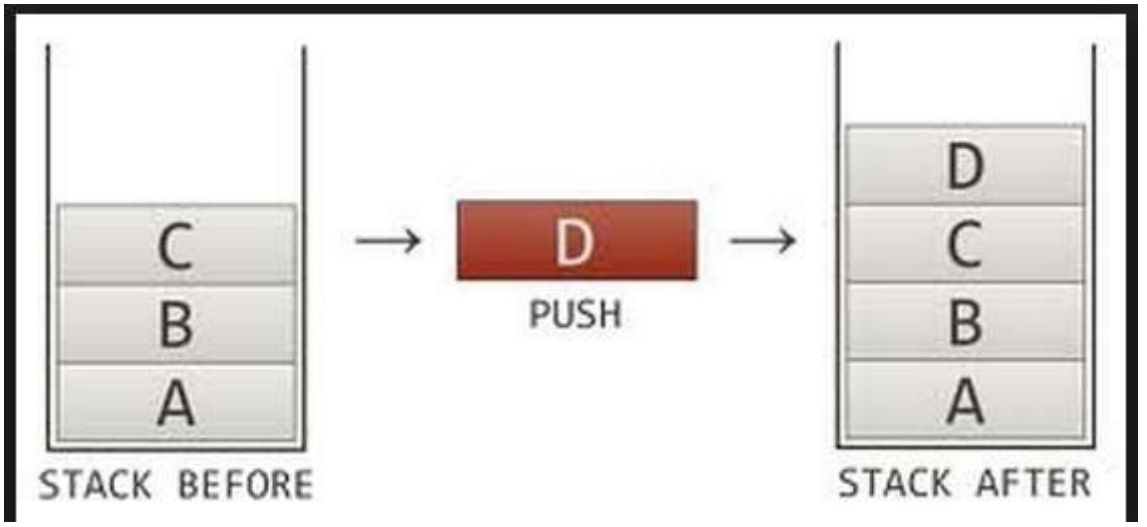
- PUSH(S,TOP,X):

1. [check Stack is Overflow or not] if Top>= Size-1 then
   write(" Stack Overflow");
   return
2. [increment Top by 1]

   top <- top +1
3. [insert element x]

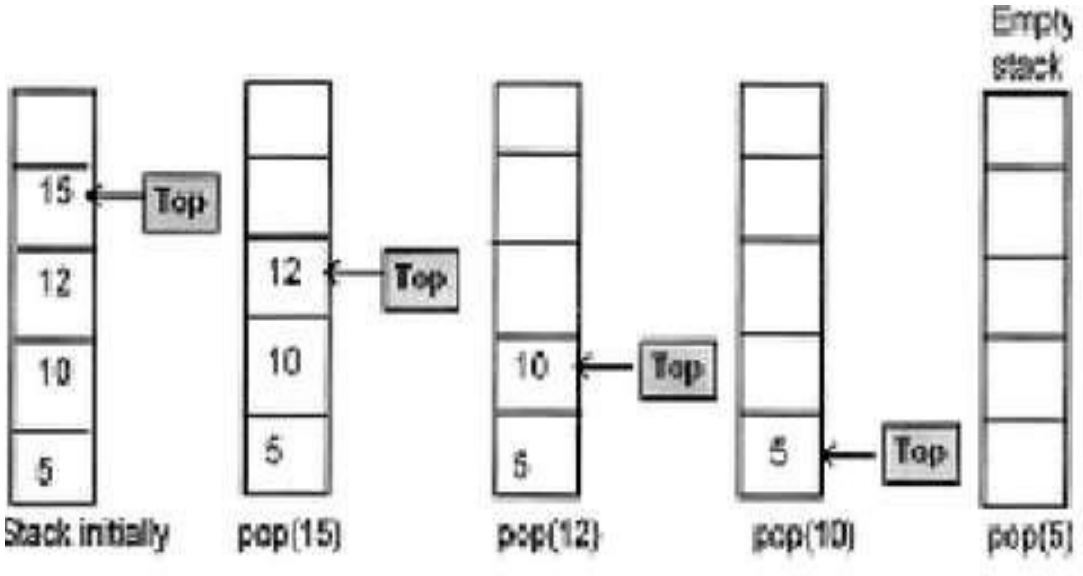
   S[Top] <- x

4.[finish]
   Exit



POP operation

- POP(S,TOP):



1. [check Stack is underflow??] if Top = -1 then
   write(" Stack underflow");
   Exit
2. [get the top most element]

   Temp <- S [top]

3. [Decrement Top of the Stack]
   Top <- Top -1
4. [Return the Poped Element]
   Return Temp

PEEP operation

- PEEP (S,TOP,I):

1. [check Stack is underflow??]

   if Top-I+1 < 0 then

   write(“ Stack underflow”);
   Exit

2. [get the I^{th} element]

   Temp <- S [Top - I + 1]

3 . [Return the I^{th} Element]

   Return Temp

CHANGE operation

- CHANGE (S,TOP,I,X):

1. [check Stack is

   underflow??] if Top-i+1 < 0

   then

   write(" Stack underflow");
   Exit

2. [Change the i^{th} element]

   S [Top - i + 1] <- X

3 .[Finish]
   Exit

Applications of stack:

- String Reversal/String Recognition
- Balanced Parenthesis
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, Photoshop.
- Saving local variables when one function calls another, and this one calls another
- Used in many algorithms like Tower of Hanoi, tree traversals.
- Page-visited history in a Web browser

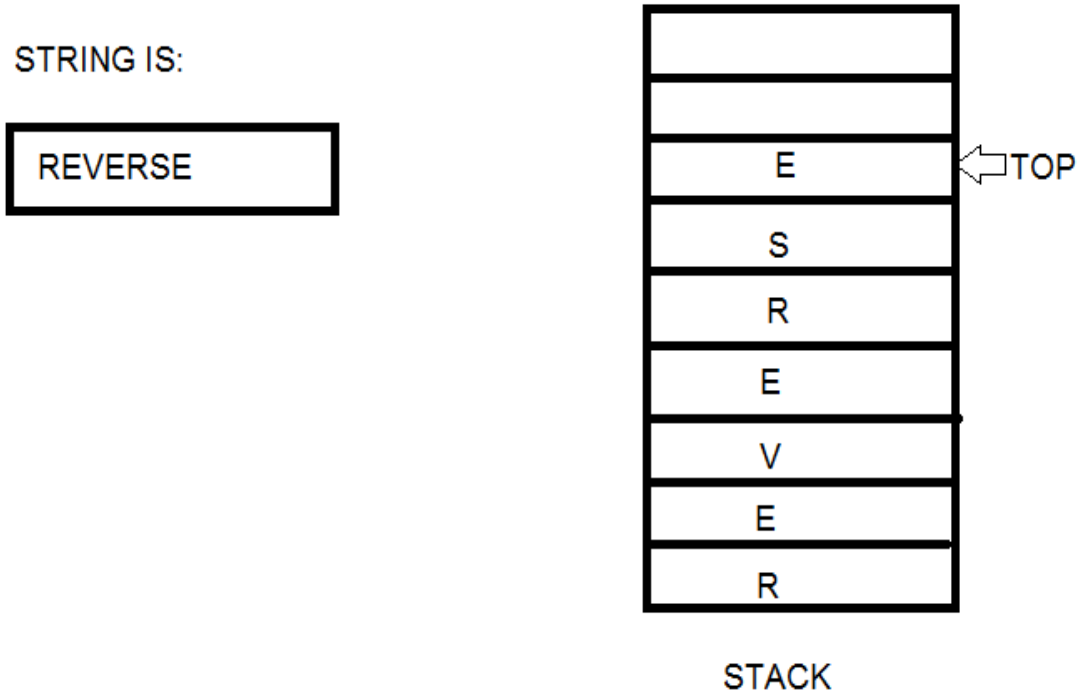
APPLICATIONS OF STACKS ARE:

I. **Reversing Strings:**

- A simple application of stack is reversing strings. To reverse a string , the characters of string are pushed onto the stack one by one as the string is read from left to right.
- Once all the characters of string are pushed onto stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operation results in the reversal of the string.

For example:

To reverse the string 'REVERSE' the string is read from left to right and its characters are pushed . LIKE:



Example :Use of Stack

Example:

String Recognization :

L = { WCW^R |W ∈ { a, b}*} where W^R reverse of String W

Language L define the string of character a and b.

C will act as separator between W and W^R

For example: abCba, aabCbaa, abbbCbbba, -- Valid strings

String Recognization

Steps:

1. Create stack datastructure
2. Read the string from the user
3. Use loop and Push the character onto stack from string one by one until C (separator) encountered.
4. Once "C" occur in string (do not Push C on the Stack). stop push operation.
5. Use loop, Pop one element from the stack and fetch one character from the string after Character C.
6. Check both elements are equals or not . If both are equals then repeat step 5 and 6 until all character in string are fetched and top become -1. if at any moments mismatch occur then stop process. print appropriate message.

Algorithm:

- STRRECO: given the string named str of the form WCW^R on alphabet {a,b,C}.this algorithm determine the whether the input string str is according to the given language rule or not. S indicate stack and top is pointer pointing top of the stack.

- 1. [get the string from the user and initialize the index variable to get character one by one from the string]
- READ(str)
- I <-0
- PUSH(S,top,'#')
- 2.[fetch one character at a time and push on the stack until separator C occur in string] .
- repeat while str[i] not 'C'
- Push(S,top, str[i])
- i <- i+1
-

Algorithm:

3. [scan characters following the 'C' one at a time and pop character from the stack and compare]

```
i < i +1
repeat while s[i] not NULL
    x <- POP (S ,top)
    if X NOT EQUAL to S[i]
     then
         Write ( "invalid String")
         goto step 5
```

4.[compare the top and end of string simulteneously if step 3 is successful] .

```
if S[i]= NULL and S[top] = '#'
then
        write ( "String is valid")
else  write ( "invalid String")
```

Algorithm:

5. [finished the algorithm]

   exit

Using a Stack to Process Algebraic Expressions

- Algebraic expressions composed of
  - Operands (variables, constants)
  - Operators (+, -, /, *, ^)
- Operators can be unary or binary
- Different precedence notations
  - Infix a + b
  - Prefix + a b
  - Postfix a b +

Using a Stack to Process Algebraic Expressions

- Precedence must be maintained
  - Order of operators
  - Use of parentheses (must be balanced)
- Use stacks to evaluate parentheses usage
  - Scan expression
  - Push symbols
  - Pop symbols

II. Checking the validity of an expression containing nested parenthesis:

- Stacks are also used to check whether a given arithmetic expressions containing nested parenthesis is properly parenthesized.
- The program for checking the validity of an
  - expression verifies that for each left parenthesis braces or bracket ,there is a corresponding closing symbol and symbols are appropriately nested.

For example:

VALID INPUTS	INVALID INPUTS
{} ({[]}) {[]()} [{({}[](})}]	{(} ([(())]) {}[]) [{)}(]}]

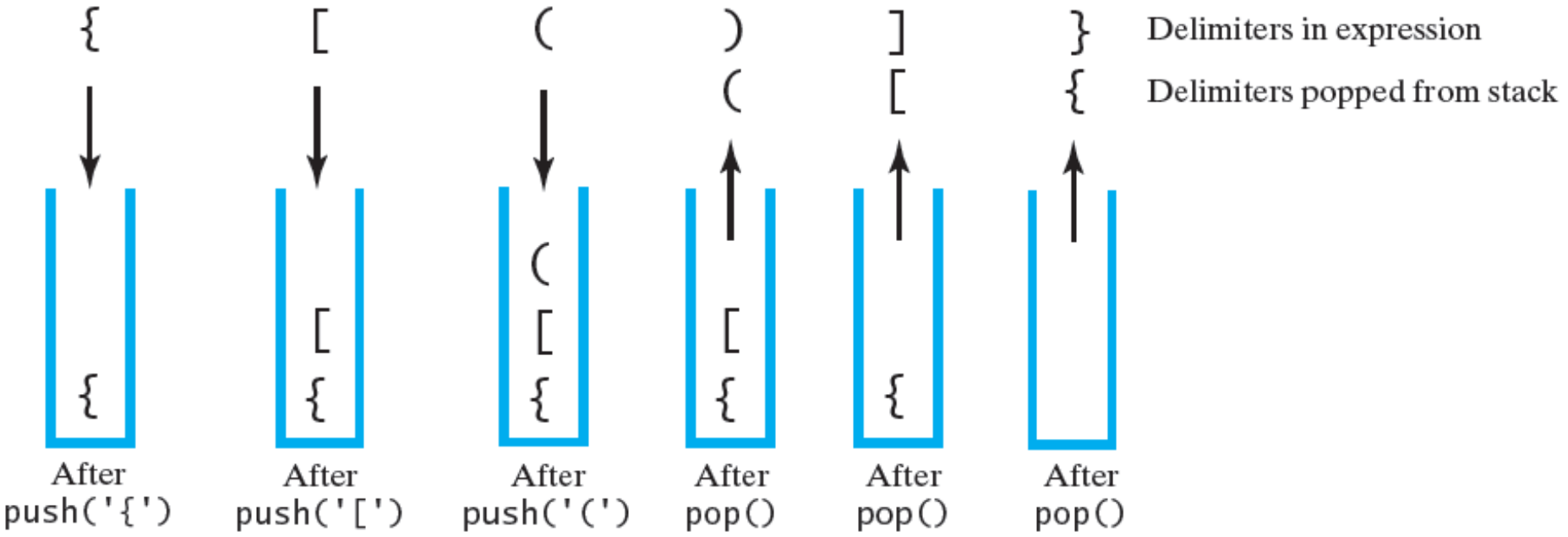


Figure 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }

Figure 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ]) }

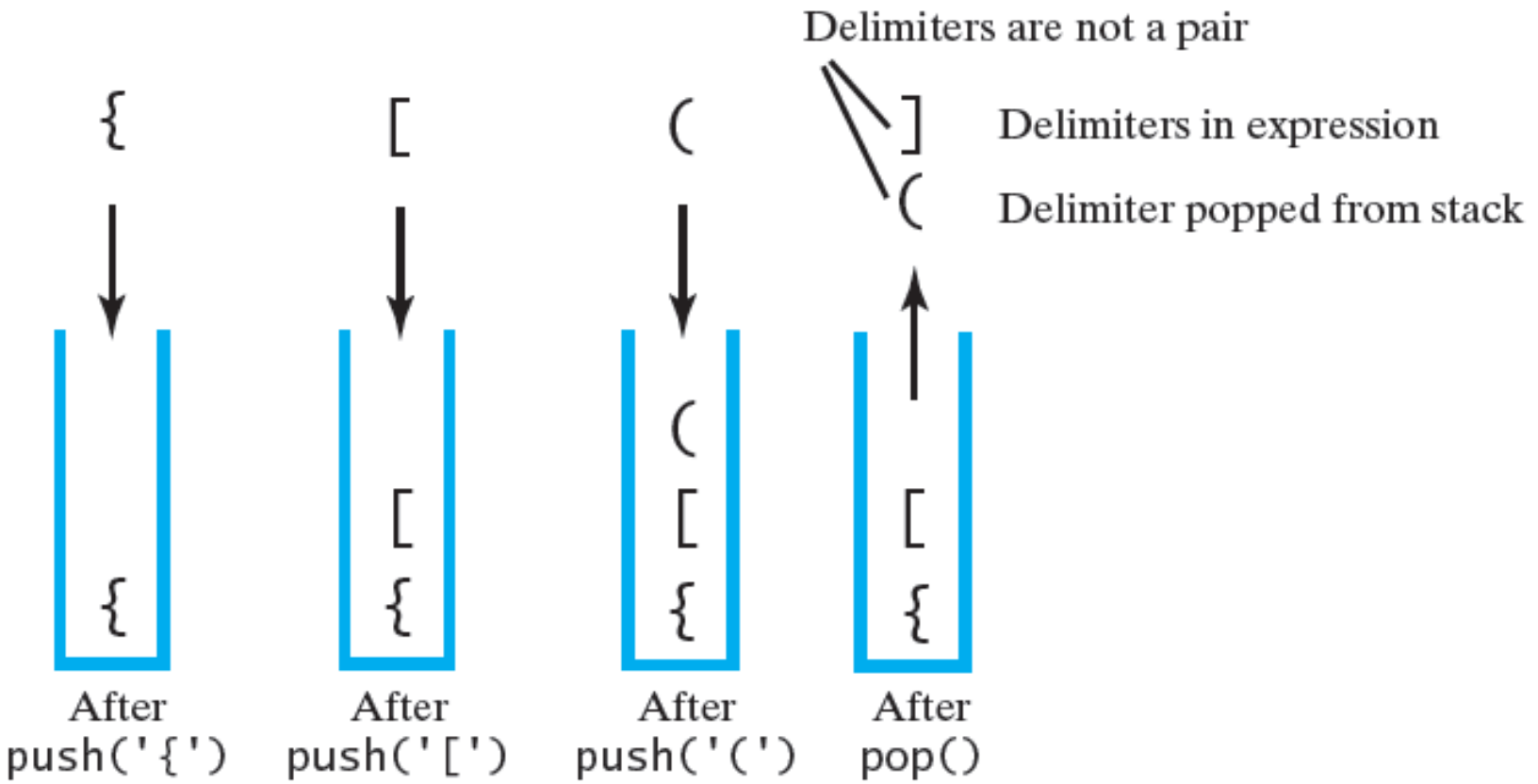
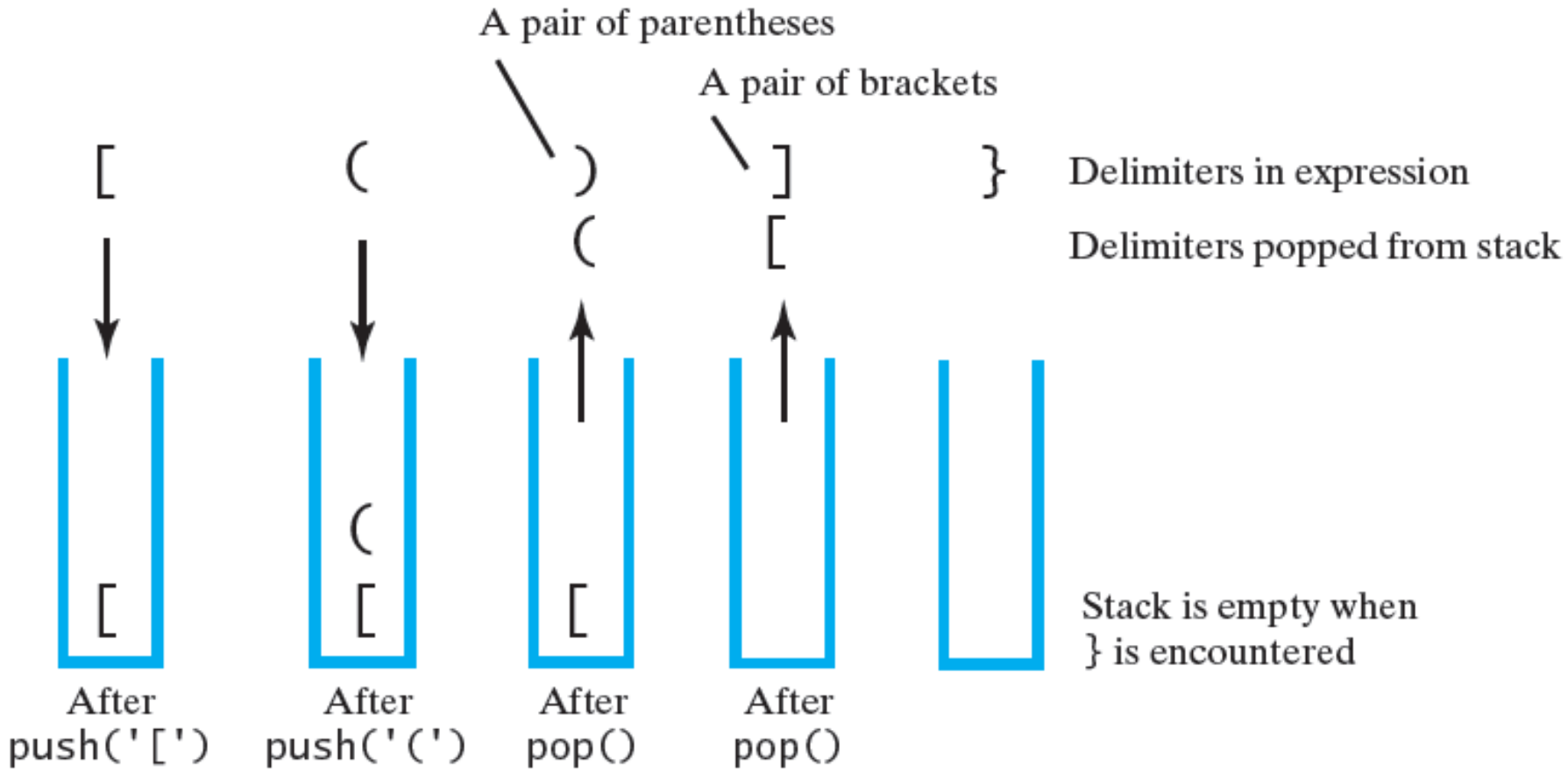


Figure 5-5 The contents of a stack during the scan of an expression that c



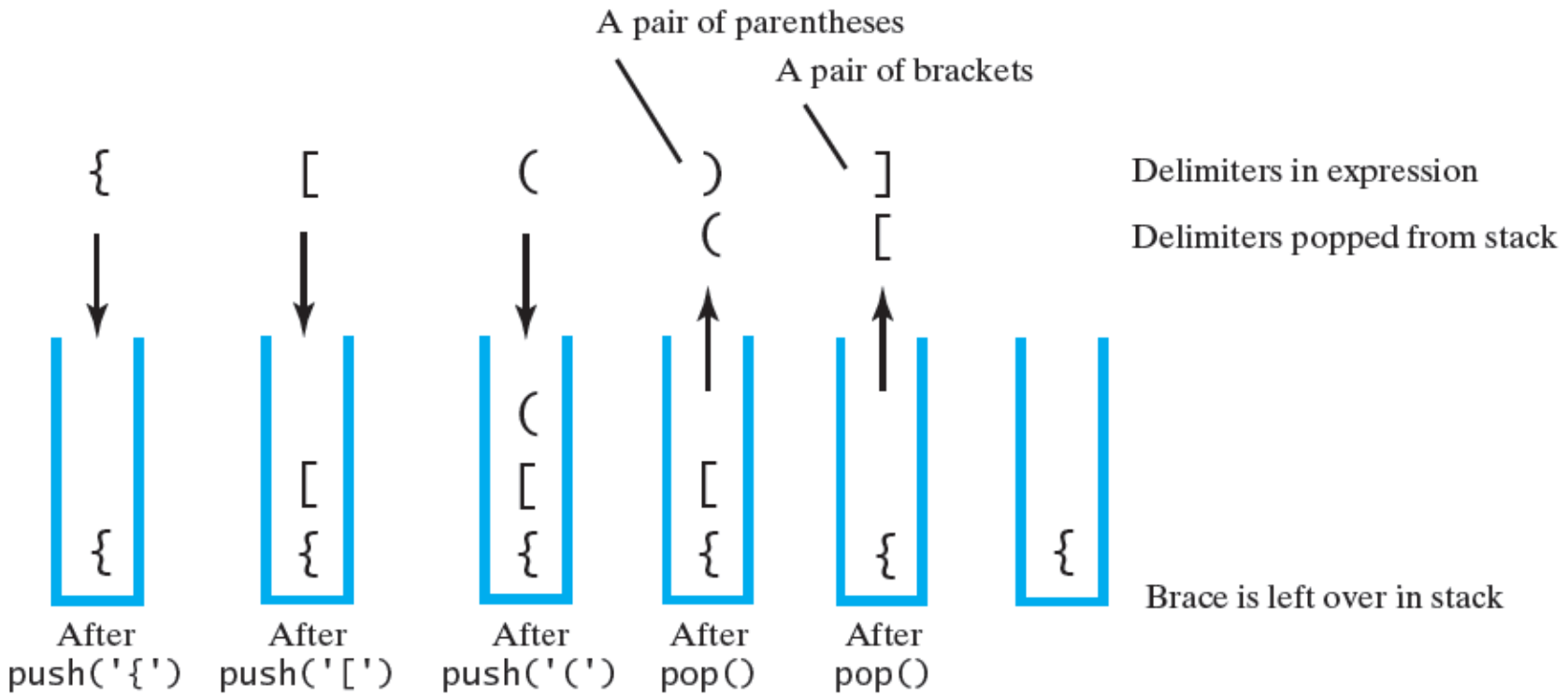


Figure 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]

Expression

[Read the Expression from the user. check is it valid or not ?]

1.
```
READ(S)
I <- 0
SPUSH(S,TOP, '#')
X <-CHK_EXP(S)
if X = 0 then
 then
        Write("invalid Expression")  exit
```

2. [Read one character from the string and do accordingly]

```
Repeat while s[i] not NULL  if
        s[i] = '('
        Then     SPUSH(S  ,top ,s[i])
```

Expression

```
    Else if
            s[i]=‘)
    ” Then
       C  <- SPOP(S, TOP)
        if  C  NOT  EQUAL to ‘(‘
        Then
              write ( “string is not valid”)
               EXIT
    Else
       continue
[3] [Check the termination conditions]
      if  s[i] = NULL and  s[Top] = “#”
      then
            Write (“Valid string”)
```

Expression

```
    else
       Write ( “Invalid String”)
[4]  [finish algorithms]
       Exit
```

Write the C programs for the same. your program should contains followings module.

1. Void main()
2. CHK_EXP(char [])
3. isopd(char)
4. isopt(char)
5. SPUSH(char [],int* ,char)
6. SPOP(char [],int*)

Expression

- What is expression????
- Note: expression is given to the program in form of string
- Valid Expression Rules:

[a+{b-(c-d)*f}/g]

- There should not be 2 consecutive OPD(operand)
- There should not be 2 consecutive OPT.(operator)
- Operand is not followed by opening parenthesis.
- Operator is not immediately followed y closing parenthesis.
- Last symbol should not be a operator.

- If any of the above condition is become true then expression is not a valid expression.

Infix to Postfix

- Manual algorithm for converting infix to postfix
  (a + b) * c
  - Write with parentheses to force correct operator precedence ((a + b) * c)
  - Move operator to right inside parentheses
    ((a b +) * c)
  - Remove parentheses
    a b + c *

Infix to Postfix

- Algorithm basics
  - Scan expression left to right
  - When operand found, place at end of new expression
  - When operator found, save to determine new position

Figure 5-7 Converting the infix expression **a + b * c** to postfix form

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Figure 5-8 Converting an infix expression to postfix form: a - b + c

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
–	*a*	–
b	*a b*	–
+	*a b* –	
	a b –	+
c	*a b* – *c*	+
	a b – *c* +	

Figure 5-8 Converting an infix expression to postfix form: a ^ b ^ c

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Infix to Postfix Conversion

1. Operand
   - Append to end of output expression
2. Operator ^
   - Push ^ onto stack
3. Operators +, -, *, /
   - Pop from stack, append to output expression
   - Until stack empty or top operator has lower precedence than new operator
   - Then push new operator onto stack

Infix to Postfix Conversion

4. Open parenthesis
   - Push (onto stack
5. Close parenthesis
   - Pop operators from stack and append to output
   - Until open parenthesis is popped.
   - Discard both parentheses

FIGURE 5-9 The steps in converting the infix expression a / b * (c + (d - e)) to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*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 –*	* (+ (
	a b / c d e –	* (+
)	*a b / c d e – +*	* (
	a b / c d e – +	*
	*a b / c d e – + **	

Evaluating Postfix Expressions

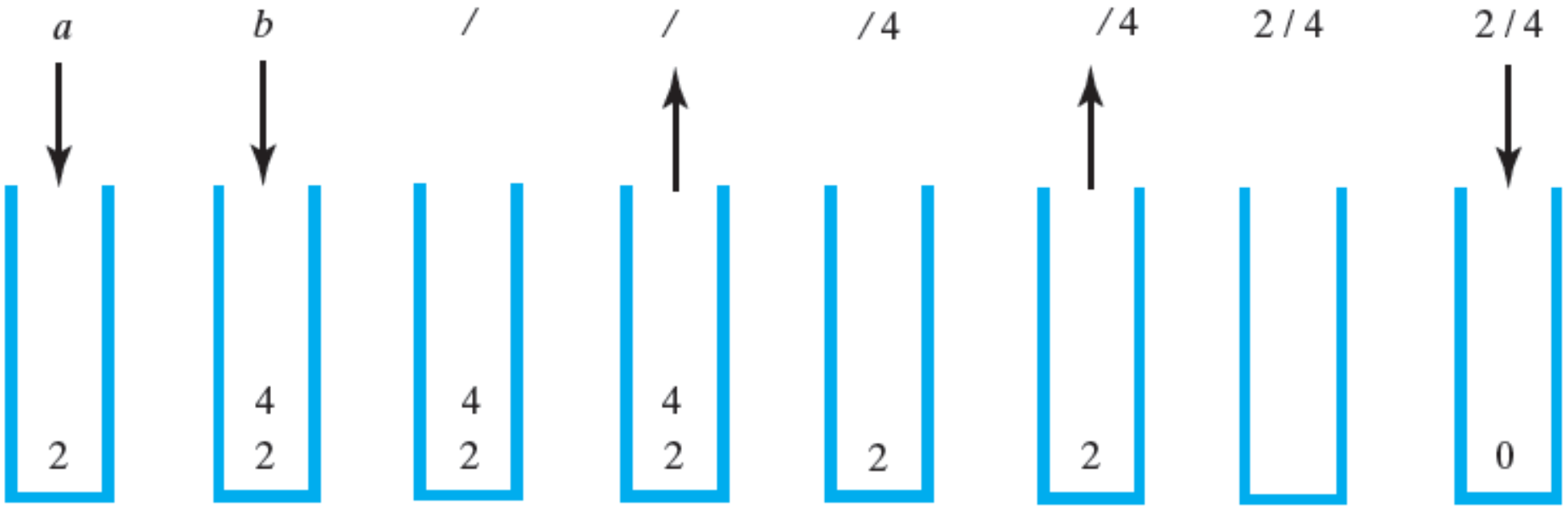


FIGURE 5-10 The stack during the evaluation of the postfix expression a b / when a is 2 and b is 4

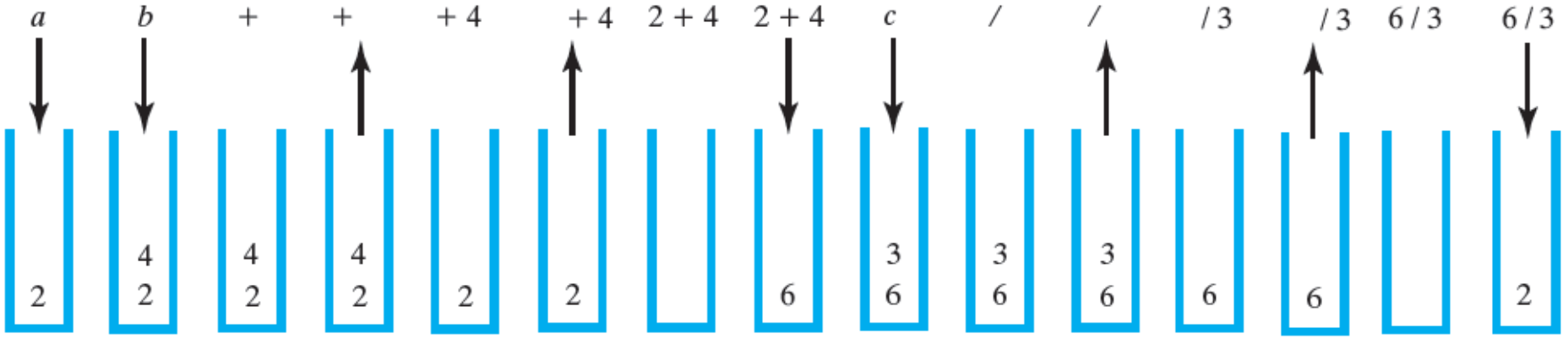


FIGURE 5-11 The stack during the evaluation of the postfix expression a b + c / when a is 2, b is 4, and c is 3

Evaluating Infix Expressions

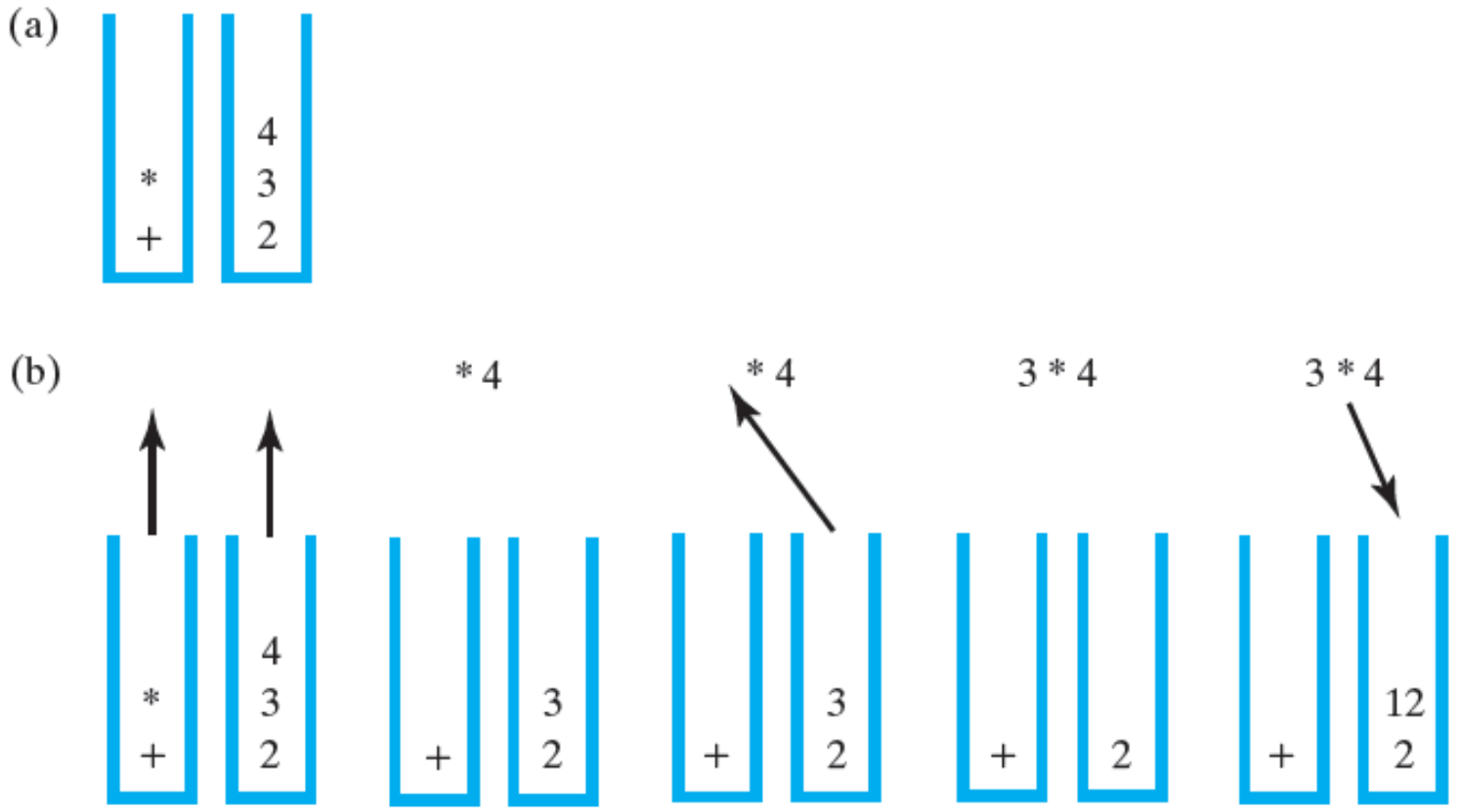


FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (a) after reaching the end of the expression; (b) while performing the multiplication;

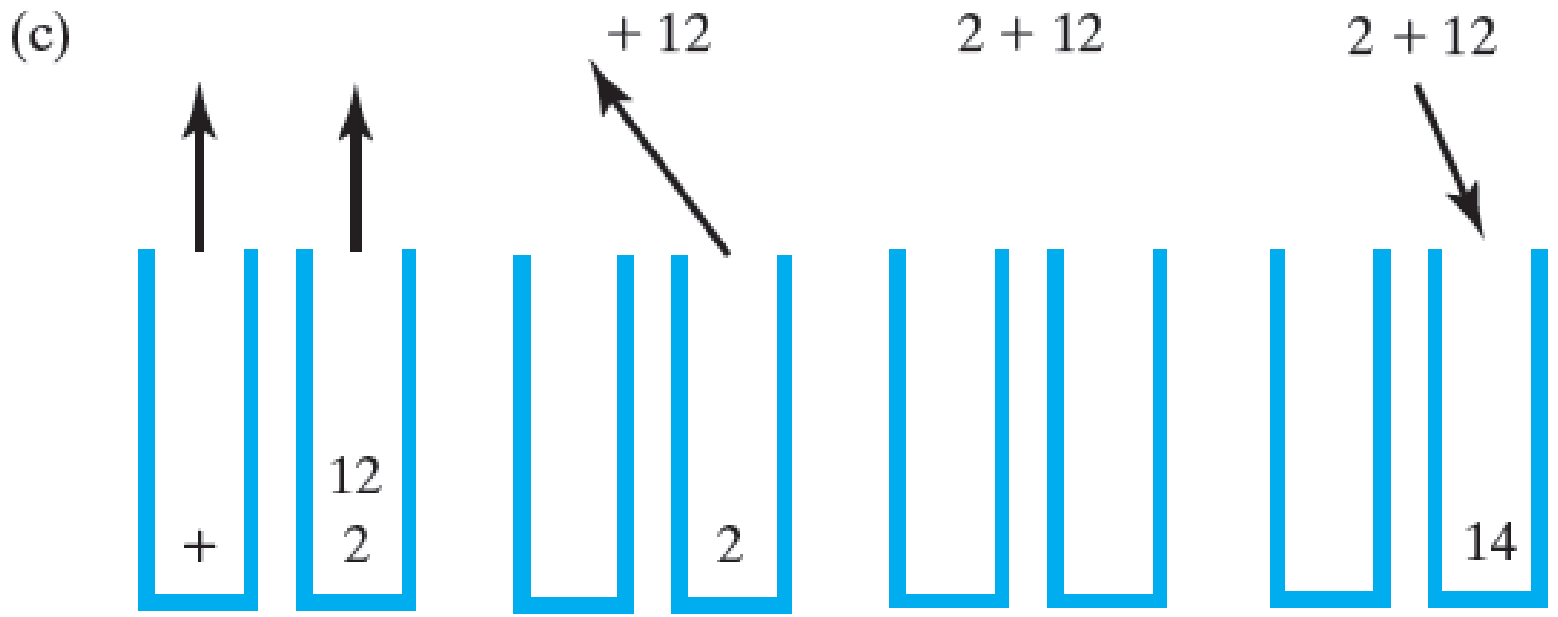


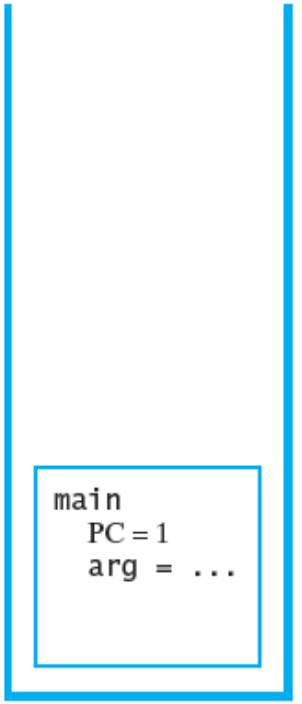
FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4: (c) while performing the addition

The Program Stack

```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120      methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```



(a)

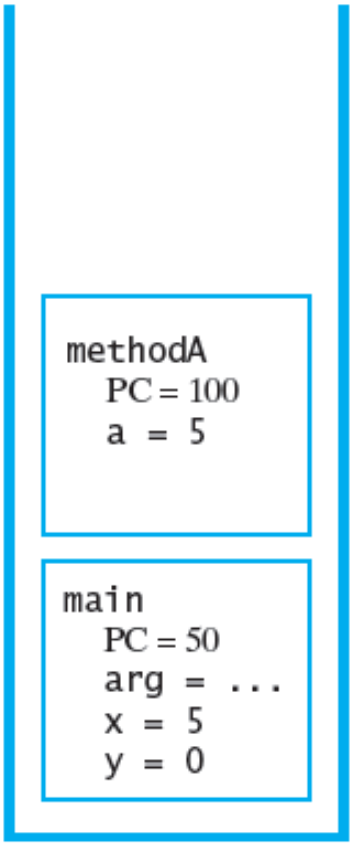
FIGURE 5-13 The program stack at three points in time:
(a) when **`main`** begins execution; (PC is the program counter)

The Program Stack

```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120      methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```



(b)

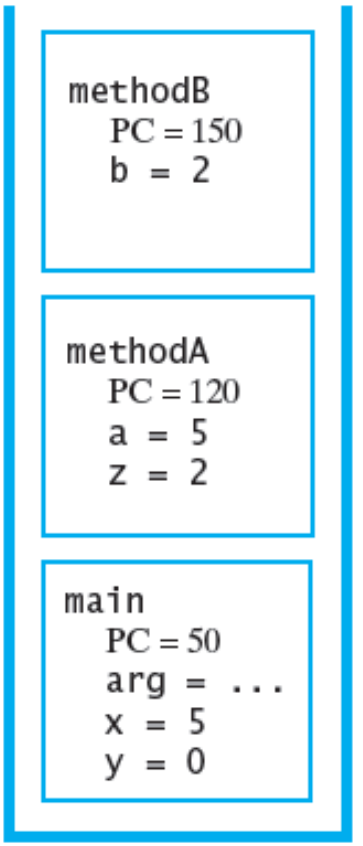
FIGURE 5-13 The program stack at three points in time:
(b) when **`methodA`** begins execution; (PC is the program counter)

The Program Stack

```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120      methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```



(c)

FIGURE 5-13 The program stack at three points in time:
(c) when **`methodB`** begins execution; (PC is the program counter)

Expression

- How do we check given character is operand or operator?
- Assignment :
- Create C programs which checks expression given as input is valid or not?
- Your program should consist of followings modules
- void main()
- int CHK_EXP(char [])
- int isoperand(char)
- int isoperator(char)

Expression

- Stack application:

- ASSIGNMENT :Write a algorithm to check given expression contains the valid parenthesis or not?????

- Two conditions must besatisfied

    1. expression is validexpression

    2. Numbers of opening parenthesis equals to numbers of rightparenthesis.

Steps:

Expression: Infix ,Prefix, Postfix

- Expression is a statement consist of operator and operands
- Operands may be a constant value or may be variables.
- There are three way(Notations) to represent expressions.

Infix Expression, Prefix Exp, Postfix exp

These notations are due to the relatives positions of the operators with respect to their operands

INFIX:

operator is appear between their operands. Exm: a+b

PREFIX:

Expression: Infix , Prefix, Postfix cont..

.

Postfix:

operator is appear after their operands. For exa:
ab+

Conversions: Infix to Postfix, infix to Prefix

Infix to Postfix:

- This conversion also called reverse polish notation or suffix conversion.
- Basic rule is to put the operator after their respective operators.
- For example: if infix expression is X + Y then

RPT is XY+

Infix to Postfix: cont..

.

- If more then one operators appear in the expression then
- Operators precedence(priority) rule would be considered.
- Operator having highest priority is converted first in suffix notation. Then operator having second highest converted next .so on.
- Operator having lowest priority is would be converted at last.

- For Example:

So, a + b * c

1. = a + (bc*)
2. = abc* +

Infix to Postfix: cont..

- If more then one operators appear in the expression and

  Operators having same priority then go left to right and first come first serve bases first operator is converted in suffix notation.

- For Example:

a + b * c / d

So,

= a + (bc*) / d

= a + bc*d /

= abc*d/ +

Use of Stack in Infix to Postfix conversion

- Stack is used as a intermediate storage in such a application.
- while conversion, stack is used to store the some unprocessed symbols

Example:

a + b * c / d

So,

= a + (bc*) / d

= a + bc*d /

= abc*d/ +

Evaluation Postfix Expression

Steps:

1. First Convert Infix expression to Postfix expression.
2. Start scanning the postfix expression from left to right.
3. If the element is an operand, push it into the stack.
4. If the element is an operator, pop two operands from the stack. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back to the stack. Repeat the same process until all the elements are scanned.
5. When the expression is ended, the number in the stack is the final answer.

Evaluation Postfix Expression

algorithm:

```
while (not end of input)
   symbol = input symbol
   if (symbol is operand)
      push(operand_stack, symbol)
   else
      operand2 = pop(operand_stack)
      operand1 = pop(operand_stack)
      result = apply symbol to operand1 and
   operand2    push(operand_stack, result)
return pop(operand_stack)
```

Evaluation Postfix Expression

Example:

Let the given expression be “2 3 1 * + 9 -“. We scan all elements one by one.

1) Scan ‘2’, it’s a number, so push it to stack.

Stack

contains ‘2’

2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3′

3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1′

4) Scan “*”, it’s an operator, pop two operands from

5) Scan '+', it's an operator, pop two operands from

stack, apply the + operator on operands, we get 3

+ 2 which results in 5. We push the result '5' to

stack. Stack now becomes '5'.

6) Scan '9', it's a number, we push it to the stack.

Stack now becomes '5 9'.

7) Scan '-', it's an operator, pop two operands from

stack, apply the - operator on operands, we get 5

- 9 which results in -4. We push the result '-4'

Example:

postfix expression: 6 2 3 + - 3 8 2 / + * 2 ^ 3 +

Evaluation Postfix Expression

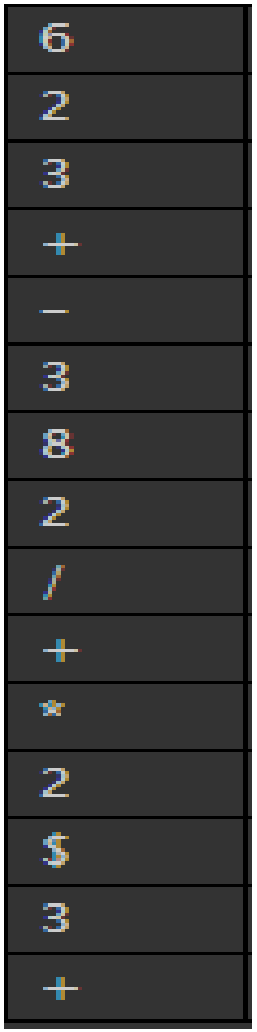
ch	opnd1	opnd2	value	operandstk
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
–	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

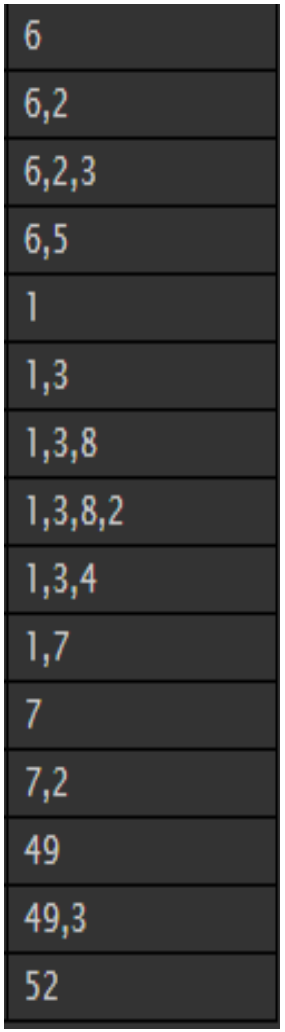
Example:

postfix expression: 6 2 3 + - 3 8 2 / + * 2 ^ 3 +

scanned string stack output string

Evaluation Postfix Expression





Example 3.4.3 : Evaluate the following postfix expression and show stack after every step in tabular form. ABC + *DE\–
Given A = 5, B = 6, C = 2, D = 12, E = 4

Solution :

Input	stack
ABC + * DE/–	
BC + *DE/–	5
C + * DE/–	6 5
+ * DE/–	2 6 5
* DE/-	8 5
DE/–	40
E/–	12 40
/–	4 12 40
–	3 40
	37

End

value of the expression = 37

Use of Stack in Infix to Postfix conversion cont.

- Stack Simulation: Infix to postfix

Given: input string, output string, Stack

Steps:

1. Scan one symbol at a time from the input string in Left to Right .say it is current symbol.
2. If current symbol is operand then store it into output string.
3. If current symbol is operator then check the priority level of current symbol and top of the stack symbol. If the priority of top of the stack symbol is greater then or equal to current symbol then pop that symbol and append to the output string. Repeat the same process until you get the symbol on top of the stack whose precedence level is lower then current symbol. Finally, Push the current symbol on the stack.

Use of Stack in Infix to Postfix conversion cont.

.

4. When null encountered in input string then pop all symbols from stack and append them to the end of

   output string.

NOTE:

1. Consider step 2 & 3 in form of else if ladder.
2. only operators will be store on the stack. operands will never be store on the stack.

Conversion of P * Q^R + S

Expression	Stack	Output	Remark
P*Q ∧ R + S	NULL	–	–
*Q∧R + S	NULL	P	Operand must be printed
Q∧R + S	*	P	* operation will be performed if the next operator is of lower or equal precedence.
∧R + S	*	PQ	Operand must be printed
R + S	* ∧	PQ	* Can not be performed as ∧ has higher precedence
+ S	* ∧	PQR	Operand must be printed
+ S	NULL	PQR∧*	All higher precedence operators (compare to +) are popped and printed and finally the current operator is pushed on top of the stack.
S	+	PQR∧*	–
NULL	+	PQR∧* S	Operand must be printed
Null	Null	PQR∧* S +	Finally, all operators are popped and printed.

input string : **a + b - c * d – e + f**

Scanned symbol	Stack contents	outputstring
	#	
a		
+		
b		
-		
c		
*		
d		
-		
e		
+		
f		

Convert Infix To Prefix Expression

Steps:

1. Reverse the input string.
2. Start scanning the expression from left to right.
3. If the element is an operand, push it into the stack.
4. If it is closing parenthesis, push it on the stack. Regardless of the current symbol on the stack.
5. If the element is an operator, then

   a) if stack is empty, push on stack.

   b)else remove and output all stack symbol whose precedence values are greater then to the precedence of the current input symbol. Finally push current symbol on stack.

   c)if it is a opening parenthesis, pop operator from stack and add it to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.

Convert Infix To Prefix Expression

Steps:

6. Repeat steps 3,4,5 until last symbol is scanned from the input string.
7. The remaining operator pop from the stack and add it to the output string.
8. Reverse the output string and print it.

<u>Note:</u>

on stack -> ‘)’ has low priority

in infix -> ‘)’ has highest priority

never push ‘(‘ on stack.

Infix to Prefix

We have infix expression in the form of:

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

Now reading expression from right to left and pushing operators into stack and variables to output stack

Input	Output_stack	Stack
)	EMPTY	)
G	G	)
+	G	)+
F	GF	)+
(	GF+	EMPTY
-	GF+	-
)	GF+	-)
)	GF+	-))
E	GF+E	-))
+	GF+E	-))+
D	GF+ED	-))+
(	GF+ED+	-)
*	GF+ED+	-)*
C	GF+ED+C	-)*
+	GF+ED+C*	-)+
)	GF+ED+C*	-)+)
B	GF+ED+C*B	-)+)
-	GF+ED+C*B	-)+)-
A	GF+ED+C*BA	-)+)-
(	GF+ED+C*BA-	-)+
(	GF+ED+C*BA-+	-
EMPTY	GF+ED+C*BA-+-	EMPTY

Infix to Prefix

Out put_stack = **GF+ED+C*BA-+-**

Reversing the output_stack we get prefix expression: **-+-AB*C+DE+FG**

Task b:

Evaluating the prefix expression:
Assigning the values to variables

A=7	B=6	C=5	D=4	E=3	F=2	G=1

Reading expression from right to left:

reading	operation	Stack_output
1	1	Empty
2	12	
+	12+	3
3	3 3	3
4	334	3
+	3 3 4+	3 7
5	3 7 5	3 7
*	3 7 5*	3 7 35
6	3 35 6	3 35
7	3 35 6 7	3 35
-	3 35 6 7-	3 35 1
+	3 35 1+	3 36
-	3 36 -	33

Pop stack_output : 33

Result=33

Recursion

- What's behind this function?

```
int f(int a){
      if (a==1)
        return(1);
      else
        return(a * f(a-1));
}
```

It computes f! (factorial)

Factorial:

$$a! = 1 * 2 * 3 * \ldots * (a-1) * a$$

Note:

$$a! = a * (a-1)!$$

remember:

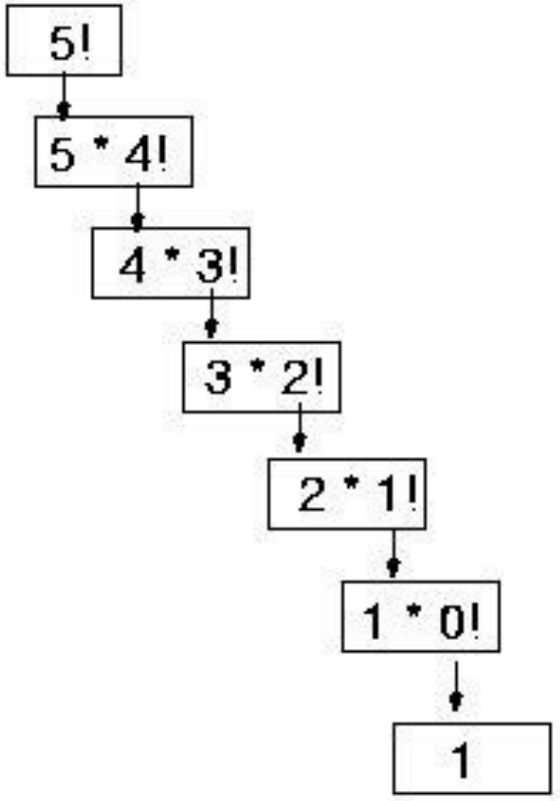
...splitting up the problem into a smaller problem of the same type...

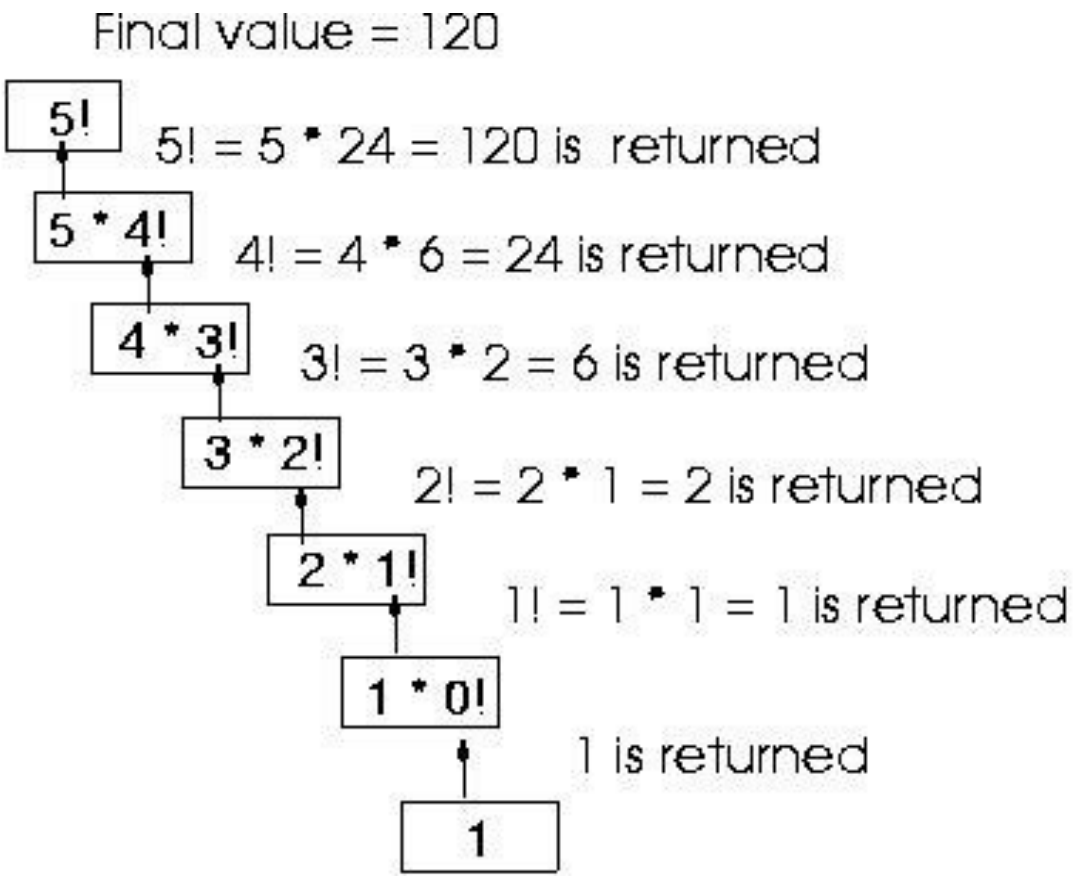
a!

a * (a-1)!

Tracing the example

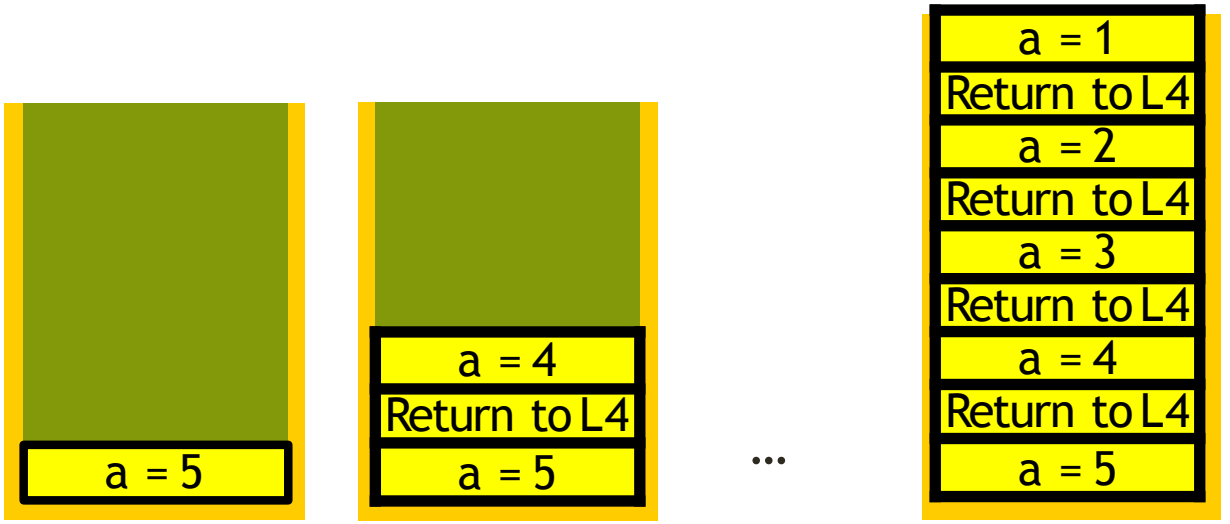
```
public int factorial(inta)
{
        if (a==0)
          return(1);
        else
          return(a *  factorial(a-1));
}
```





Watching the Stack

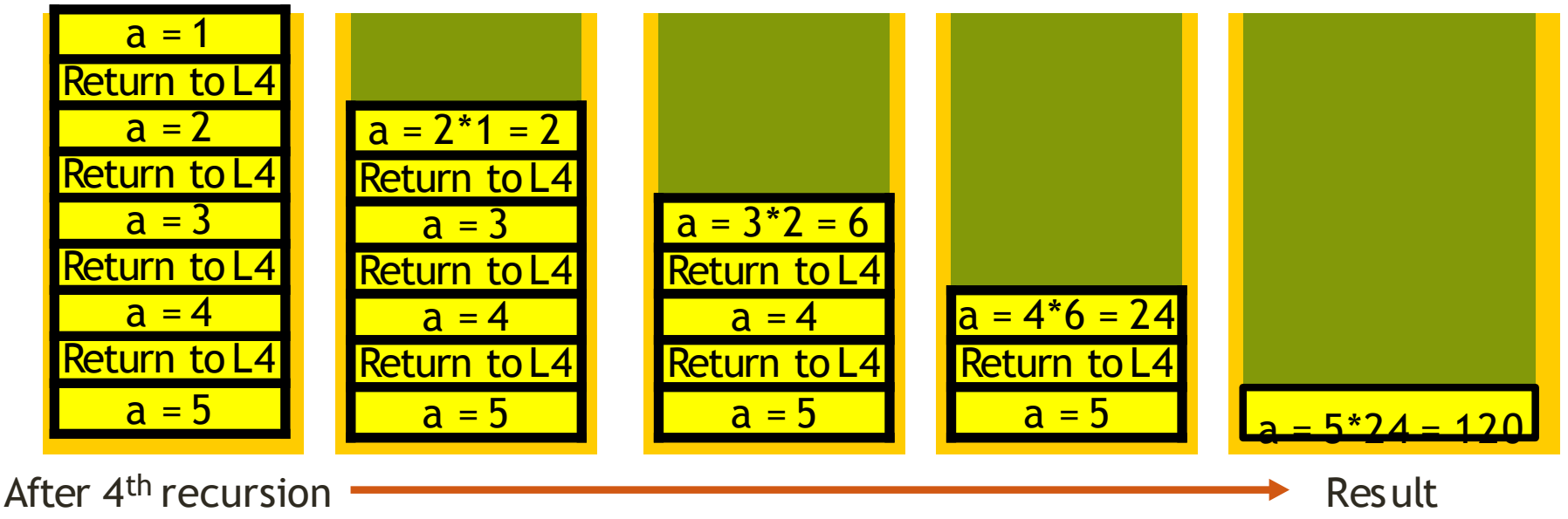
```
public int factorial(inta){
        if (a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```



Every call to the method creates a new set of local variables !

Watching the Stack

```
public int factorial(inta){
        if (a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```



Towers of Hanoi

The Tower of Hanoi is a mathematical game or puzzle. It consists of three pillars A,B and C. There are N discs of decreasing size so that no two discs are of the same size. Initially all the discs are stacked on one pillar in their decreasing order size. Let this pillar be A. other two pillars are empty. The objective of the puzzle is to move all the discs from one pillar to other using third pillar as auxiliary, obeying the following rules:

1) Only one disk must be moved at a time.
2) Each move consists of taking the upper disk from one of the pillar and sliding it onto another pillar, on top of the other disks that may already be present on that pillar.
3) No disk may be placed on top of a smaller disk.

- Solution:
- Consider three pillars name A,B and C.

  A is source pillar, C is Destination pillar and B is intermediate pillar.

So we want to transfer N discs from A to C using B.

Steps:

1) Move N-1 discs from A to B using C.
2) Move discs N from A to C.
3) Move N-1 discs from B to C using A.

<u>Algo:</u>

MOVE(N,ORG,INT,DES)

1) If N >0 then

   MOVE(N-1,ORG,DES,INT)

   ORG-> DES (MOVE from ORG to DES)

   MOVE(N-1,INT,ORG,DES)

   end if

For $N = 3$, how this recursion will solve the problem, is shown in Figure 4.9.

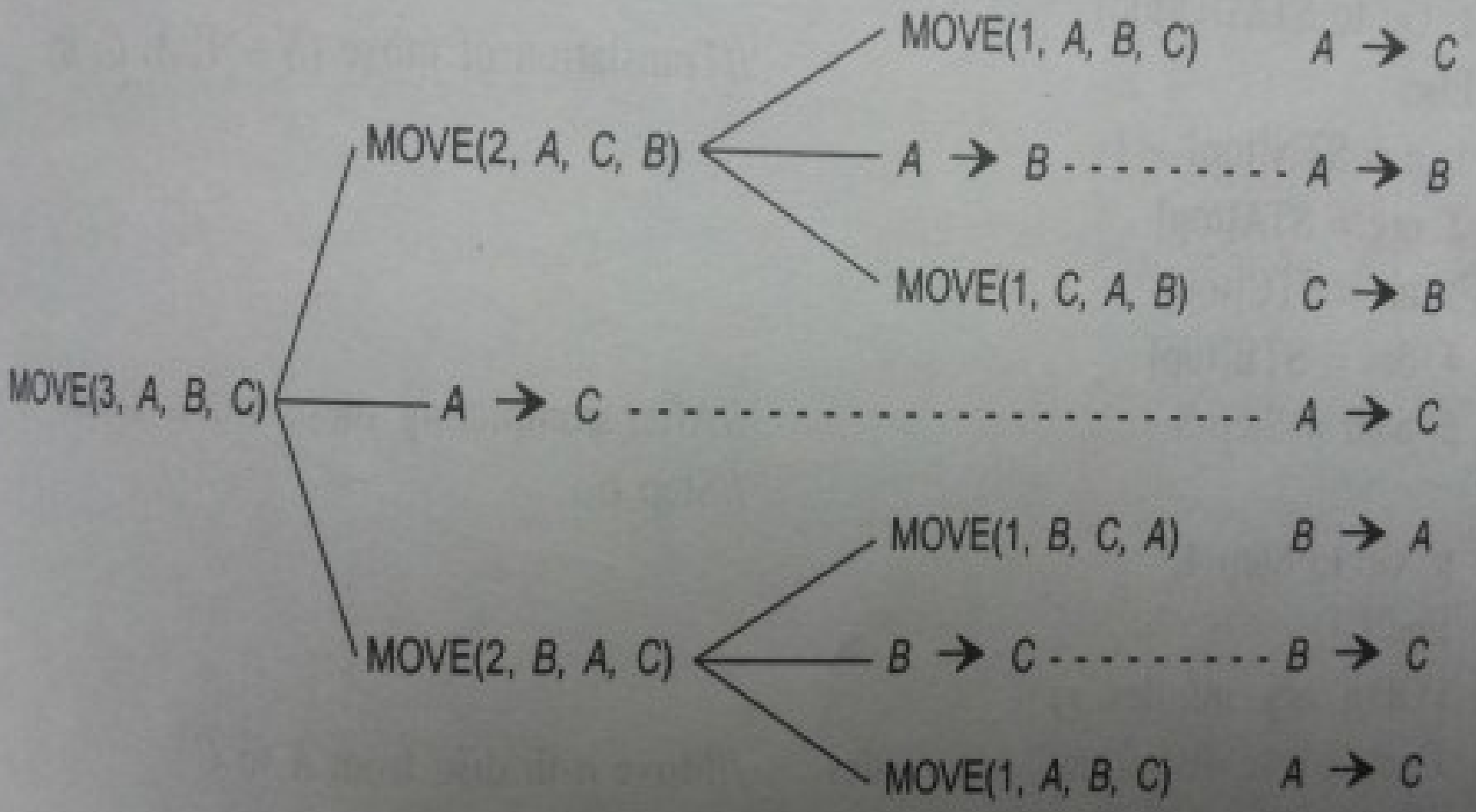
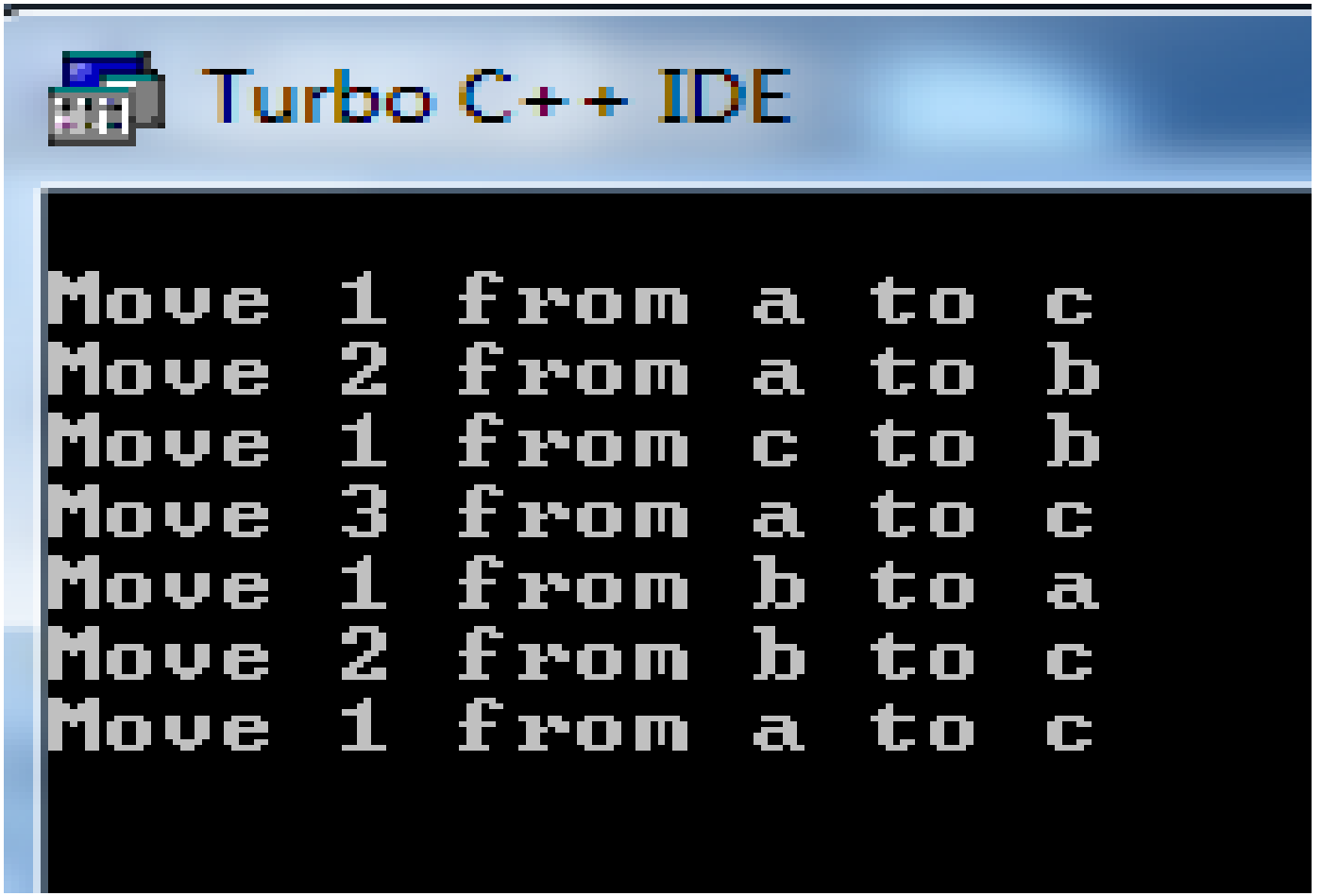


Fig 4.9 Tower of Hanoi (with $N = 3$) solution with recursion.

Prog:

```
void hanoi(int n, char s, char m, char d)
{
    if(n!=0)
    {
        hanoi(n-1,s,d,m);
        printf("\nMove %d from %c to %c",n,s,d);
        hanoi(n-1,m,s,d);
    }
}

void main()
{
    int n=3;
    char s='a';
    char d='c';
    char m='b';
    clrscr();
    hanoi(n,s,m,d);
    getch();
}
```


Turbo C++ IDE
Move 1 from a to c
Move 2 from a to b
Move 1 from c to b
Move 3 from a to c
Move 1 from b to a
Move 2 from b to c
Move 1 from a to c