Unit-III

STACKS

- Stack is an important data structure which stores its elements in an ordered manner. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 7.1. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position. A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out. Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

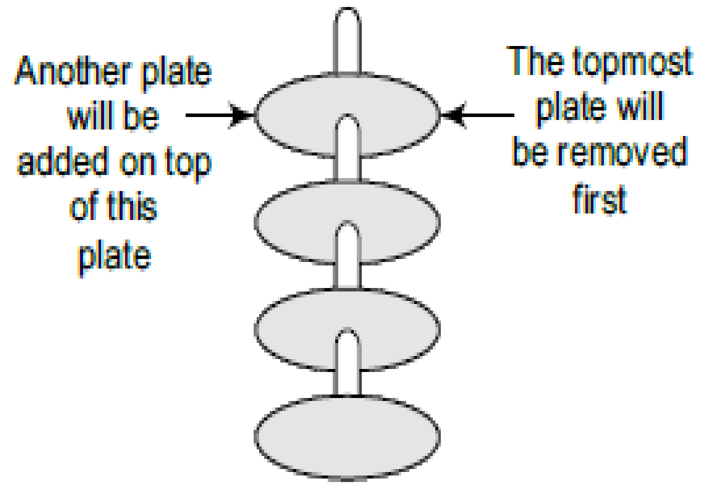
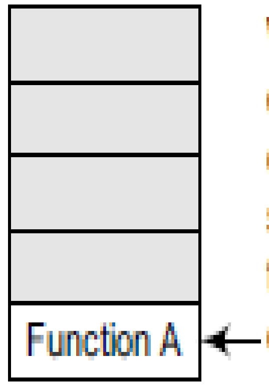
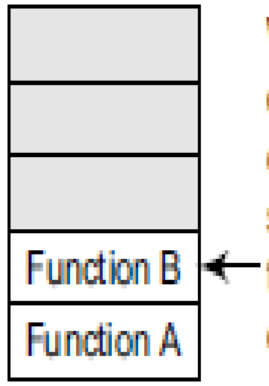

Another plate will be added on top of this plate
The topmost plate will be removed first

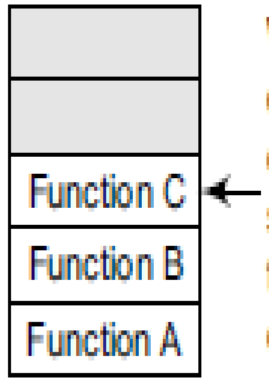
Figure 7.1 Stack of plates



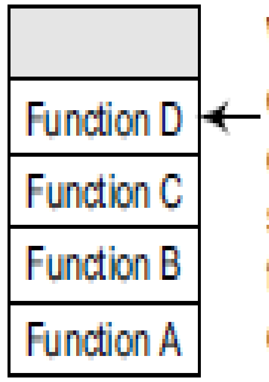
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.



When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.



When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.



When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Figure 7.2 System stack in the case of function calls

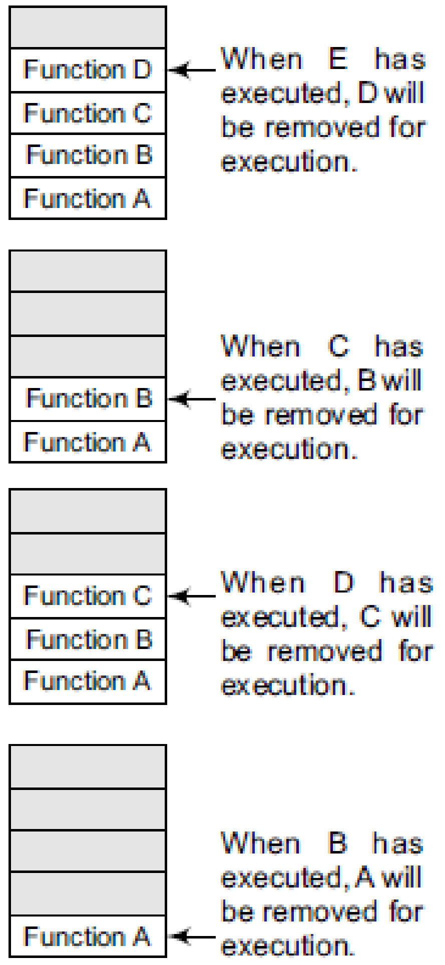

Function D
Function C
Function B
Function A
When E has executed, D will be removed for execution.
Function B
Function A
When C has executed, B will be removed for execution.
Function C
Function B
Function A
When D has executed, C will be removed for execution.
Function A
When B has executed, A will be removed for execution.

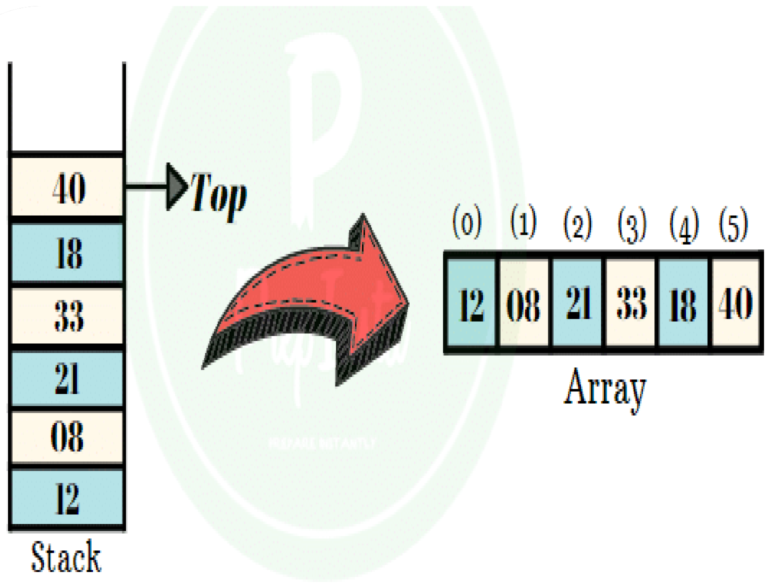
Figure 7.3 System stack when a called function returns to the calling function

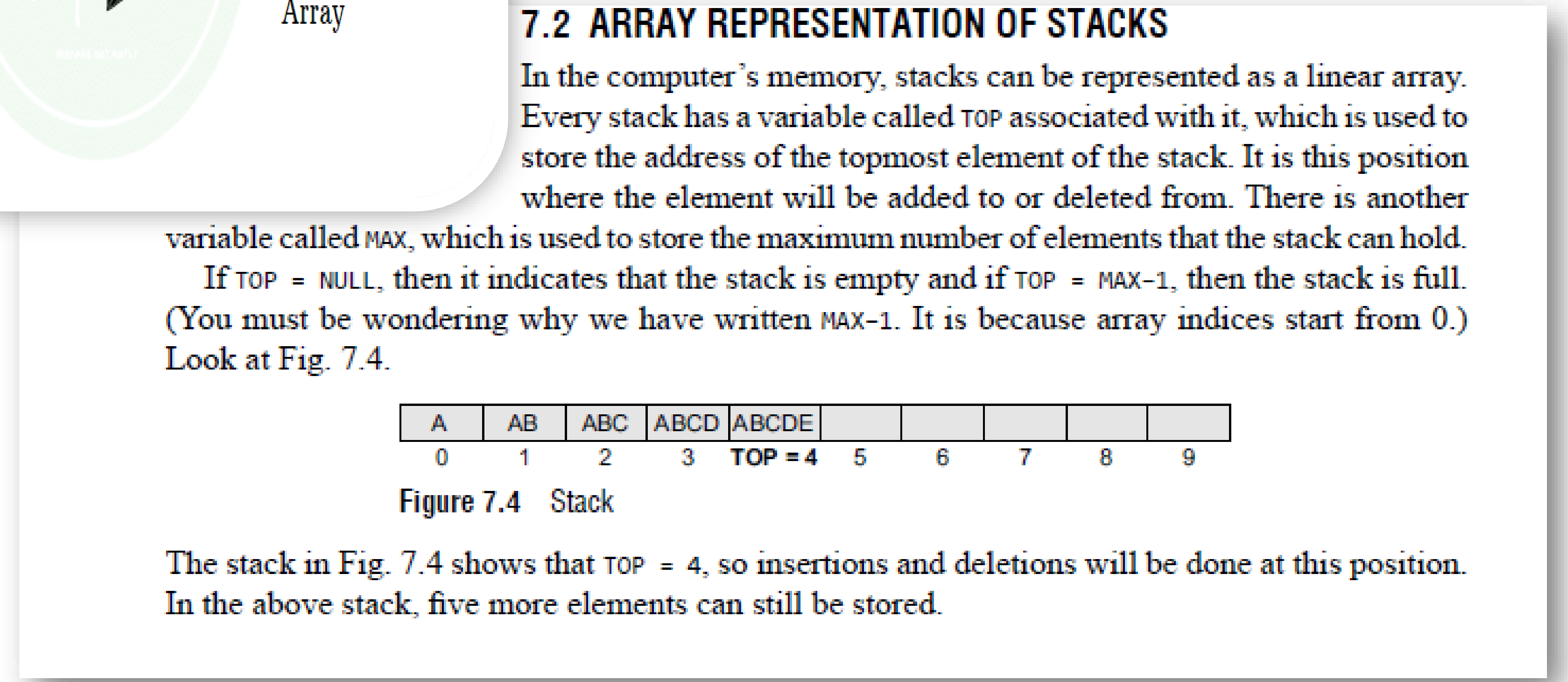
In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function. Look at Fig. 7.2 which shows this concept.

Now when function E is executed, function D will be removed from the top of the stack and executed. Once function D gets completely executed, function C will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed. This is shown in Fig. 7.3.

The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks.





7.2 ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called `TOP` associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called `MAX`, which is used to store the maximum number of elements that the stack can hold.

If `TOP = NULL`, then it indicates that the stack is empty and if `TOP = MAX-1`, then the stack is full. (You must be wondering why we have written `MAX-1`. It is because array indices start from 0.) Look at Fig. 7.4.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	**TOP = 4**	5	6	7	8	9

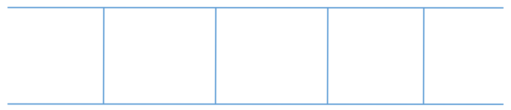
Figure 7.4 Stack

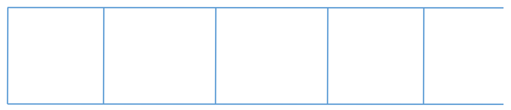
The stack in Fig. 7.4 shows that `TOP = 4`, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

STACKS VS ARRAYS

- An array is a contiguous block of memory.
- A stack is a first-in-last-out data structure with access only to the top of the data.
- Since many languages does not provide facility for stack, it is backed by either arrays or linked list.
- The values can be added and deleted on any side from an array.
- But in stack, insertion and deletion is possible on only one side of the stack. The other side is sealed.

Eg: a[10] –array a[10] - stack





Data Structure Of Stack

- CREATE (*S*) which creates *S* as an empty stack;
- ADD (*i,S*) which inserts the element *i* onto the stack *S* and returns the new stack;
- DELETE (*S*) which removes the top element of stack *S* and returns the new stack;
- TOP (*S*) which returns the top element of stack *S*;
- ISEMTS (*S*) which returns true if *S* is empty else false;

```
structure STACK (item)
1 declare CREATE ( ) ->   stack
2 ADD (item, stack)   -> stack
3 DELETE (stack)   -> stack
4 TOP (stack)  ->  item
5 ISEMTS (stack)->  boolean;
6 for all S   ∈ stack, i ∈ item let
7 ISEMTS (CREATE) :: = true
8 ISEMTS (ADD (i,S)) :: = false
9 DELETE (CREATE) :: = error
10 DELETE (ADD (i,S)) :: = S
11 TOP(CREATE) :: = error
12 TOP(ADD(i,S)) :: = i
13 end
end STACK
```

7.3 OPERATIONS ON A STACK

A stack supports three basic operations: push, pop, and peek. The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack. The peek operation returns the value of the topmost element of the stack.

7.3.1 Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed. Consider the stack given in Fig. 7.5.

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

Figure 7.5 Stack

To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as shown in Fig. 7.6.

1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9

Figure 7.6 Stack after insertion

```
Step 1: IF TOP = MAX-1
              PRINT "OVERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

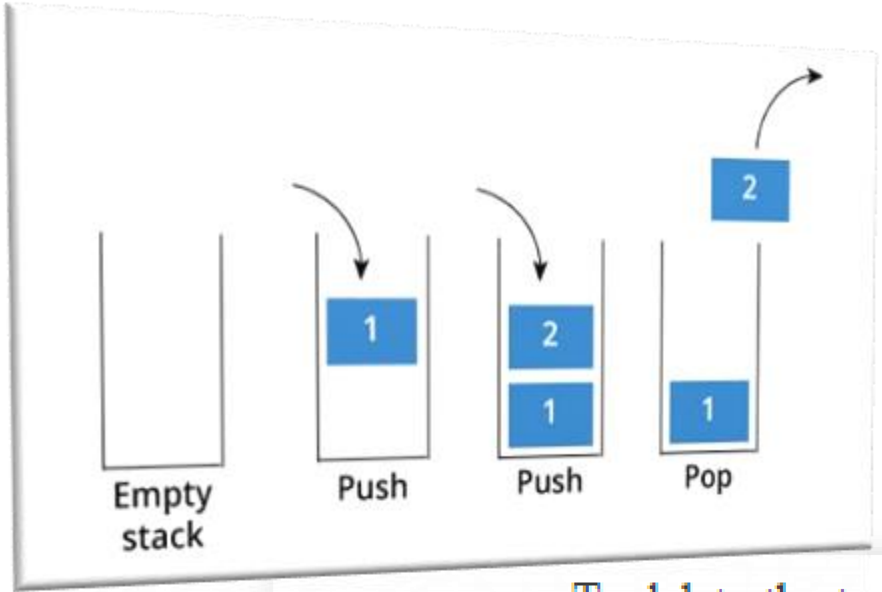
Figure 7.7 shows the algorithm to insert an element in a stack. In Step 1, we first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by TOP.

7.3.2 Pop Operation

The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack given in Fig. 7.8.

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

Figure 7.8 Stack



To delete the topmost element, we first check if `TOP=NULL`. If the condition is false, then we decrement the value pointed by `TOP`. Thus, the updated stack becomes as shown in Fig. 7.9.

```
Step 1: IF TOP = NULL
              PRINT "UNDERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

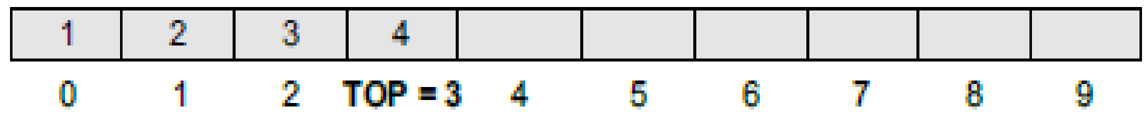


Figure 7.9 Stack after deletion

Figure 7.10 shows the algorithm to delete an element from a stack. In Step 1, we first check for the `UNDERFLOW` condition. In Step 2, the value of the location in the stack pointed by `TOP` is stored in `VAL`. In Step 3, `TOP` is decremented.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for Peek operation

7.3.3 Peek Operation

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack. The algorithm for Peek operation is given in Fig. 7.11.

However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned. Consider the stack given in Fig. 7.12.

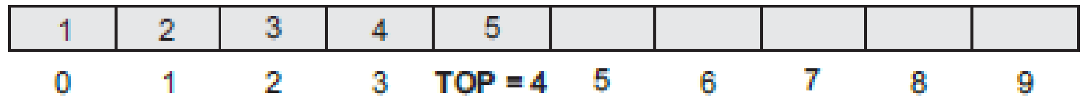


Figure 7.12 Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

7.4 LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

The linked representation of a stack is shown in Fig. 7.13.

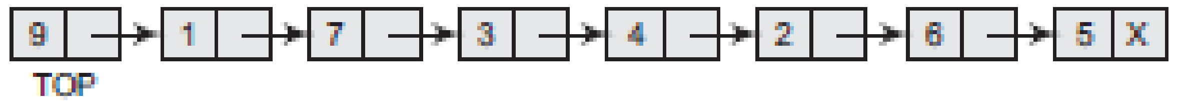


Figure 7.13 Linked stack

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
             SET NEW_NODE -> NEXT = NULL
             SET TOP = NEW_NODE
        ELSE
             SET NEW_NODE -> NEXT = TOP
             SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

7.5.1 Push Operation

The `push` operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

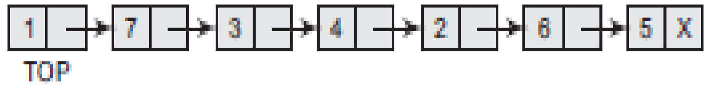


Figure 7.14 Linked stack

To insert an element with value 9, we first check if `TOP=NULL`. If this is the case, then we allocate memory for a new node, store the value in its `DATA` part and `NULL` in its `NEXT` part. The new node will then be called `TOP`. However, if `TOP!=NULL`, then we insert the new node at the beginning of the linked stack and name this new node as `TOP`. Thus, the updated stack becomes as shown in Fig. 7.15.

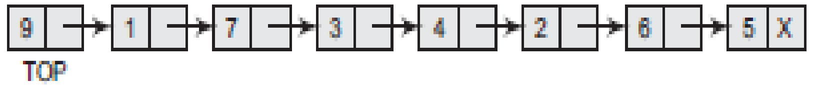


Figure 7.15 Linked stack after inserting a new node

7.5.2 Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already

mpty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.

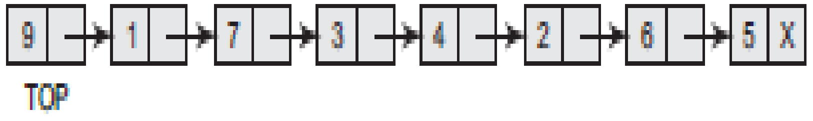


Figure 7.17 Linked stack

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second lement of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

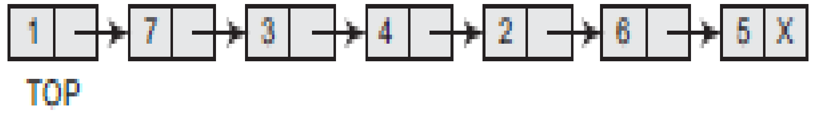


Figure 7.18 Linked stack after deletion of the topmost element

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

```
: IF TOP = NULL
      PRINT "UNDERFLOW"
      Goto Step 5
  [END OF IF]
```

```
/*Stack Implementation through Array*/
#include <stdio.h>
#include <conio.h>
int stack[5];
int top=-1;             //index pointing to the top of stack

void push()
{
    int y;

    if((top+1) == 5)
    {
                printf("\nStack Full\n");
                return;
    }
    else
    {
                printf("\n Enter Number");
                scanf("%d",&y);
                top++;
                stack[top]=y;
                printf("\n%d added to Stack at %d Position\n",y,top);
    }
}
```

```c
void pop()
{
    int a;

    if(top == -1)
    {
            printf("\nStack Empty\n");
    }
    else
    {
            a=stack[top];
            printf("\n %d deleted from stack from %d position",a,top);
            top--;

    }
}
```

```
void display()
{
    int i;

    if(top==-1)
    {
            printf("\nStack Empty\n");
    }
    else
      {
           printf("\n\n\nContents of the stack are : \n");
           for(i=top;i>=0;i--)
           {
                   printf("%d", stack[i]);
                   if(top == i)
                           printf("\t<--- TOP\n");
                   else
                           printf("\n");
           }
      }
}
```

```
void main()
{

    int i,num,menuselect;
    clrscr();
    printf("\n\t\tProgram for stack using array\n");
    do
    {
                printf("\n\n\tMain Menu: \n1.Add element to stack\n2.Delete element from the stack\n3.Display
Stack\n4.Exit");
                printf("\nSelect menu : ");
                scanf("%d",&menuselect);

                switch(menuselect)
                {
                 case 1 : push();
                            break;

                 case 2 : pop();
                            break;

                 case 3 : display();
                            break;

                 case 4 : printf("\nExiting the program");
                            break;

                 default: printf("Invalid menu item selected.");
                }
    } while(menuselect != 4);
}
```

7.6 MULTIPLE STACKS

While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent `OVERFLOW` conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated.

So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size. Figure 7.20 illustrates this concept.

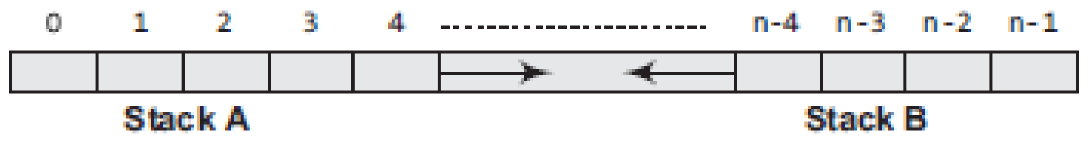


Figure 7.20 Multiple stacks

In Fig. 7.20, an array `STACK[n]` is used to represent two stacks, `Stack A` and `Stack B`. The value of `n` is such that the combined size of both the stacks will never exceed `n`. While operating on

these stacks, it is important to note one thing—`Stack A` will grow from left to right, whereas `Stack B` will grow from right to left at the same time.

Extending this concept to multiple stacks, a `stack` can also be used to represent `n` number of stacks in the same array. That is, if we have a `STACK[n]`, then each `stack I` will be allocated an equal amount of space bounded by indices `b[i]` and `e[i]`. This is shown in Fig. 7.21.

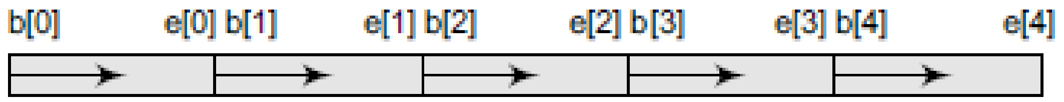
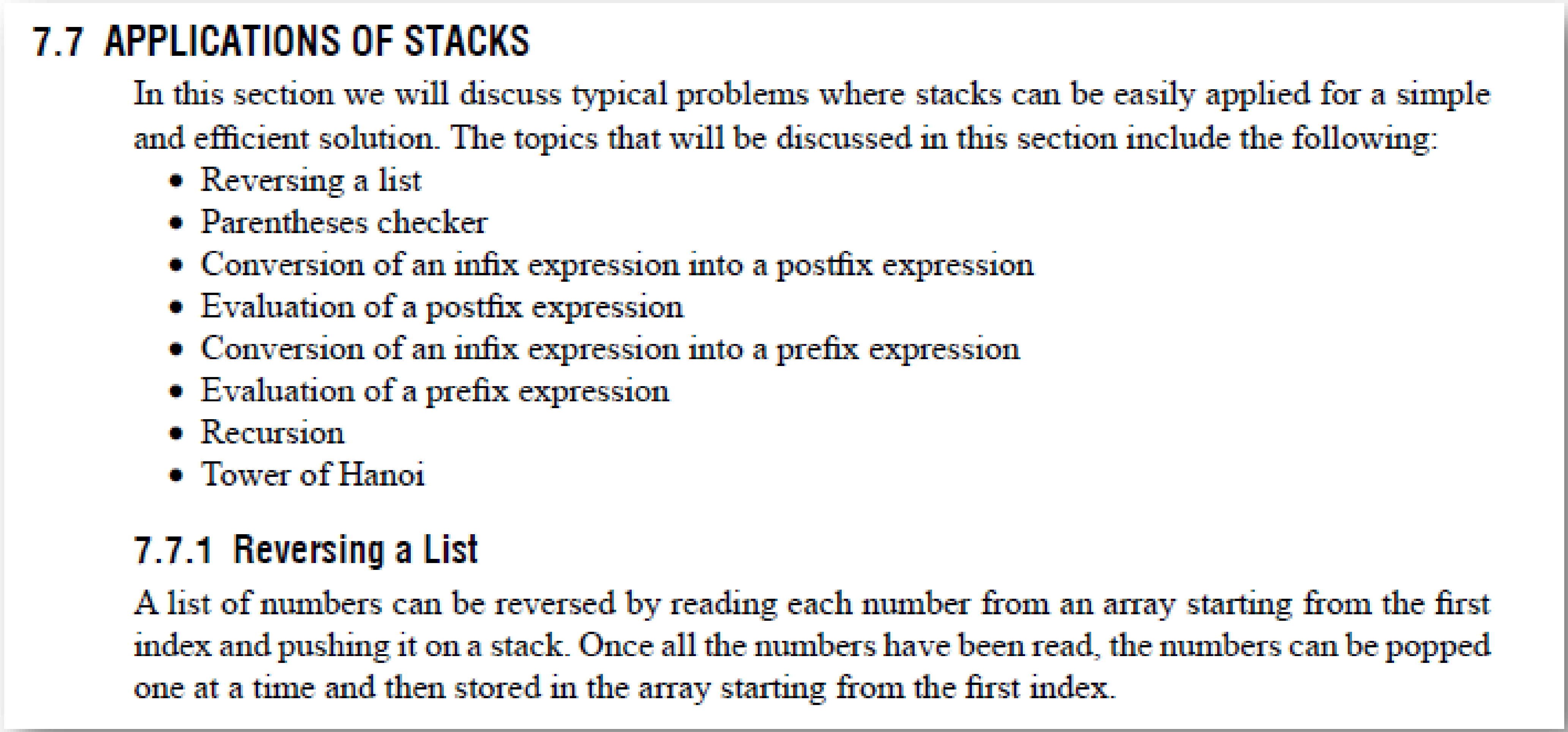


Figure 7.21 Multiple stacks



7.7 APPLICATIONS OF STACKS

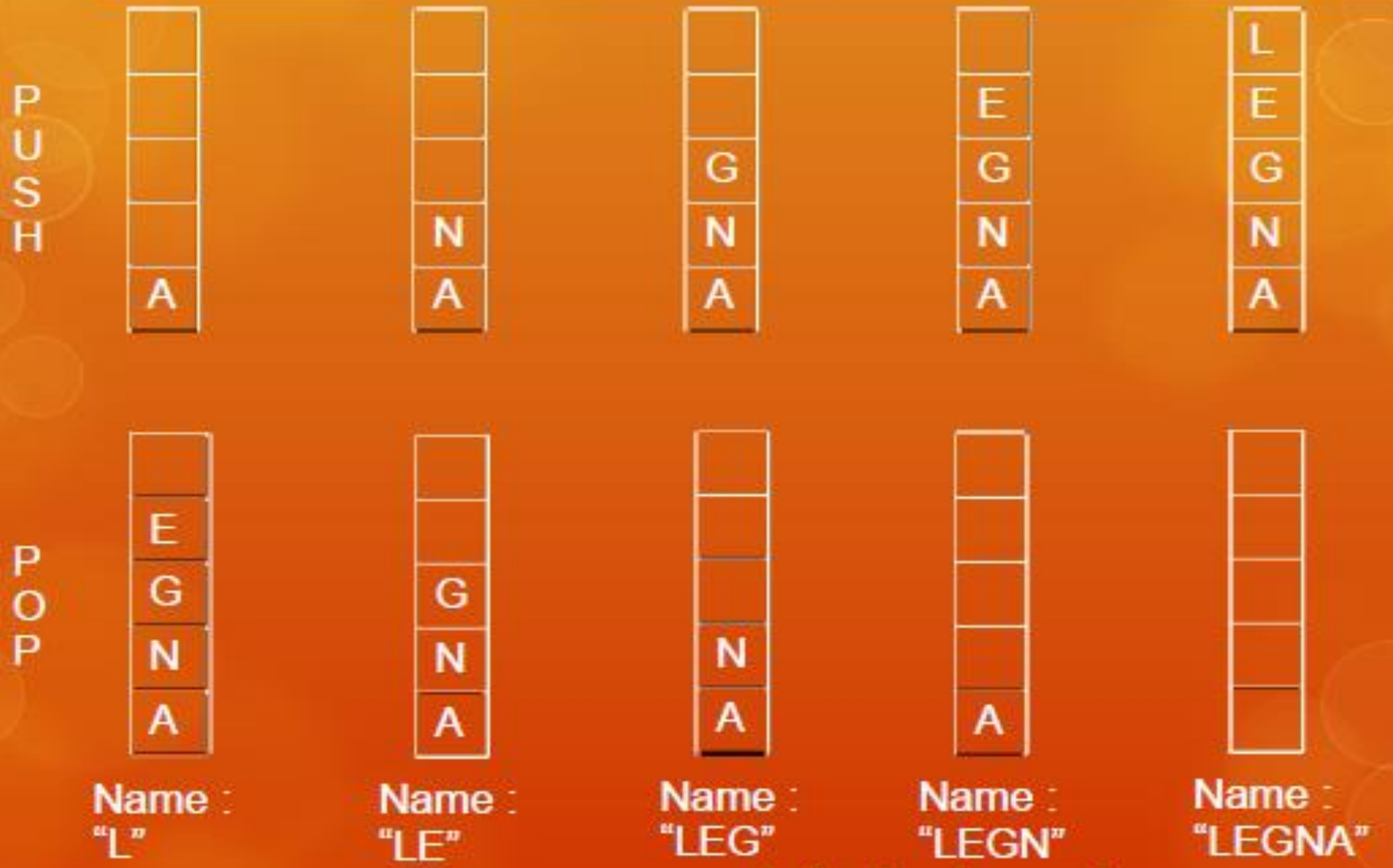
In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

7.7.1 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.

Reverse string – Name=“Angel”



7.7.2 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.

PARENTHESIS CHECKING

```
Procedure check()
    Declare a character stack S.
    Now traverse the expression.
        a) If the current character is a starting bracket then push it to
           stack.
        b) If the current character is a closing bracket then pop from
           stack and if the popped character is the matching starting bracket
           then fine else parenthesis are not balanced.
    After complete traversal, if there is some starting bracket left in stack
then "not balanced"
End procedure
```

Eg: **[a+(b*c)+{(d-e)}]**

[	Push [
[(	Push (
[	) and (matches, Pop (
[{	Push {
[{ (	Push (
[{	matches, pop (
[	Matches, pop {
	Matches, pop [

Thus, parenthesis match here

Polish Notations

Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as `A+B` in infix notation, the same expression can be written as `AB+` in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

The expression `(A + B) * C` can be written as:

`[AB+]*C`

`AB+C*` in the postfix notation

* Application of stack for Expression Evaluation & Expression conversion

→ Any Expression is made up of Operands (variables, operators, & delimiters.

Arithmetic operators ⇒ +, −, *, /, ** (Exponentiation), unary +, unary −

Relation operators ⇒ <, ≮, <=, =, ≠, >, ≯, >=

Boolean operators ⇒ && (Boolean AND),
|| (Boolean OR),
! (Boolean NOT)

unary +, unary −,
As well as some more operators like pre-increment
post-increment, ternary operator may be also ther

* A Big Question is that in what order the operations are to be carried out.

e.g. If we are given following expression & are to find it's final value.

$X \leftarrow A \mid B ** C + D * E - A * C$

If $A = 4$, $B = C = 2$, $D = E = 3$

we might want to evaluate above expression & want to assign final value to x.

some students may solve
t like below

4|(2**2)+(3*3)-(4*2)

4|4) + 9 - 8

$\boxed{2}$

*Whereas.
some other students may solve
it like below

(4|2)**(2+3)*(3-4)*2
=(4|2)** 5*(-1)*2
= 2**5 * -2.
= 32 * -2.
= $\boxed{-64}$

There may be many such problem while evaluating any expression

ence there must be some mechanism to tell us that what will be the unique Answer.

llowing are the rules to overcome the confusions.

tion- ① To fix the order of evaluation, each operator is assigned the priority.

Operator	Priority
**, unary −, unary +, ¬	6
*, /	5
+, −	4
$<$, $\nless$, $\leq$, $=$, $\neq$, $\ngeq$, $>$, $\ngtr$	3
and	2
or	1

Figure 3.7. Priority of arithmetic, Boolean and relational operators

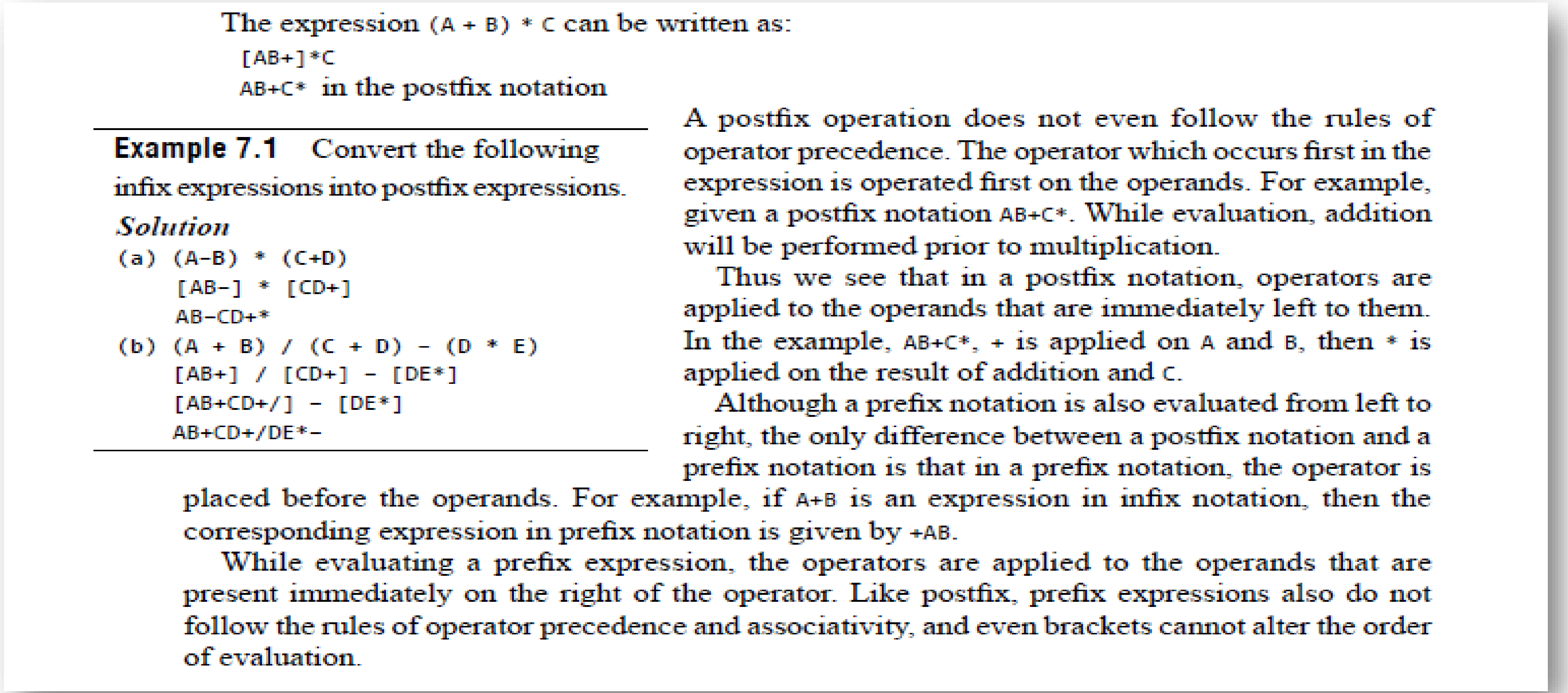
- After Assigning priorities also, there may be still some problmes
- Just like in below example:
- Ex-1) -A^B
- How to interpret this ?
- Case-1) (-A) ^ B
- Case-2) –(A^B)
- Assume A = -1 and B=2
- Then Answer of Case-1) will be➔ (- (-1)) ^ 2 ➔ 1
- And Answer of Case-2) will be ➔ - (-1^2) ➔ -1

- Conclusion: There are many problems in Infix expression like

1) One Infix expression may produce different results by different students
2) After assigning priorities also the problem may not be solved completely.
3) Associativity(Left to Right) or (Right to Left) may solve the problem.

Evaluation of Arithmetic Expressions

Polish Notations



The expression (A + B) * C can be written as:

```
[AB+]*C
AB+C*  in the postfix notation
```

Example 7.1 Convert the following infix expressions into postfix expressions.

Solution

```
(a) (A–B) * (C+D)
    [AB–] * [CD+]
    AB–CD+*
(b) (A + B) / (C + D) – (D * E)
    [AB+] / [CD+] – [DE*]
    [AB+CD+/] – [DE*]
    AB+CD+/DE*–
```

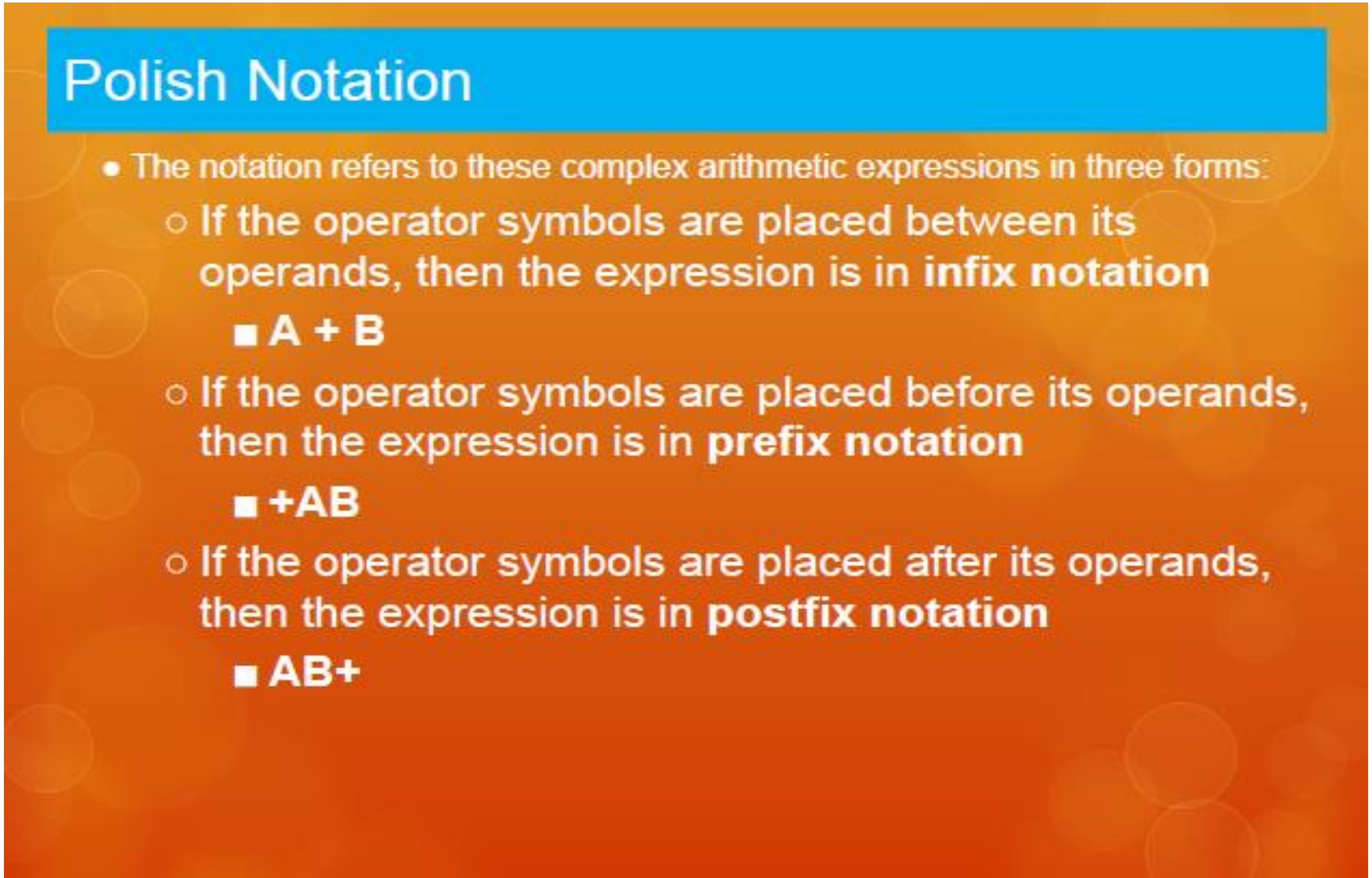
A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, AB+C*, + is applied on A and B, then * is applied on the result of addition and C.

Although a prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.

While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

- Expression Evaluation

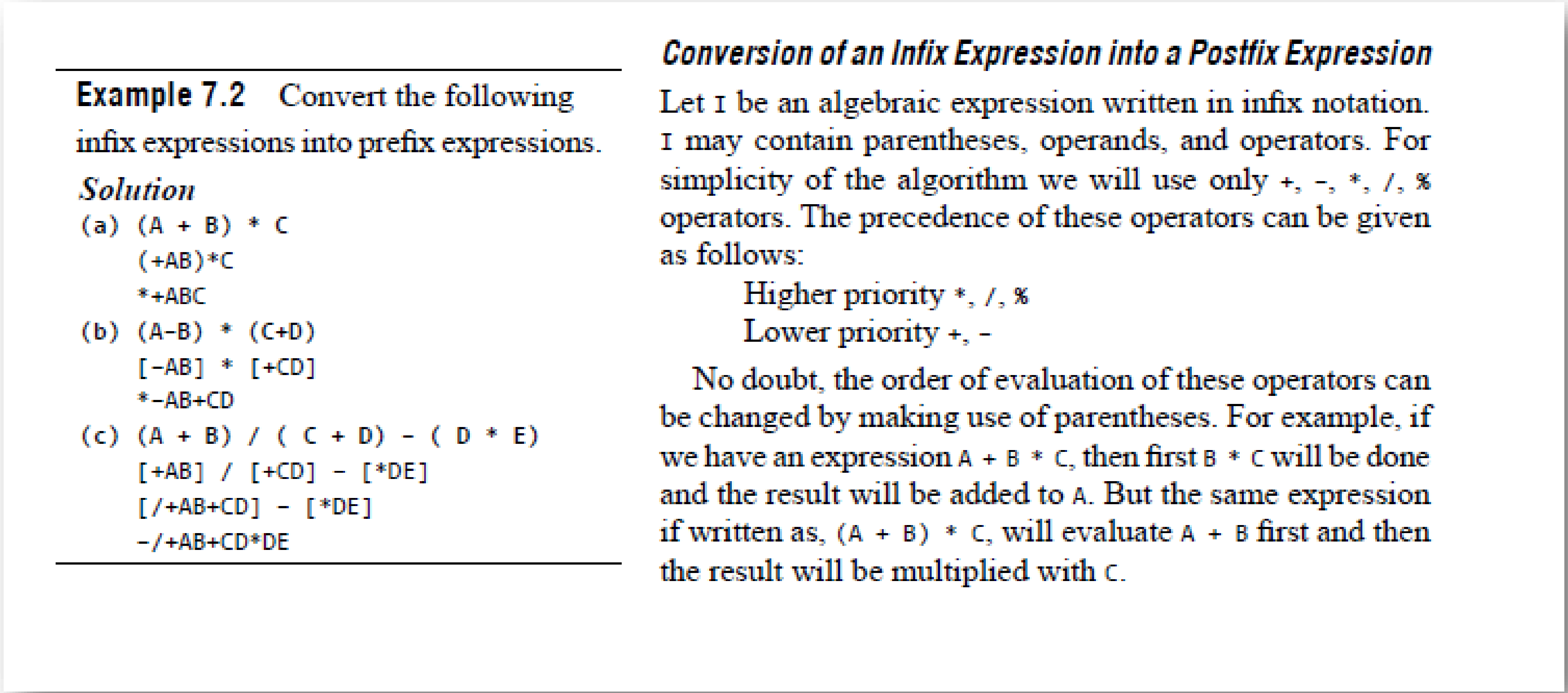


Polish Notation

- The notation refers to these complex arithmetic expressions in three forms:
  - If the operator symbols are placed between its operands, then the expression is in **infix notation**
    - A + B
  - If the operator symbols are placed before its operands, then the expression is in **prefix notation**
    - +AB
  - If the operator symbols are placed after its operands, then the expression is in **postfix notation**
    - AB+

Notation

Infix Notation	Prefix Notation	Postfix Notation
A + B	+ AB	AB +
(A - C) + B	+ - ACB	AC – B +
A + (B * C)	+ A * BC	ABC *+
(A+B)/(C-D)	/+ AB – CD	AB + CD -/
(A + (B * C))/(C – (D * B))	/+ A * BC – C * DB	ABC * + CDB * - /



Example 7.2 Convert the following infix expressions into prefix expressions.

Solution

```
(a) (A + B) * C
    (+AB)*C
    *+ABC
(b) (A-B) * (C+D)
    [-AB] * [+CD]
    *-AB+CD
(c) (A + B) / ( C + D) - ( D * E)
    [+AB] / [+CD] - [*DE]
    [/+AB+CD] - [*DE]
    -/+AB+CD*DE
```

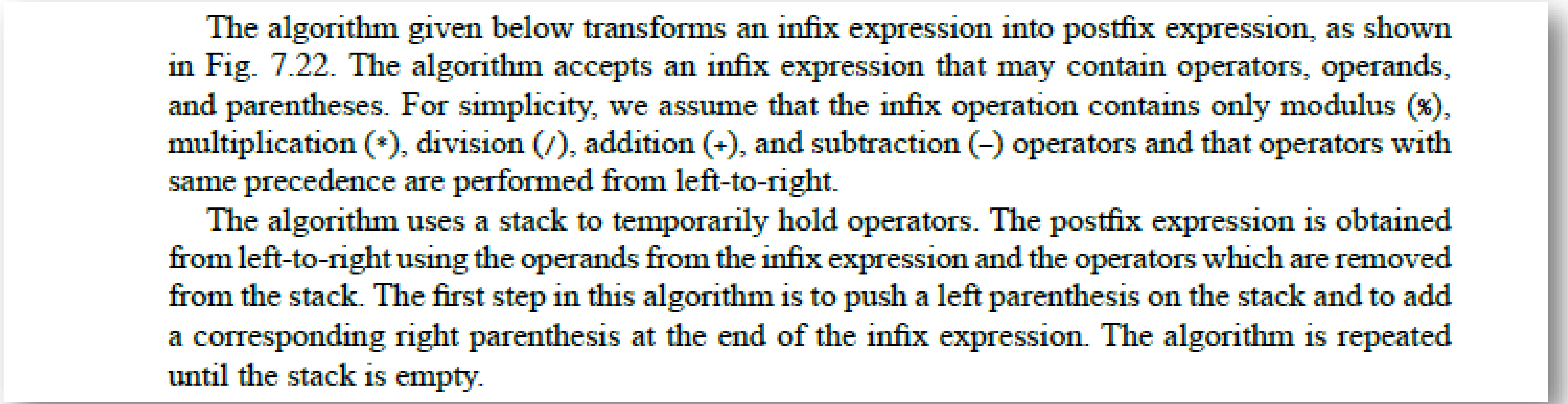
Conversion of an Infix Expression into a Postfix Expression

Let `I` be an algebraic expression written in infix notation. `I` may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only `+`, `-`, `*`, `/`, `%` operators. The precedence of these operators can be given as follows:

Higher priority `*`, `/`, `%`
Lower priority `+`, `-`

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression `A + B * C`, then first `B * C` will be done and the result will be added to `A`. But the same expression if written as, `(A + B) * C`, will evaluate `A + B` first and then the result will be multiplied with `C`.

Conversion of an Infix Expression into a Postfix Expression



The algorithm given below transforms an infix expression into postfix expression, as shown in Fig. 7.22. The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (–) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

```
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 notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Figure 7.22 Algorithm to convert an infix notation to postfix notation

Examples=> Convert following infix into postfix form.

Ex-1) X+Y*Z

Ex-2) (X+Y)*Z

Ex-3) X*Y/Z

Ex-4) X*(Y/Z)

Ex-5) A+B*C-D

Ex-6) (A+B) * (C-D)

Ex-7) A/B^C+D*E-A*C

Space For Example Solving

Answers=>

Ex-1) X+Y*Z	=> XYZ*+
Ex-2) (X+Y)*Z	=> XY+Z*
Ex-3) X*Y/Z	=> XY*Z/
Ex-4) X*(Y/Z)	=> XYZ/*
Ex-5) A+B*C-D	=> ABC*+D-
Ex-6) (A+B) * (C-D)	=> AB+CD-*
Ex-7) A/B^C+D*E-A*C	=> ABC^/DE*+AC*-

Solution

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 * –

Example 7.3 Convert the following infix expression into postfix expression using the algorithm given in Fig. 7.22.

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

A=A+B*C		B=ABC*+
Character	Stack	Output
A		A
+	+	A
B	+	AB
*	+,*	AB
C	+,*	ABC
Null		ABC*+

A=(A+B)*C		B=AB+C*
Character	Stack	Output
(	(	
A	(	A
+	(+	A
B	(+	AB
)	--	AB+
*	*	AB+
C	*	AB+C
Null	--	AB+C*

A=(A*B)+C		B=
Character	Stack	Output
(	(	
A	(	A
*	(*	A
B	(*	AB
)	--	AB*
+	+	AB*C+

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

Character	Stack	Output
A		A
+	+	A
(	+,(	A
B	+,(	AB
*	+,(,*	AB
C	+,(,*	ABC
-	+,(,-	ABC*
(	+,(,-,(	ABC*
D	+,(,-,(	ABC*D
/	+,(,-,(,/	ABC*D
E	+,(,-,(,/	ABC*DE
^	+,(,-,(,/, ^	ABC*DE
F	+,(,-,(,/, ^	ABC*DEF
)	+,(,-	ABC*DEF^/
*	+,(,-,*	ABC*DEF^/
G	+,(,-,*	ABC*DEF^/G

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

Character	Stack	Output

(A+B^D)/(E-F)+G

Character	Stack	Output
(	(	
A	(	A
+	(,+	A
B	(,+	AB
^	(,+,^	AB
D	(,+,^	ABD
)	--	ABD^+
/	/	ABD^+
(	/,(	ABD^+
E	/,(	ABD^+E
-	/,(,-	ABD^+E
F	/,(,-	ABD^+EF
)	/	ABD^+EF-
+	+	ABD^+EF-/
G	+	ABD^+EF-/G
Null	Null	ABD^+EF-/G+

(A+B^D)/(E-F)+G

Character	Stack	Output

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

Character	Stack	Output
A		A
-	-	A
B	-	AB
/	-,/	AB
(	-,/,(	AB
C	-,/,(	ABC
^	-,/,(, ^	ABC
D	-,/,(, ^	ABCD
)	-,/	ABCD^
+	+	ABCD^/-
(	+,(	ABCD^/-
E	+,(	ABCD^/-E
*	+,(,*	ABCD^/-E
F	+,(,*	ABCD^/-EF
)	+	ABCD^/-EF*
--	--	ABCD^/-EF*+

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

Character	Stack	Output

(300+23)*(43-21)/(84+7)

Character	Stack	Output
(	(	--
300	(	300
+	(,+	300
23	(,+	300 23
)	--	300 23 +
*	*	300 23 +
(	*,(	300 23 +
43	*,(	300 23 + 43
-	*,(,-	300 23 + 43
21	*,(,-	300 23 + 43 21
)	*	300 23 + 43 21 -
/	/	300 23 + 43 21 - *
(	/,(	300 23 + 43 21 - *
84	/,(	300 23 + 43 21 - * 84
+	/,(, +	300 23 + 43 21 - *

(300+23)*(43-21)/(84+7)

Character	Stack	Output

A*(B+C)/D

Character	Stack	Output
A		A
*	*	A
(	*,(	A
B	*,(	AB
+	*,(,+	AB
C	*,(,+	ABC
)	*	ABC+
/	/	ABC+*
D	/	ABC+*D
--	--	ABC+*D/

Evaluation of a Postfix Expression

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack. Let us look at Fig. 7.23 which shows the algorithm to evaluate a postfix expression.

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 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 the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Figure 7.23 Algorithm to evaluate a postfix expression

Table 7.1 Evaluation of a postfix expression

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

Let us now take an example that makes use of this algorithm. Consider the infix expression given as `9 - ((3 * 4) + 8) / 4`. Evaluate the expression.

The infix expression `9 - ((3 * 4) + 8) / 4` can be written as `9 3 4 * 8 + 4 / -` using postfix notation. Look at Table 7.1, which shows the procedure.

Examples=>Convert following postfix into infix form

Ex-1) XYZ*+

Ex-2) XY+Z*

Ex-3) XY*Z/

Ex-4) XYZ/*

Ex-5) ABC*+D-

Ex-6) AB+CD-*

Ex-7) ABC^/DE*+AC*-

- Space For Example Solving

Answers=>

Ex-1) XYZ*+	=> (X+(Y*Z))
Ex-2) XY+Z*	=> ((X+Y)*Z)
Ex-3) XY*Z/	=> ((X*Y)/Z)
Ex-4) XYZ/*	=> (X*(Y/Z))
Ex-5) ABC*+D-	=> ((A+(B*C))-D)
Ex-6) AB+CD-*	=> ((A+B) * (C-D))
Ex-7) ABC^/DE*+AC*-	=> (((A/(B^C))+(D*E))-(A*C))

12,7,3,-,/,2,1,5,+,*,+

Character	Stack	Output
12	12	
7	12,7	
3	12,7,3	
-	12,4	7-3=4
/	3	12 /4=3
2	3,2	
1	3,2,1	
5	3,2,1,5	
+	3,2,	1+ 5-6
*	3,2,6	
+	3,12	2*6=12
--	15	3+12=15
	15	

2,3,1,*,+,9,-

Character	Stack	Output
2	2	
3	2,3	
1	2,3,1	
2	2,3	3*1
+	5	2+3
9	5,9	5-9
-	-4	

10,2,8,*,+,3,-

Character	Stack	Output
10	10	
2	10,2	
8	10,2,8	
*	10,16	
+	26	
3	26,3	
-	23	

AB+C*D/

Character	Stack	Output
A	A	
B	A,B	
+	(A+B)	(A+B)
C	(A+B),C	
*	((A+B)*C)	
D	((A+B)*C),D	
/	(((A+B)*C)/D)	

5,6,2,+,*,12,4,/,-

Character	Stack	Output

PQR^/ST*+PR*-

Character	Stack	Output

PQR-S*+T-

Character	Stack	Output

P+(Q-R)*S-T

Character	Stack	Output

A/B*C+(D-E)

Character	Stack	Output

((P+Q)*s)^(T-U)

Character	Stack	Output

P/Q^R+S*T-P*R

Character	Stack	Output

((A+B)*D)^(E-F)

Character	Stack	Output

(((A/(B^C))+(D*E))-(A*C))

Character	Stack	Output

Convert Infix expression to Prefix expression

- A – Arithmetic Expression B – Prefix Expression

Step 1. Push ")" onto STACK, and add "(" to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered then:

a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has

b. Add operator to STACK

Step 6. If left parenthesis is encontered then

a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)

b. Remove the left parenthesis

Step 7. Exit

a-b/(c^d)+(e*f)

Character	Stack	Output
)	)	
f	)	f
*	),*	f
e	),*	fe
(	---	fe*
+	+	fe*
)	+,)	fe*
d	+,)	fe*d
^	+,), ^	fe*d
c	+,), ^	fe*dc
(	+	fe*dc^
/	+,/	fe*dc^
b	+,/	fe*dc^b
-	+,-	fe*dc^b/
a	+,-	fe*dc^b/a
--	--	fe*dc^b/a-+

A+B*C

Character	Stack	Output

(A+B)*C

Character	Stack	Output

A/B^C+D*E-F*G

Character	Stack	Output
G		G
*	*	G
F	*	GF
-	-	GF*
E	-	GF*E
*	-,*	GF*E
D	-,*	GF*ED
+	-,+	GF*ED*
C	-,+	GF*ED*C
^	-,+,^	GF*ED*C
B	-,+,^	GF*ED*CB
/	-,+,/	GF*ED*CB^
A	-,+,/	GF*ED*CB^A
--	--	GF*ED*CB^A/+-
	-+/A^BC*DE*FG	

P+q-r*t

Character	Stack	Output
t		T
*	*	t
r	*	tr
-	-	tr*
q	-	Tr*q
+	-,+	Tr*q
p	-,+	tr*qp
		Tr*qp+-
	-+pq*rt	

A-B/(C*D^E)

Character	Stack	Output

Evaluation of a Prefix Expression

There are a number of techniques for evaluating a prefix expression. The simplest way of evaluation of a prefix expression is given in Fig. 7.26.

```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
          (a) Scan the prefix expression
              from right, one character at a
              time.
          (b) If the scanned character is an
              operand, push it on the
              operand stack.
          (c) 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
Step 3: END
```

Figure 7.26 Algorithm for evaluation of a prefix expression

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

*+ab^cd

Character	Stack	Output
d	d	
c	d,c	
^	(c ^ d)	(c ^ d)
b	(c ^ d), b	
a	(c ^ d), b,a	
+	(c ^ d), (a + b)	(a + b)
*	((a + b) *(c ^ d))	(a + b) *(c ^ d)
	((a + b) *(c ^ d))	

+-a/b^cd*ef

Character	Stack	Output

-A/B*C^DE

Character	Stack	Output

-+/A^BC*DE*FG

Character	Stack	Output

7.7.4 Recursion

In this section we are going to discuss recursion which is an implicit application of the STACK ADT.

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 × 4 × 3 × 2 × 1
   = 120
```

This can be written as

```
5! = 5 × 4!, where 4!= 4 × 3!
```

Therefore,

```
5! = 5 × 4 × 3!
```

Similarly, we can also write,

```
5! = 5 × 4 × 3 × 2!
```

Expanding further

```
5! = 5 × 4 × 3 × 2 × 1!
```

We know, `1! = 1`

The series of problems and solutions can be given as shown in Fig. 7.27.

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the

PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Figure 7.27 Recursive factorial function

factorial of a number. 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)
```

Look at the following program which calculates the factorial of a number recursively.

From the above example, let us analyse 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.

RECURSION HANDLING

- Without stack, recursion is difficult
- Compiler automatically uses stack data structure while handling recursion.
- All computer needs to remember for each active function call, values of arguments & local variables and the location of the next statement to be executed when control goes back.
- Essentially what is happening when we call that method is that our current execution point is pushed onto the call stack and the runtime starts executing the code for the internal method call. When that method finally returns, we pop our place from the stack and continue executing.

10. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int);    // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
        return 0;
}
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```

The Fibonacci Series

The Fibonacci series can be given as

```
0 1 1 2 3 5 8 13 21 34 55 ......
```

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the `nth` term of the Fibonacci series. The general formula to do so can be given as

As per the formula, `FIB(0) =0` and `FIB(1) = 1`. So we have two base cases. This is necessary because every problem is divided into two smaller problems.

$$\text{FIB}(n) = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ \text{FIB}(n - 1) + \text{FIB}(n - 2), \text{ otherwise} \end{cases}$$

13. Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>
int Fibonacci(int);
int main()
{
        int n, i = 0, res;
        printf("Enter the number of terms\n");
        scanf("%d",&n);
        printf("Fibonacci series\n");
        for(i = 0; i < n; i++ )
        {
                res = Fibonacci(i);
```

```
                        printf("%d\t",res);
                }
                return 0;
        }
        int Fibonacci(int n)
        {
                if ( n == 0 )
                        return 0;
                else if ( n == 1 )
                        return 1;
                else
                        return ( Fibonacci(n-1) + Fibonacci(n-2) );
        }
```

Output

```
        Enter the number of terms
        Fibonacci series
           0     1         1         2         3
```

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*),
- whether any operation is pending at each recursive call (*tail-recursive* or not), and
- the structure of the calling pattern (*linear or tree-recursive*).

In this section, we will read about all these types of recursions.

Direct Recursion

A function is said to be *directly* recursiveif it explicitly calls itself. For example, consider the code shown in Fig. 7.28. 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));
}
```

Figure 7.28 Direct recursion

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 (Fig. 7.29).

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

Figure 7.29 Indirect recursion

Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. When the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

In Fig. 7.30, the factorial function that we have written is a non-tail-recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call.

```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Figure 7.30 Non-tail recursion

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

Figure 7.31 Tail recursion

Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number of calls.

However, the same factorial function can be written in a tail-recursive manner as shown Fig. 7.31.

In the code, `Fact1` function preserves the syntax of `Fact(n)`. Here the recursion occurs in the `Fact1` function and not in `Fact` function. Carefully observe that `Fact1` has no pending operation to be performed on return from recursive calls. The value computed by the recursive call is simply returned without any modification. So in this case, the amount of information to be stored on the system stack is constant (only the values of `n` and `res` need to be stored) and is independent of the number of recursive calls.

Converting Recursive Functions to Tail Recursive

A non-tail recursive function can be converted into a tail-recursive function by using an *auxiliary parameter* as we did in case of the Factorial function. The auxiliary parameter is used to form the result. When we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation. We generally use an auxiliary function while using the auxiliary parameter. This is done to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

Linear and Tree Recursion

```
int Fibonacci(int num)
{
  if(num == 0)
    return 0;
  else if (num == 1)
    return 1;
    else
    return (Fibonacci(num - 1) + Fibonacci(num - 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function

Fibonacci(7) = Fibonacci(6) + Fibonacci(5)
Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

Fibonacci(3) = 1 + 1 = 2
Fibonacci(4) = 2 + 1 = 3
Fibonacci(5) = 3 + 2 = 5
Fibonacci(6) = 3 + 5 = 8
Fibonacci(7) = 5 + 8 = 13

Figure 7.32 Tree recursion

Recursive functions can also be characterized depending on the way in which the recursion grows in a linear fashion or forming a tree structure (Fig. 7.32).

In simple words, a recursive function is said to be *linearly* recursive when the pending operation (if any) does not make another recursive call to the function. For example, observe the last line of recursive factorial function. The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to `Fact`.

On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the `Fibonacci` function in which the pending operations recursively call the `Fib onacci` function.

Recursion versus Iteration

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step by step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems, a recursive code can be written and modified in a much simpler and clearer manner.

Difference between Recursion and Iteration

- ITERATION
  - It is a process of executing statements repeatedly, until some specific condition is specified
  - Iteration involves four clear cut steps, initialization, condition, execution and updating
  - Any recursive problem can be solved iteratively
  - Iterative computer part of a problem is more efficient in terms of memory utilization and execution speed

- RECURSIVE
  - Recursion is a technique of defining anything in terms of itself
  - There must be an exclusive if statement inside the recursive function specifying stopping condition
  - Not all problems has recursive solution
  - Recursion is generally a worst option to go for simple program or problems not recursive in nature

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'.

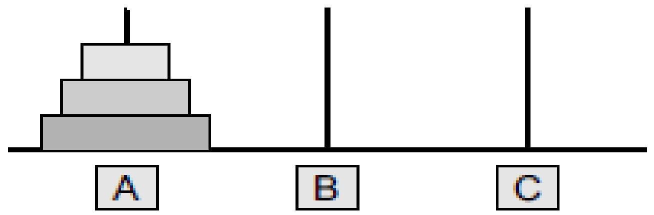


Figure 7.33 Tower of Hanoi

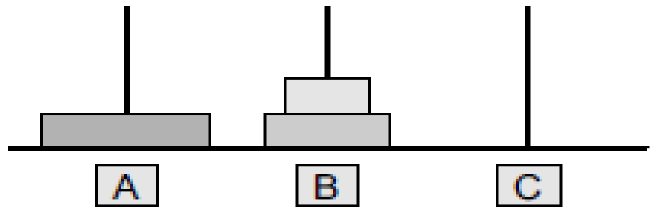


Figure 7.34 Move rings from A to B

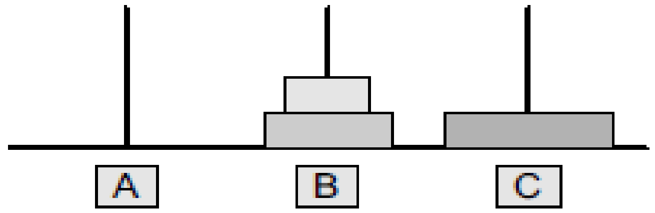


Figure 7.35 Move ring from A to C

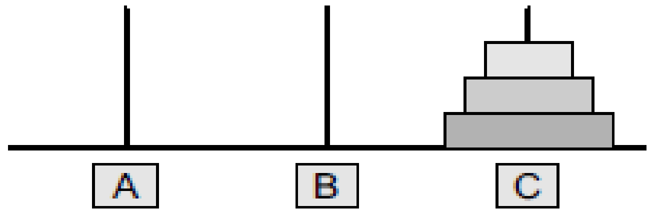


Figure 7.36 Move ring from B to C

Look at Fig. 7.33 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (`n-1 rings`) from the source pole to the spare pole. We move the first two rings from pole A to B as shown in Fig. 7.34.

Now that `n-1` rings have been removed from pole A, the `nth` ring can be easily moved from the source pole (A) to the destination pole (C). Figure 7.35 shows this step.

The final step is to move the `n-1` rings from the spare pole (B) to the destination pole (C). This is shown in Fig. 7.36.

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: `if n=1`

- Move the ring from A to C using B as spare

Recursive case:

- Move `n` – `1` rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move `n` – `1` rings from B to C using A as spare

The following code implements the solution of the Tower of Hanoi problem.

```
#include <stdio.h>
```

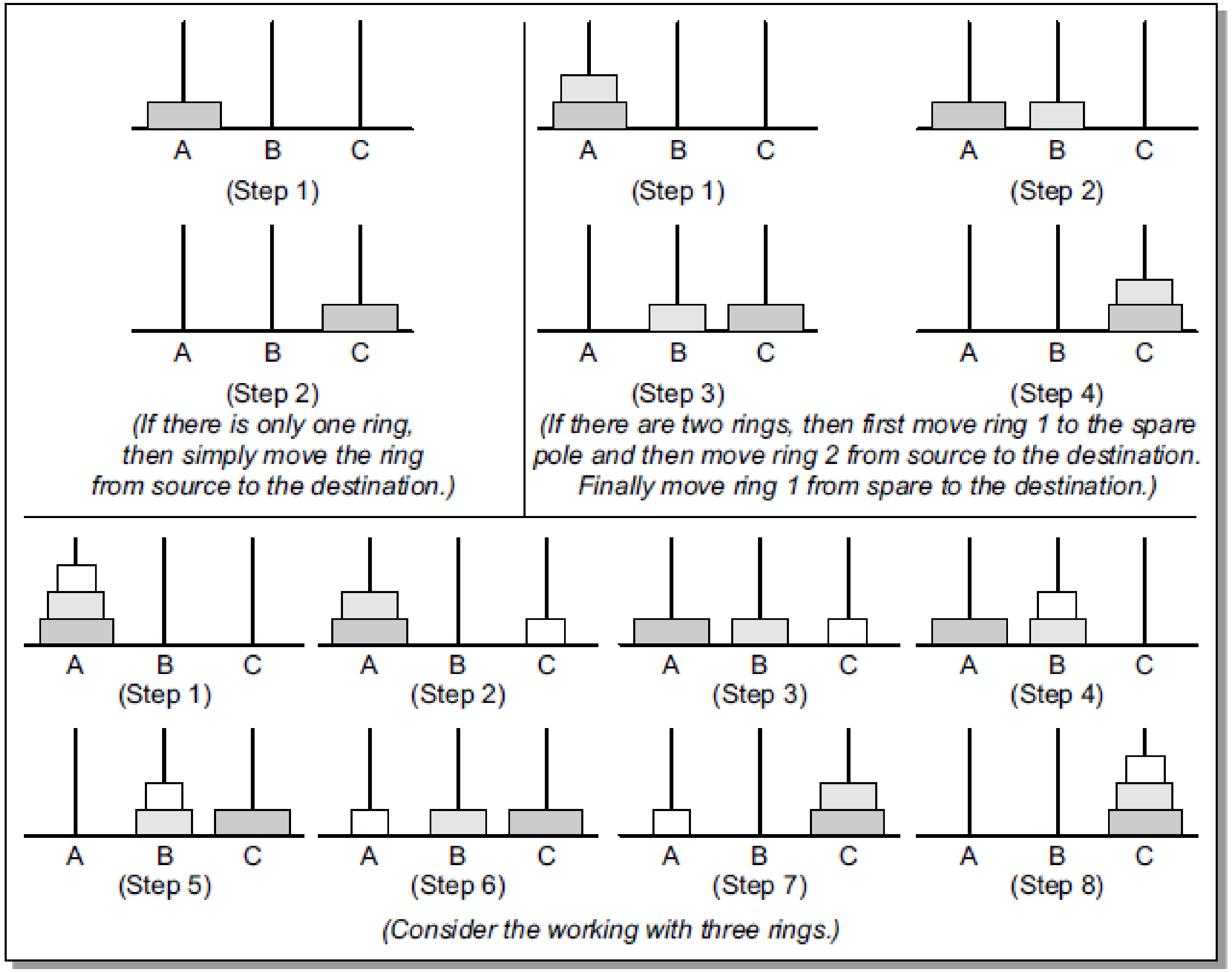

A
B
C
(Step 1)
(Step 2)
(If there is only one ring, then simply move the ring from source to the destination.)
(Step 3)
(Step 4)
(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)
(Step 5)
(Step 6)
(Step 7)
(Step 8)
(Consider the working with three rings.)

Figure 7.37 Working of Tower of Hanoi with one, two, and three rings

Use of stack in function call

- **| proc** MAIN **| |** proc **A1 | |** proc **A2 | |** proc **A3 |**
- **| ___ | | ___ | | ___ | | ___ |**
- **| ___ | | ___ | | ___ | | ___ |**
- **| ___ | | ___ | | ___ | | ___ |**
- **| call** A1 **| |** call **A2 | |** call **A3 | | ___ |**
- **|** r: **| |** s: **| |** t: **| | |**
- **| ___ | | ___ | | ___ | | ___ |**
- **| ___ | | ___ | | ___ | | ___ |**
- **| end | | end | | end | | end |**