BASUDEV GODABARI DEGREE COLLEGE , KESAIBAHAL

Department of Computer Science

"SELF STUDY MODULE"

Module Details :

- Class - 2nd Semester (2020-21) Admission Batch
- Subject Name : COMPUTER SCIENCE
- Paper Name : DataStructure

UNIT – 2 : STRUCTURE

Stack: Definition, Representation
Stack operations,
Applications (Infix–Prefix–Postfix Conversion & Evaluation, Recursion).
Queues: Definition,
Representation, Types of queue, Queue operations,
Applications.

Learning Objective

After Learning this unit you should be able to

- Know the Concept Stack
- How to use (Infix–Prefix–Postfix Conversion & Evaluation, Recursion).

Queues: Representation, Types of queue, Queue operations,
Applications.

You Can use the Following Learning Video link related to above topic :

https://www.youtube.com/watch?v=whYm5cqiqL8
https://www.youtube.com/watch?v=aCVpMz7_qn8
https://www.youtube.com/watch?v=zp6pBNbUB2U
https://www.youtube.com/watch?v=RY4GkLahbC

You Can also use the following Books :

Text book

1Classic Data Structure , D. Samanta , PHI , 2/ed.

REFERENCES

1. Ellis Horowitz, Sartaj Sahni, "Fundamentals of Data Structures",
1. Galgotia Publications, 2000. 2In
2. **And also you can download any book in free by using the following website.**

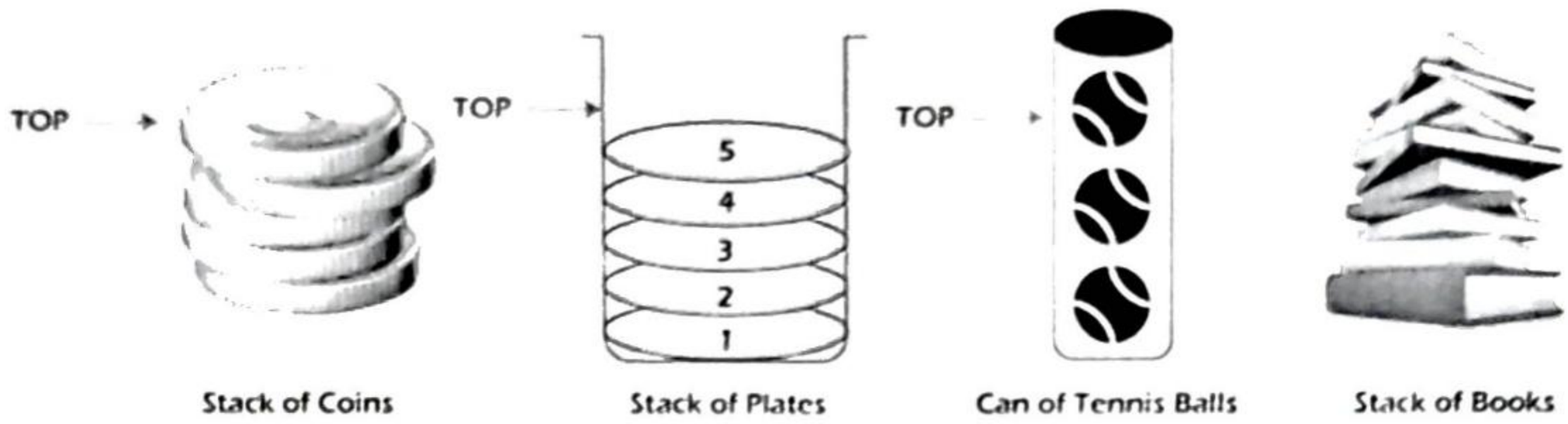
- https://www.pdfdrive.com/

Data Structure

What do you mean by Stack?

A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.

An everyday analogy of a stack data structure is a stack of books on a desk, Stack of plates, table tennis, Stack of bootless, Undo or Redo mechanism in the Text Editors, etc.



Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

1. Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: A + (B - C)

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new notation.

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

Example: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example: AB +, CD+, etc.

All these expressions are in postfix notation because the operator comes after the operands.

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation

A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

	Infix Expression	Stack	Postfix Expression
i)	A + B / C + D * (E - F) ^ G)	(	A
ii)	A + B / C + D * (E - F) ^ G)	(	A
iii)	A + B / C + D * (E - F) ^ G)	(	AB
iv)	A + B / C + D * (E - F) ^ G)	(+	ABC
v)	A + B / C + D * (E - F) ^ G)	/ + (	ABC/+
vi)	A + B / C + D * (E - F) ^ G)	+ (	ABC/+D
vii)	A + B / C + D * (E - F) ^ G)	+ (	ABC/+D
viii)	A + B / C + D * (E - F) ^ G)	* + (	ABC/+D
ix)	A + B / C + D * (E - F) ^ G)	* + (	ABC/+D
x)	A + B / C + D * (E - F) ^ G)	- * + (	ABC/+DE
xi)	A + B / C + D * (E - F) ^ G)	- * + (	ABC/+DE
xii)	A + B / C + D * (E - F) ^ G)	- * + (	ABC/+DEF
xiii)	A + B / C + D * (E - F) ^ G)	* + (	ABC/+DEF-
xiv)	A + B / C + D * (E - F) ^ G)	^ * + (	ABC/+DEF-
xv)	A + B / C + D * (E - F) ^ G)	^ * + (	ABC/+DEF-G
xvi)	A + B / C + D * (E - F) ^ G)		ABC/+DEF-G^*+

In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression:

Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.

Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

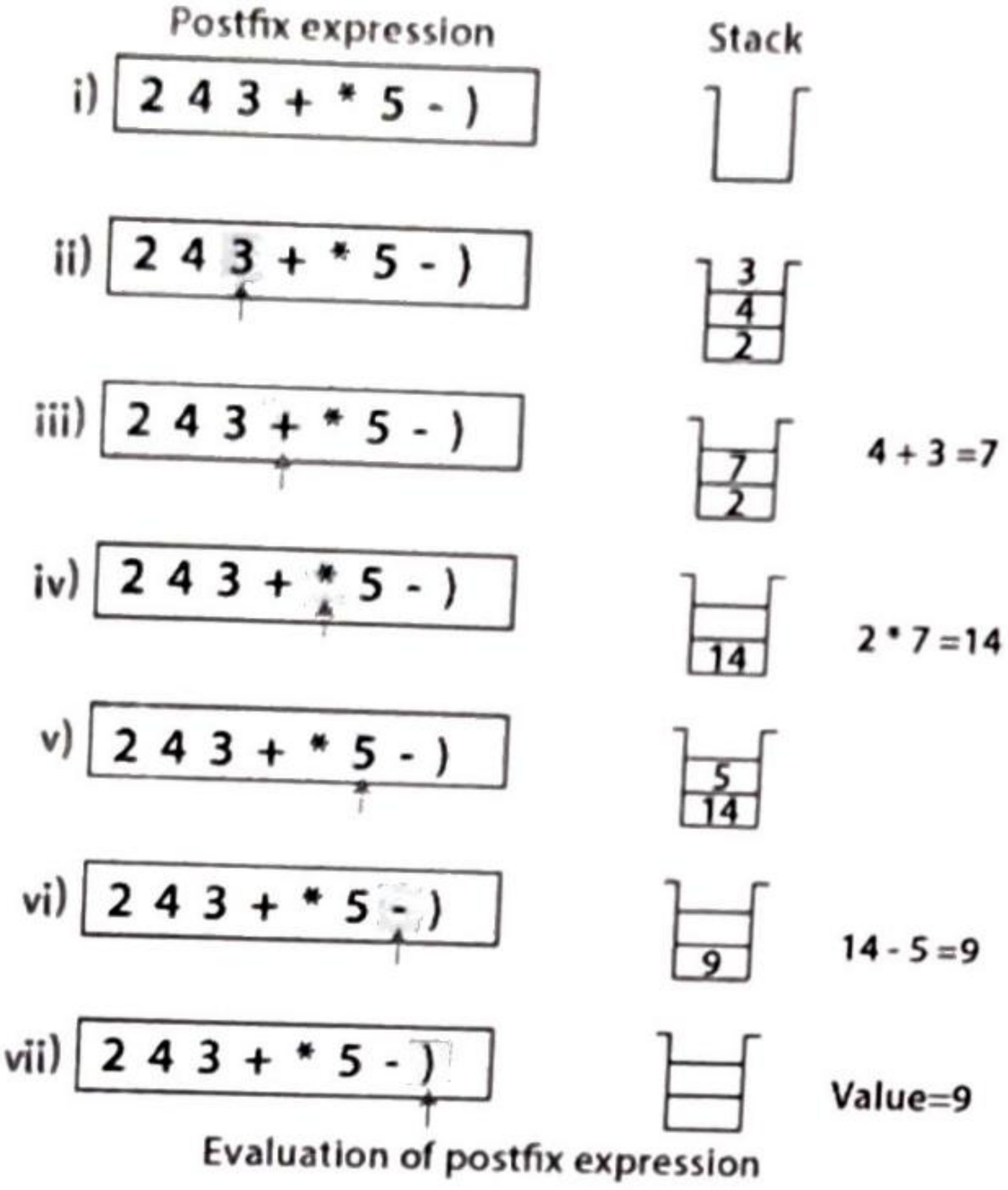
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example:

Now let us consider the following infix expression 2 * (4+3) - 5.

Its equivalent postfix expression is 2 4 3 + * 5.

The following step illustrates how this postfix expression is evaluated.



Evaluation of postfix expression

2. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

Valid Delimiter **Invalid Delimiter**

While (i > 0)

While (i >

/* Data Structure */

/* Data Structure

{ (a + b) - c }

{ (a + b) - c

To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

On matching, the following cases may arise.

- If the delimiters are of the same type, then the match is considered successful, and the process continues.
- If the delimiters are not of the same type, then the syntax error is reported.

When the end of the program is reached, and the Stack is empty, then the processing of the source program stops.

Example: To explain this concept, let's consider the following expression.

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

Input left	Characters Read	Stack Contents
[{{(a-b) * (c-d)}/f]	[	[
{{(a-b) * (c-d)}/f]	{	[{
{(a-b) * (c-d)}/f]	(	[{ (
(a-b) * (c-d)}/f]	a	[{ (
a-b) * (c-d)}/f]	-	[{ (
-b) * (c-d)}/f]	b	[{ (
b) * (c-d)}/f]	)	[{
) * (c-d)}/f]	*	[{
* (c-d)}/f]	(	[{ (
(c-d)}/f]	g	[{ (
c-d)}/f]	-	[{ (
-d)}/f]	d	[{ (
d)}/f]	)	[{
)}/f]	)	[
}/f]	/	[
/f]	f	[
f]		
]		

4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

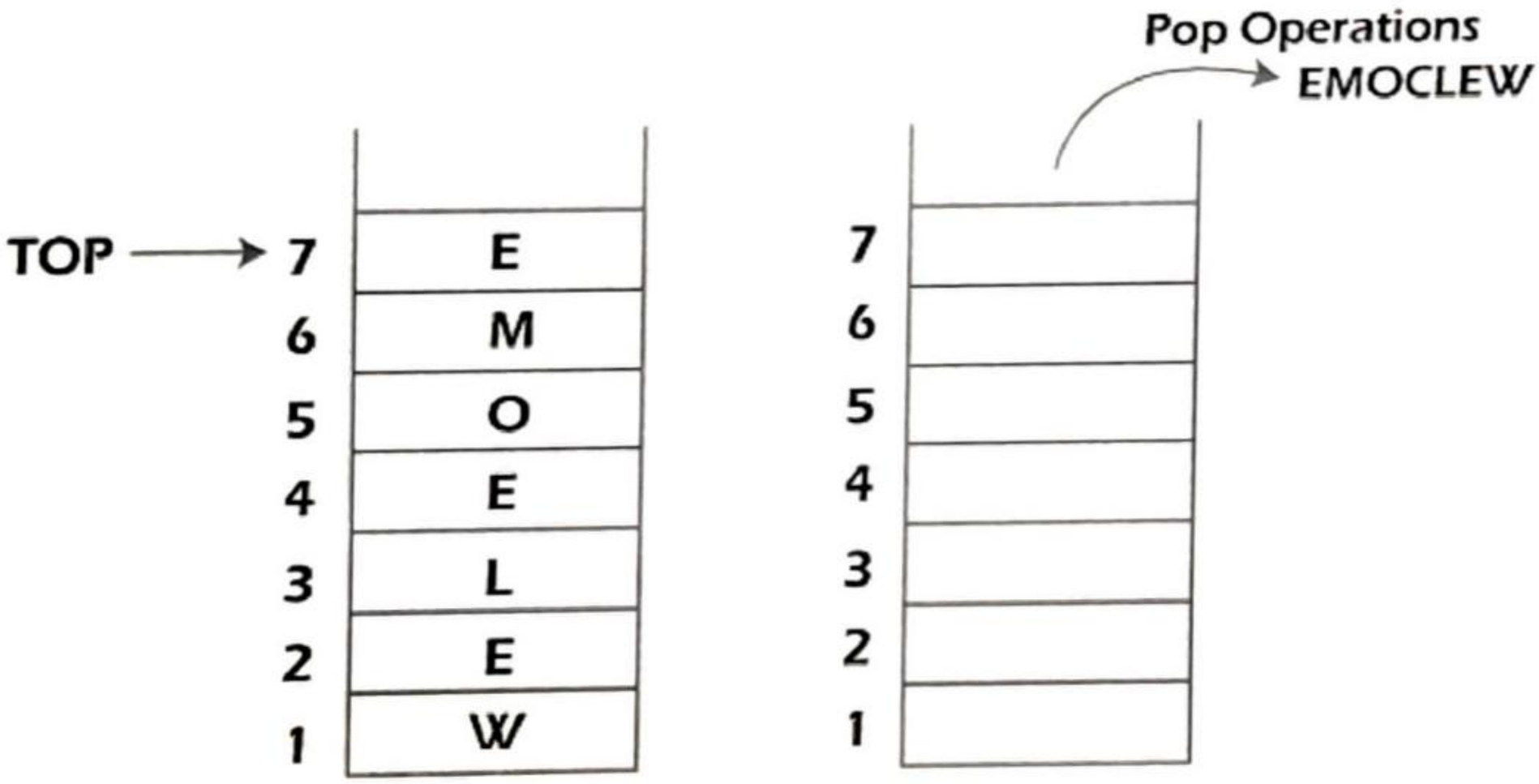
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

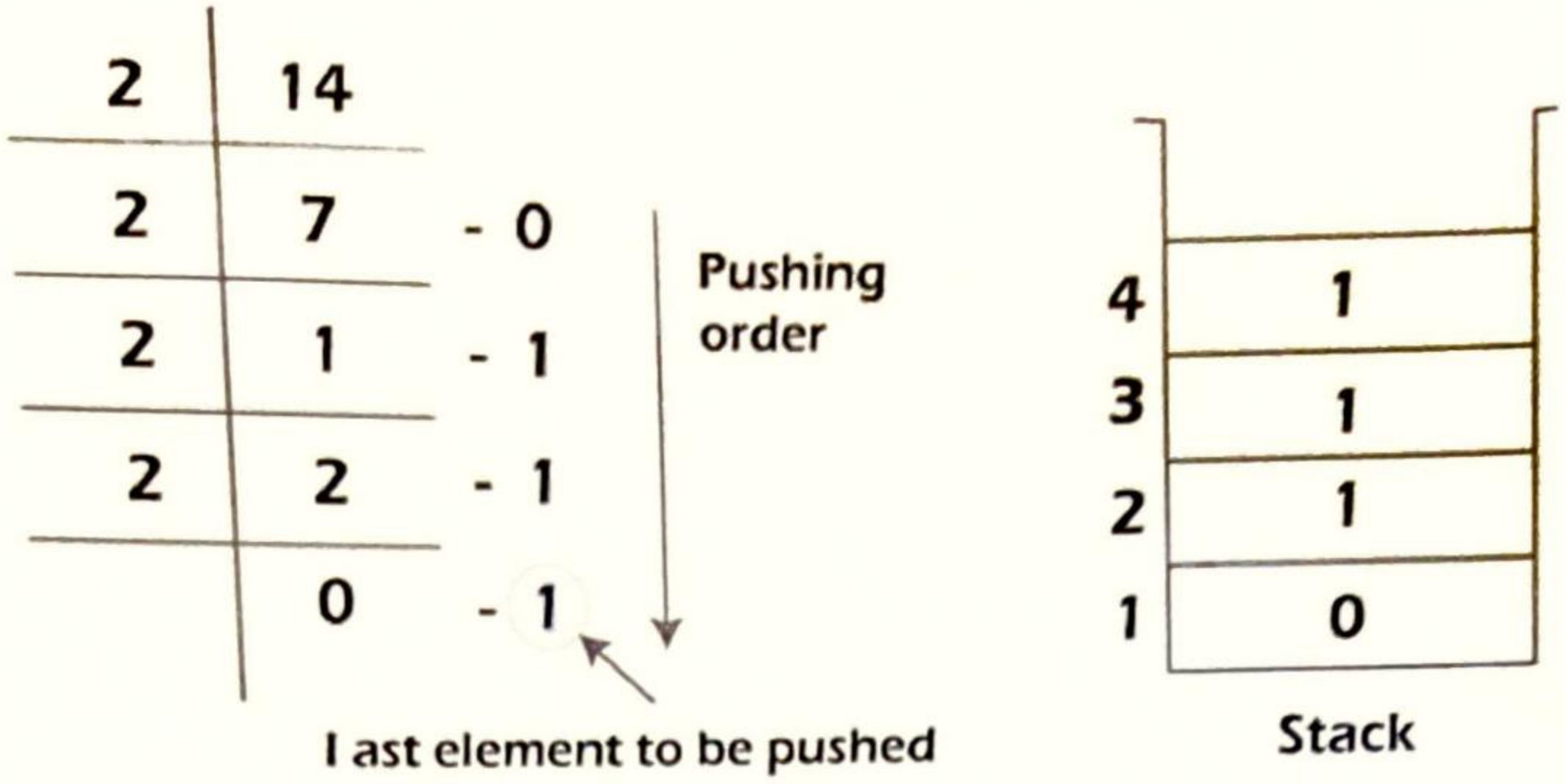
A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Converting Decimal to Binary:

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form. For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

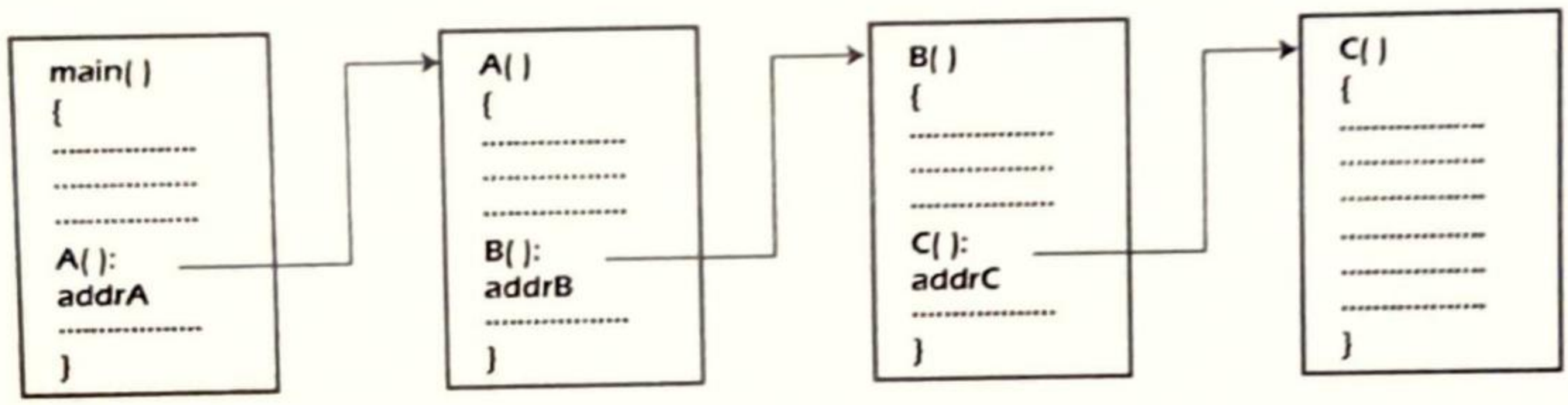
Example: Converting 14 number Decimal to Binary:



In the above example, on dividing 14 by 2, we get seven as a quotient and one as the reminder, which is pushed on the Stack. On again dividing seven by 2, we get three as quotient and 1 as the reminder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number **1110.**

5. Processing Function Calls:

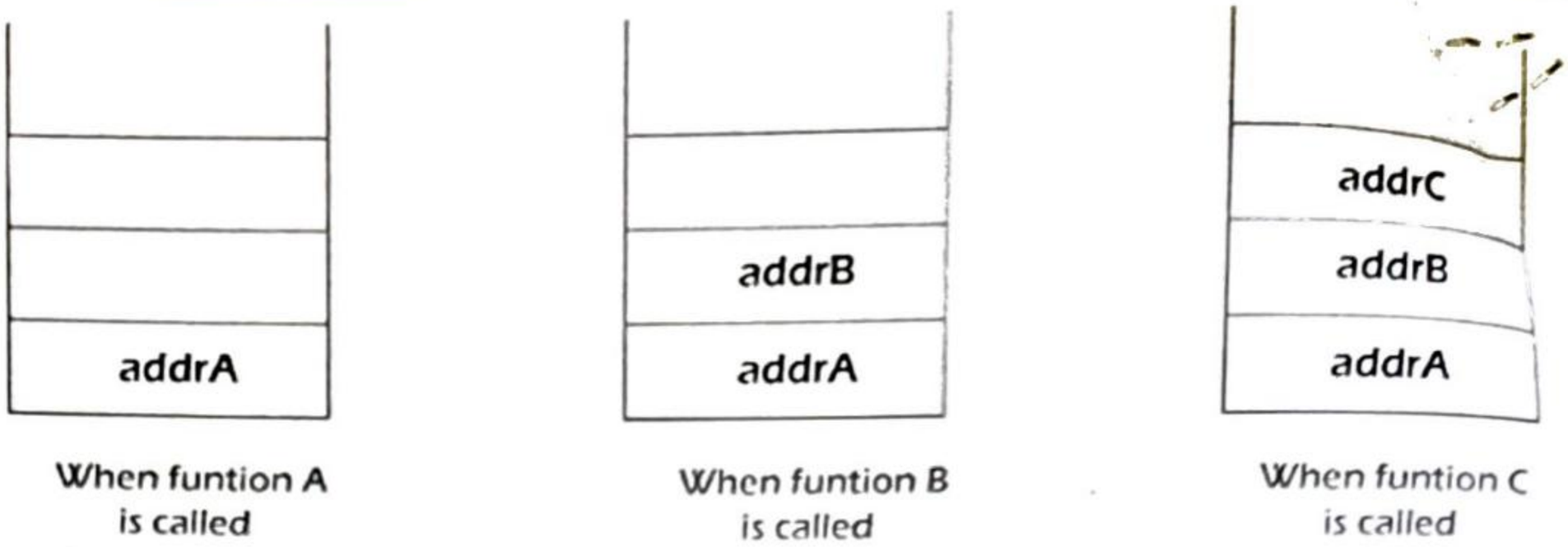
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



When funtion A is called

When funtion B is called

When funtion C is called

Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.