Noida Institute of Engineering and Technology, Greater Noida

Stacks And Queues

Unit:3

Basic of Data Structure

Course Details
(B Tech 3rd Sem)

Sanchi Kaushik
Assistant Professor
AIML

Syllabus

Stacks: Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack, Application of stack: Infix, Prefix, Postfix Expressions and their mutual conversion, Evaluation of postfix expression.

Recursion: Principles of recursion, Tail recursion, Removal of recursion, Problem solving using iteration and recursion with examples such as binary search, Fibonacci series, and Tower of Hanoi, Trade-offs between iteration and recursion.

Queues: Array and linked implementation of queues, Operations on Queue: Create, Insert, Delete, Full and Empty, Circular queues, Dequeue and Priority Queue.

Objective of Unit

Objective of the course is to make students able to:

1. Learn the basic types for data structure, implementation and application.
2. Know the strength and weakness of different data structures.

3. Use the appropriate data structure in context of solution of given problem.

4. Develop programming skills which require to solve given problem.

Course Outcome

CO2:

Describe how stacks, queues, are represented in memory, and identify the alternative implementations of data structures with respect to its performance to solve a real world problem.

Prerequisite and Recap

- Stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.
- Queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle.

Stack and Queue(CO2)

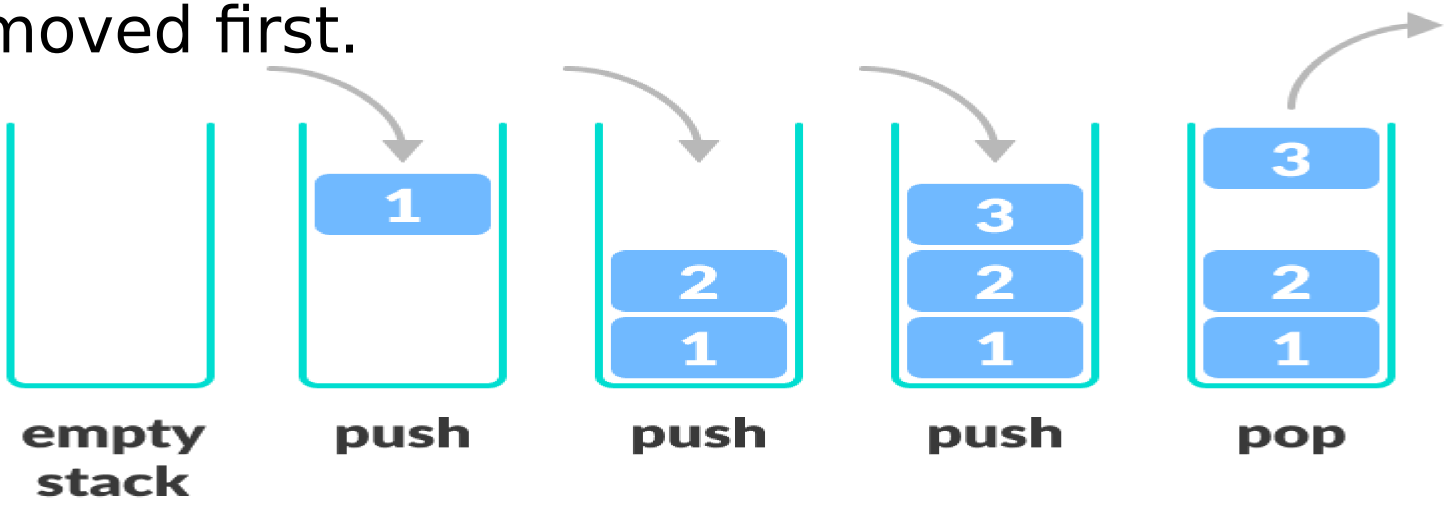
Topic mapping with Course Outcome

Topic	CO1	CO2	CO3	CO4	CO5
Stack	-	2	-	-	-
Queue	-	2	-	-	-

Introduction to Stack

Stack

A stack is a **linear data structure** that follows a particular order for the insertion and manipulation of data. Stack uses Last-In-First-Out (LIFO) method for input and output of data. This means the last element inserted inside the stack is removed first.



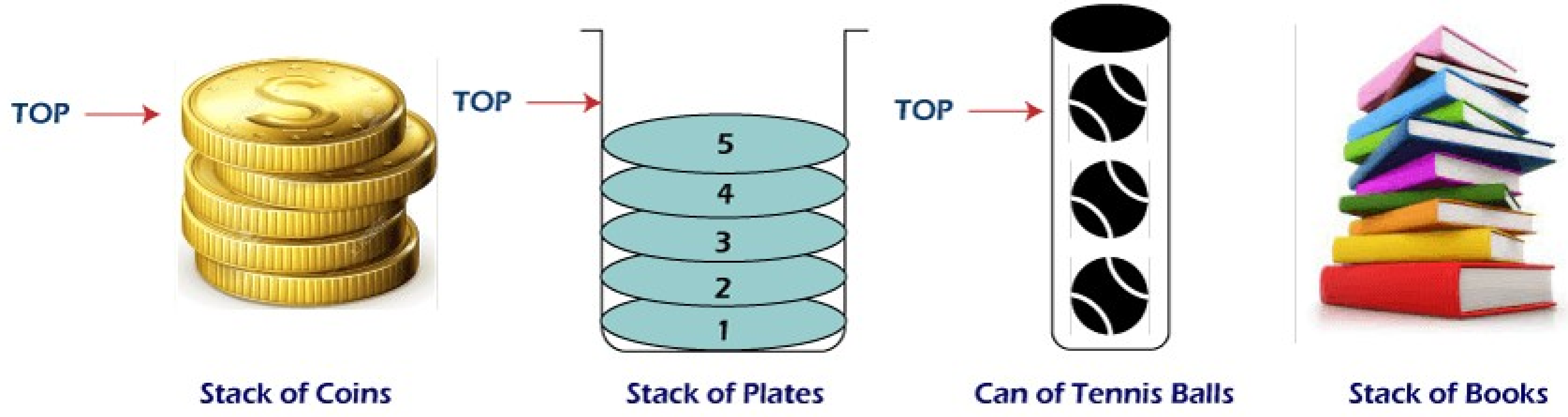
Introduction to Stack

Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure.
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Introduction to Stack

Example of Stack (LIFO)



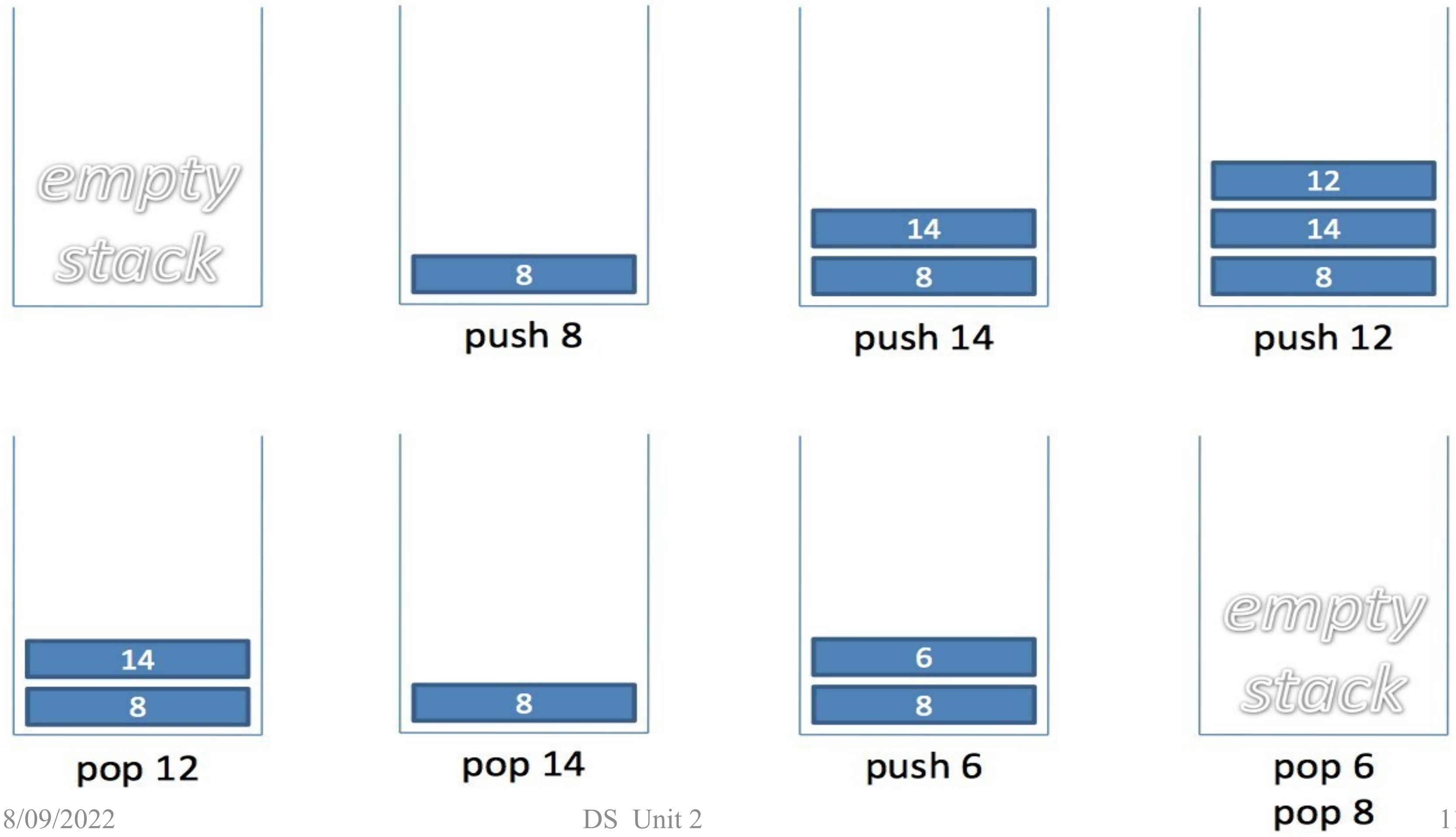
Introduction to Stack

Stack Operations

- push(): This function adds data elements to the stack.
- pop(): It removes the top element from the stack.
- top(): Returns the topmost element i.e. element at the top position of the stack.
- isEmpty(): Checks whether the stack is empty or not i.e. Underflow condition.
- isFull(): Checks whether the stack is full or not i.e. Overflow condition

Introduction to Stack

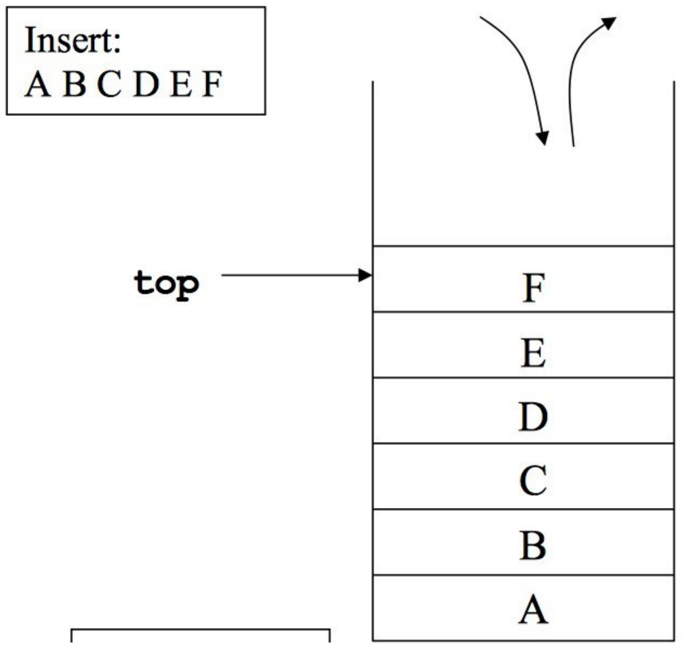
Stack Example

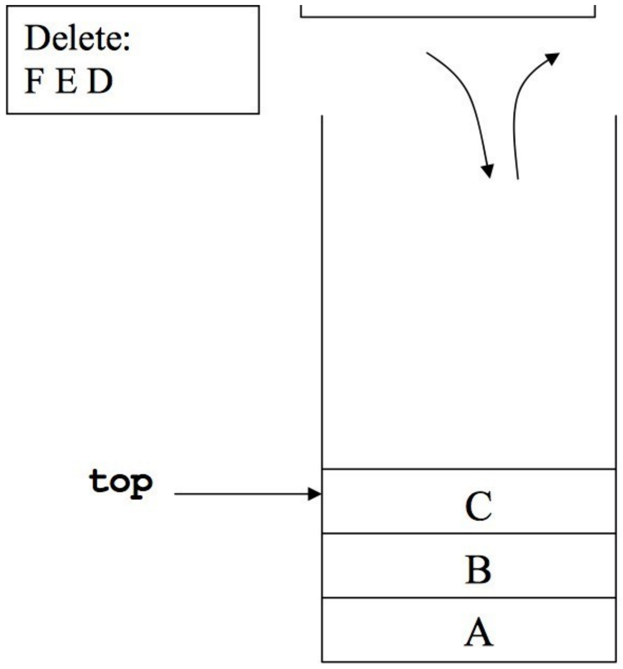


Introduction to Stack

Implementation of Stack

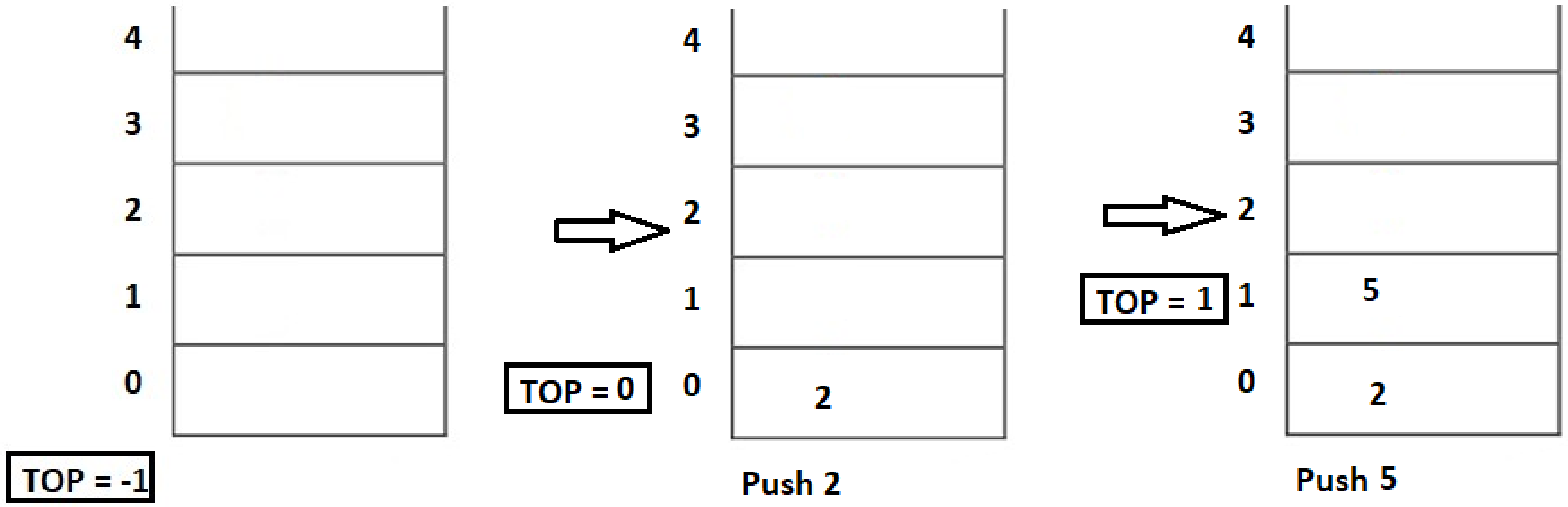
- Using Array
- Using Link List



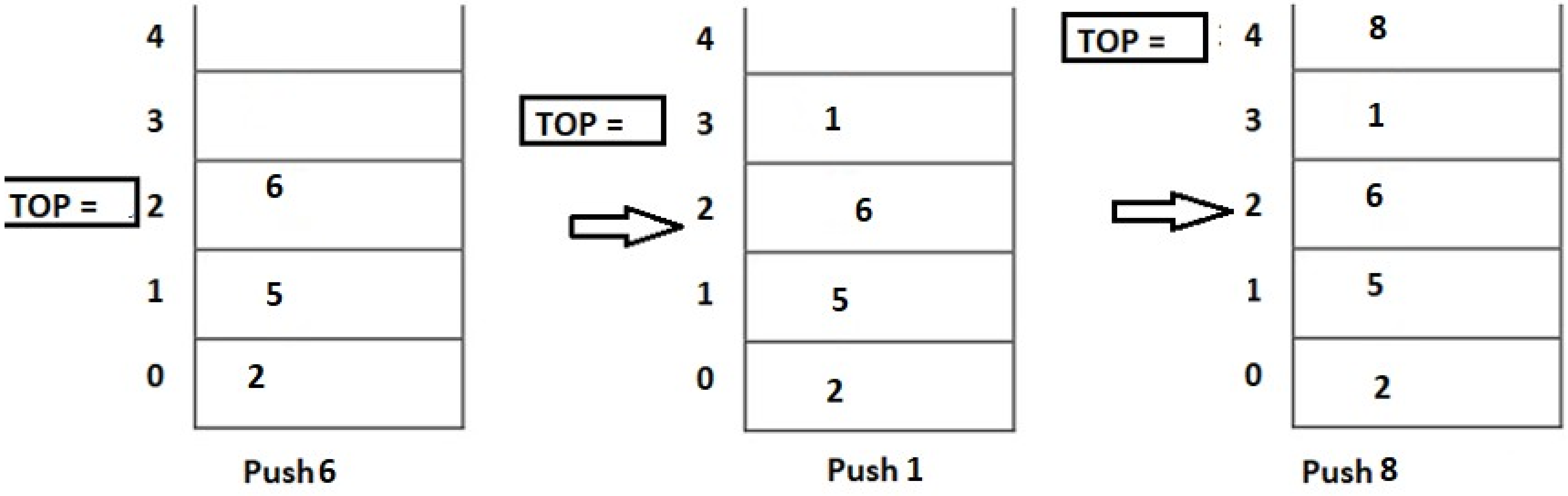


Introduction to Stack

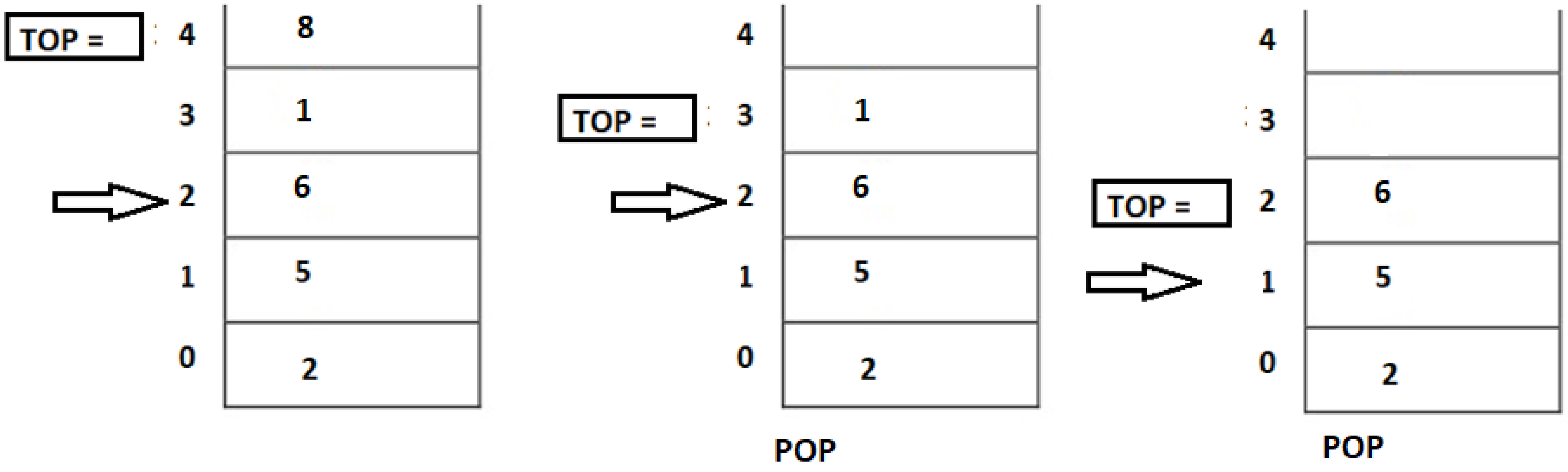
Stack Implementation using Array ...



Introduction to Stack



Introduction to Stack



Introduction to Stack

Stack Implementation using Array using array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1 - Create a one dimensional array with fixed size (int stack[SIZE])

Step 2 - Define a integer variable 'top' and initialize with '-1'. (int top = -1)

Step 3 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Introduction to Stack

Algorithm: PUSH(Insert) Operation in Stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

```
Step 1: If Top=Max-1
Print “Overflow : Stack is full” and Exit
End If
Step 2: Top=Top+1
Step 3: Stack[TOP]=Element
Step 4: End
```

Introduction to Stack

Algorithm: POP(Delete) Operation in Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

```
Step 1: If TOP=-1  .Print "Underflow: Stack is empty" and Exit
End if
Step 2: Set Del_element=Stack[Top]
Step 3: Top=Top-1
Step 4: delete Del_Element
Step 5: End
```

Introduction to Stack

display() - Displays the elements of a Stack using array

We can use the following steps to display the elements of a stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one .
- **Step 3 -** Repeat above step until **i** value becomes '0'.

Introduction to Stack

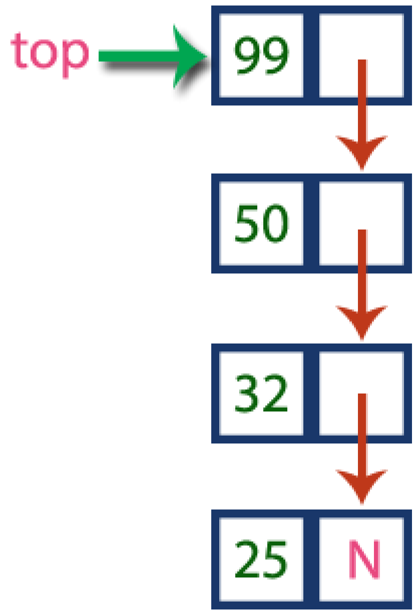
Stack Implementation using Link List

- The major problem with the stack implemented using an array is, it works only for a fixed number of data values.
- Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.
- Stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation.

Introduction to Stack

Stack Implementation using Link List

- In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'.
- Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



Introduction to Stack

Stack Implementation using Link List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 2 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 3 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

Introduction to Stack

push(value) - Inserting an element into the Stack using Link List

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top == NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

Introduction to Stack

pop() - Deleting an Element from a Stack using Link List

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

Introduction to Stack

display() - Displaying stack of elements using Link List

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

Introduction to Stack

Stack Application

1. Evaluation of Arithmetic Expressions
2. Backtracking
3. Reverse a Data
4. Processing Function Calls

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".

Evaluation of Arithmetic Expressions

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

1. Infix Notation
2. Prefix Notation
3. Postfix Notation

Introduction to Stack

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.

Introduction to Stack

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.

Introduction to Stack

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.

Introduction to Stack

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-+

Introduction to Stack

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1.Push “(“onto Stack, and add “)” to the end of X.

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

3.If an operand is encountered, add it to Y.

4.If a left parenthesis is encountered, push it onto Stack.

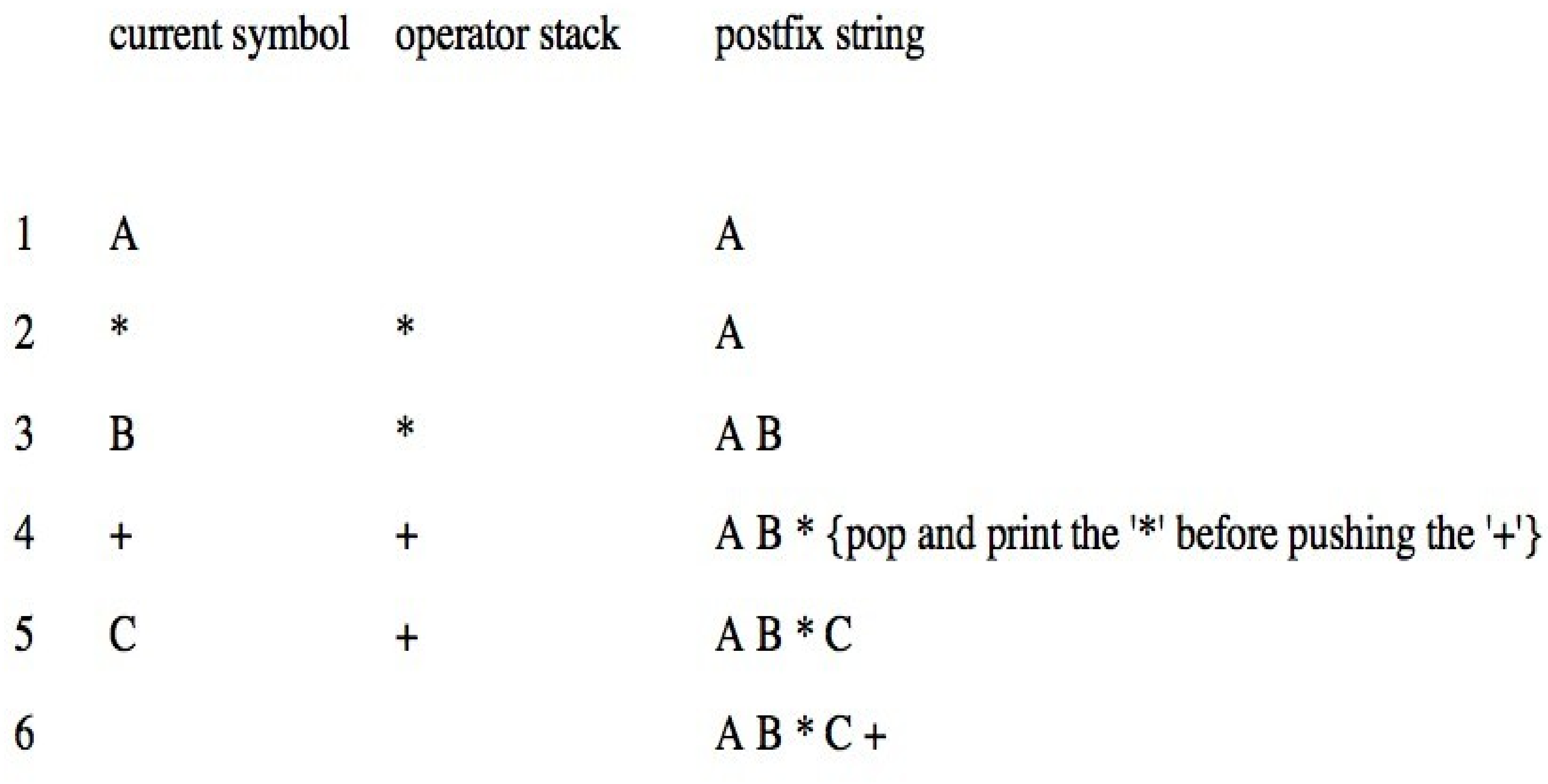
Introduction to Stack

```
5. If an operator is encountered ,then:
    Repeatedly pop from Stack and add to Y each operator (on the top of
    Stack) which has the same precedence as or higher precedence than
    operator.
    Add operator to Stack.   [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top
       of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
       [End of If]
    7. END.
```

Introduction to Stack

Infix to Postfix using stack

- Example A*B+C become AB*C+

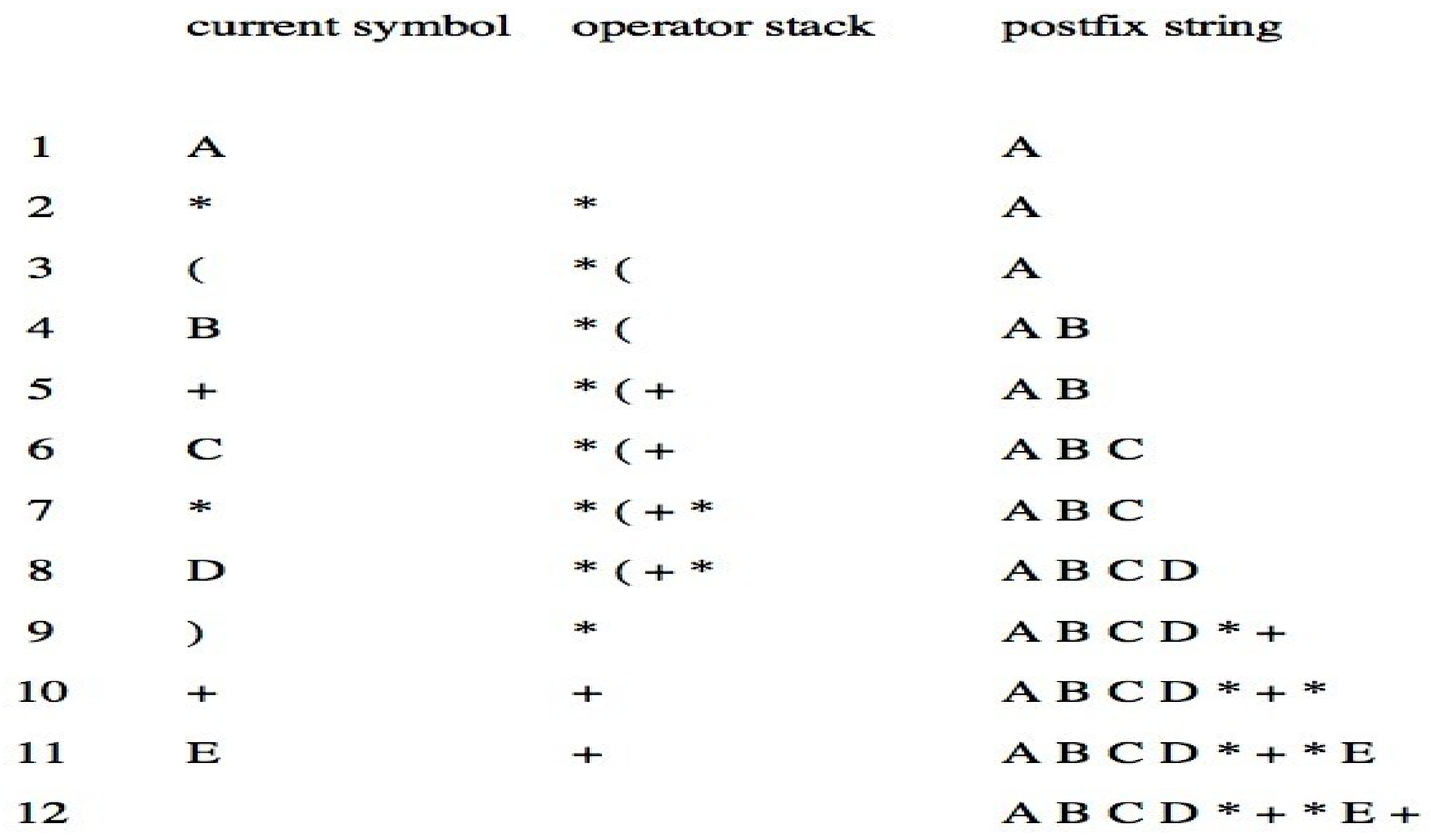


	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

Introduction to Stack

Infix to Postfix using stack ...

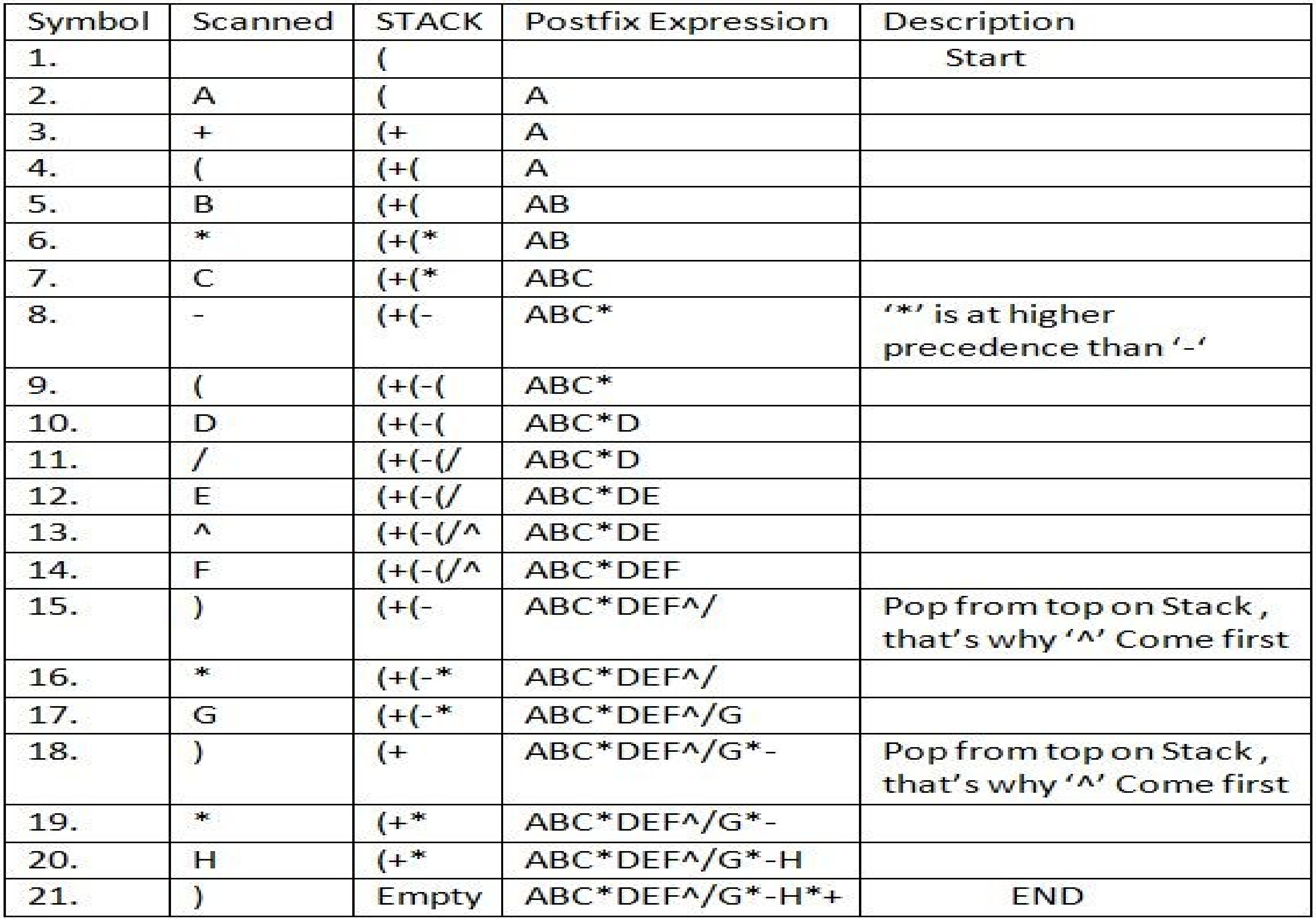
Example A * (B + C * D) + E becomes A B C D * + * E



	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Introduction to Stack

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Introduction to Stack

Algorithm of Infix to Prefix

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

Introduction to Stack

Algorithm of Infix to Prefix

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 same

or higher precedence than the operator.

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

Introduction to Stack

Evaluation of Postfix Expressions Using Stack

Algorithm

```
1) Add ) to postfix expression.
2) Read postfix expression Left to Right until ) encountered
3) If operand is encountered, push it onto Stack      [End If]
4) If operator is encountered, Pop two elements
       i) A -> Top element
       ii) B-> Next to Top element
       iii) Evaluate B operator A
       push B operator A onto Stack
5) Set result = pop
6) END
```

Introduction to Stack

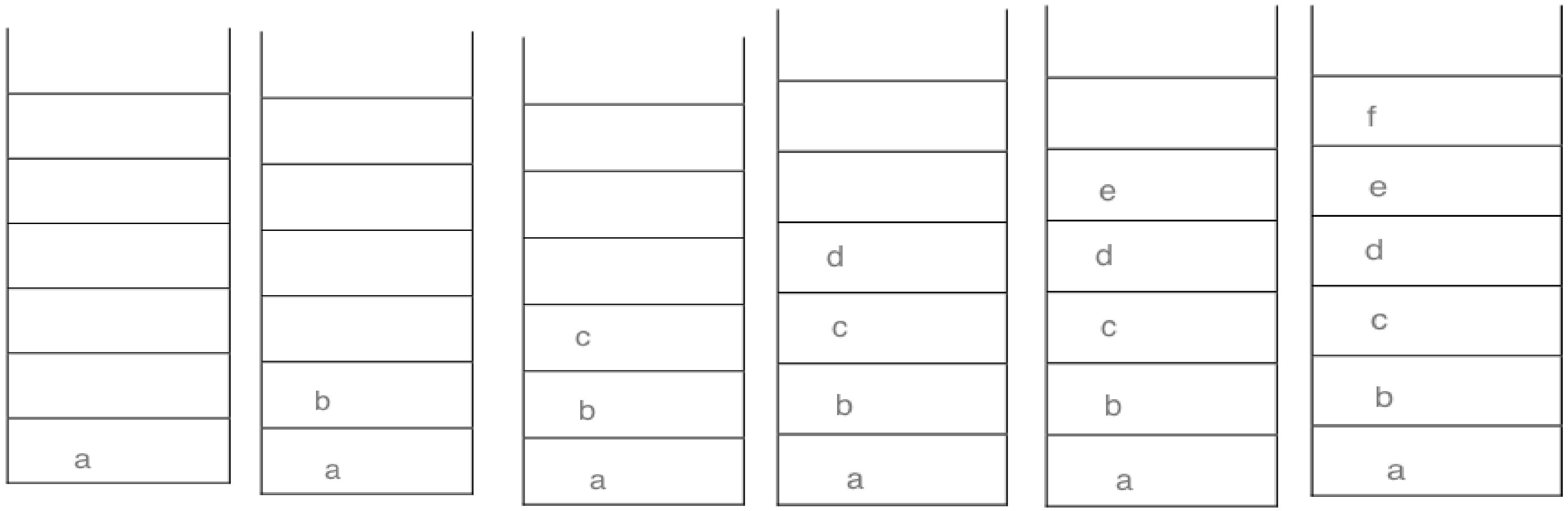
Evaluating Arithmetic Postfix Expression

PostFix Expression 2 3 4 + * 5 *

Move	Token	Stack
1	2	2
2	3	2 3
3	4	2 3 4
4	+	2 7 (3+4=7)
5	*	14 (2*7=14)
6	5	14 5
7	*	70 (14*5=70)

Introduction to Stack

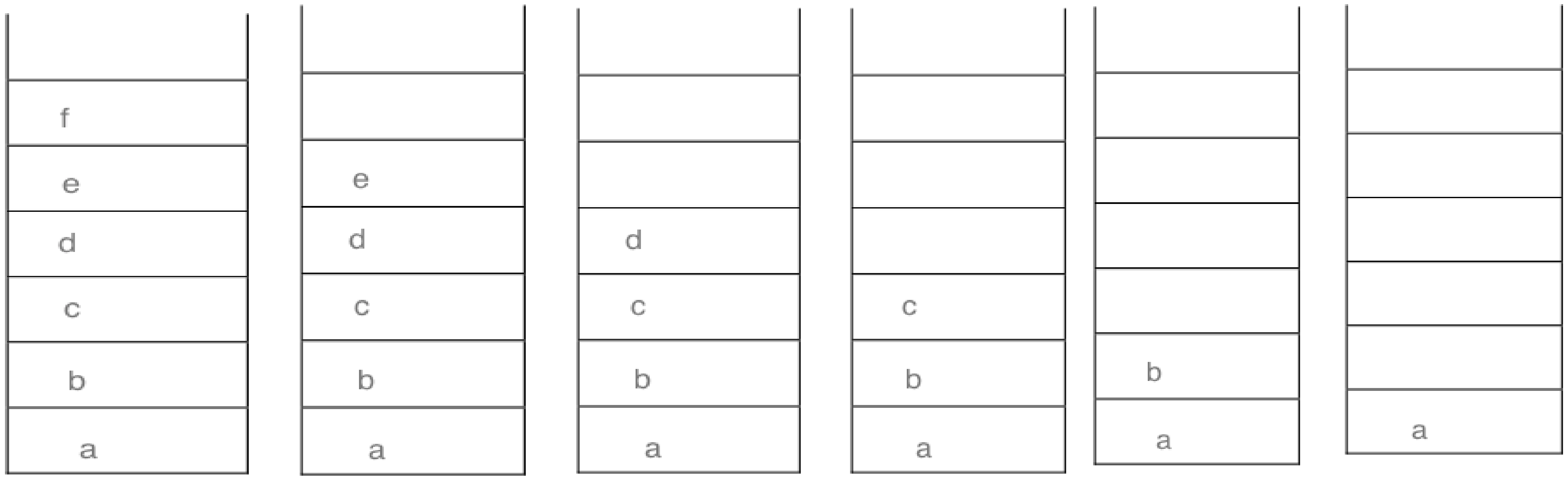
Reverse String...



String is a b c d e f PUSH to SACK

Introduction to Stack

Reverse String...



Reversed String: f e d c b a POP from SACK

Introduction to Stack

Recursion

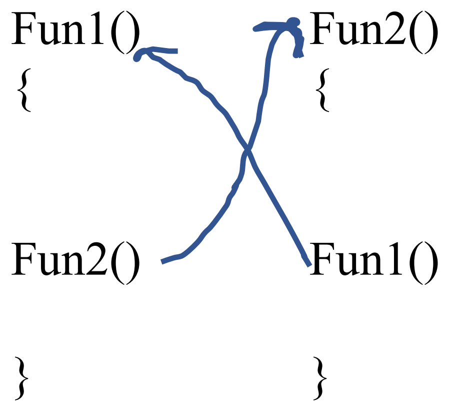
- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied
- To solve a problem recursively, two conditions must be satisfied.
  - First, the problem must be written in a recursive form
  - Second the problem statement must include a base condition

Introduction to Stack

Types of Recursions:

Recursion are mainly of **two types** depending on whether **a function calls itself from within itself** or **more than one function call one another mutually.** The first one is called **direct recursion** and another one is called **indirect recursion**.

```
fun()
{
-------------
------------
----------
fun()
}
```



```
Fun1()          Fun2()
{               {

Fun2()          Fun1()

}               }
```

Introduction to Stack

Tail Recursion:-

A recursive function is said to be tail recursion if the recursive call is last thing done by the function. There is no need to keep the record of previous state.

Non-Tail Recursion:-

A recursive function is said to be non tail recursion if the recursive call is not the last thing done by the function. After returning back there is something left to evaluate.

Introduction to Stack

Tail Recursion:-

```
#example of tail recursion
def fun(n):
    if n==0:
        return
    else:
        print(n)
        return fun(n-1)
n=int(input("enter number"))
fun(n)
```

NIET Greater Noida GET FUTURE READY

Introduction to Stack

Non-Tail Recursion:-

```
#example of non- tail recursion
def fun(n):
    if n==0:
        return
    fun(n-1)
    print(n)
n=int(input("enter number"))
fun(n)
```

Introduction to Stack

Calculate Factorial of the number using Recursion.

$$\begin{tikzcd}
\boxed{4!} & = 4 * 3! \arrow[dl] & = 4 * 6 & = \boxed{24} \\
3! & = 3 * 2! \arrow[dl] & = 3 * 2 & = 6 \arrow[ul] \\
2! & = 2 * 1! \arrow[dl] & = 2 * 1 & = 2 \arrow[ul] \\
1! & = 1 \arrow[ur] & &
\end{tikzcd}$$

Introduction to Stack

Calculate Factorial of the number using Recursion.

```
def factorial(n):

    if n == 0:
        return 1

    return n * factorial(n-1)

num = 5;
print("Factorial of", num, "is",
factorial(num))
```

Recursion

Program for Fibonacci numbers

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..
- In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation
  F(n) = F(n-1) + F(n-2)with seed values F0 = 0 and F1 = 1.

Recursion

Program to print the fibonacci series upto n_terms

```
def recur_fibo(n):
    if n ==0:
        return 0
    elif n==1:
        return 1
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = int(input("enter number"))
print("Fibonacci sequence:")
for i in range(nterms):
       print(recur_fibo(i))
```

Introduction to Stack

<u>Advantages of recursion</u>

1. The code may be easier to write.

2. To solve such problems which are naturally recursive such as tower of Hanoi.

3. Reduce unnecessary calling of function.

4. Extremely useful when applying the same solution.

5. Recursion reduce the length of code.

6. It is very useful in solving the data structure problem.

7. Stacks evolutions and infix, prefix, postfix evaluations etc.

Introduction to Stack

Disadvantages of recursion

1. Recursive functions are generally slower than non-recursive function.

2. It may require a lot of memory space to hold intermediate results on the system stacks.

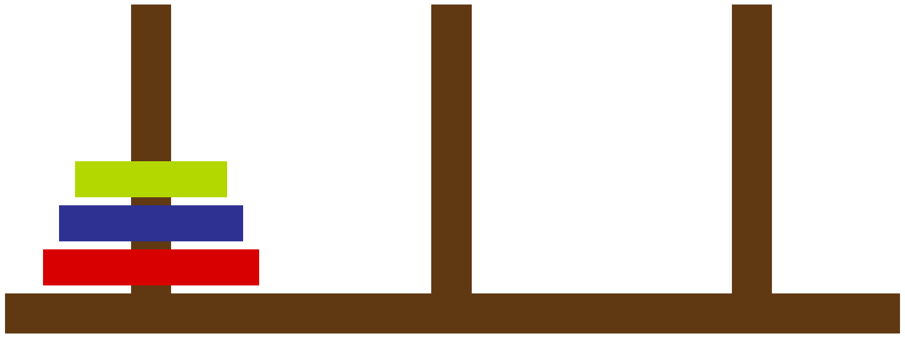
3. Hard to analyze or understand the code.

4. It is not more efficient in terms of space and time complexity.

5. The computer may run out of memory if the recursive calls are not properly checked.

Introduction to Stack

Tower of Hanoi Problem



Tower of Hanoi is a mathematical puzzle where we have three rods and n disks.

Introduction to Stack

Tower of Hanoi Problem

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

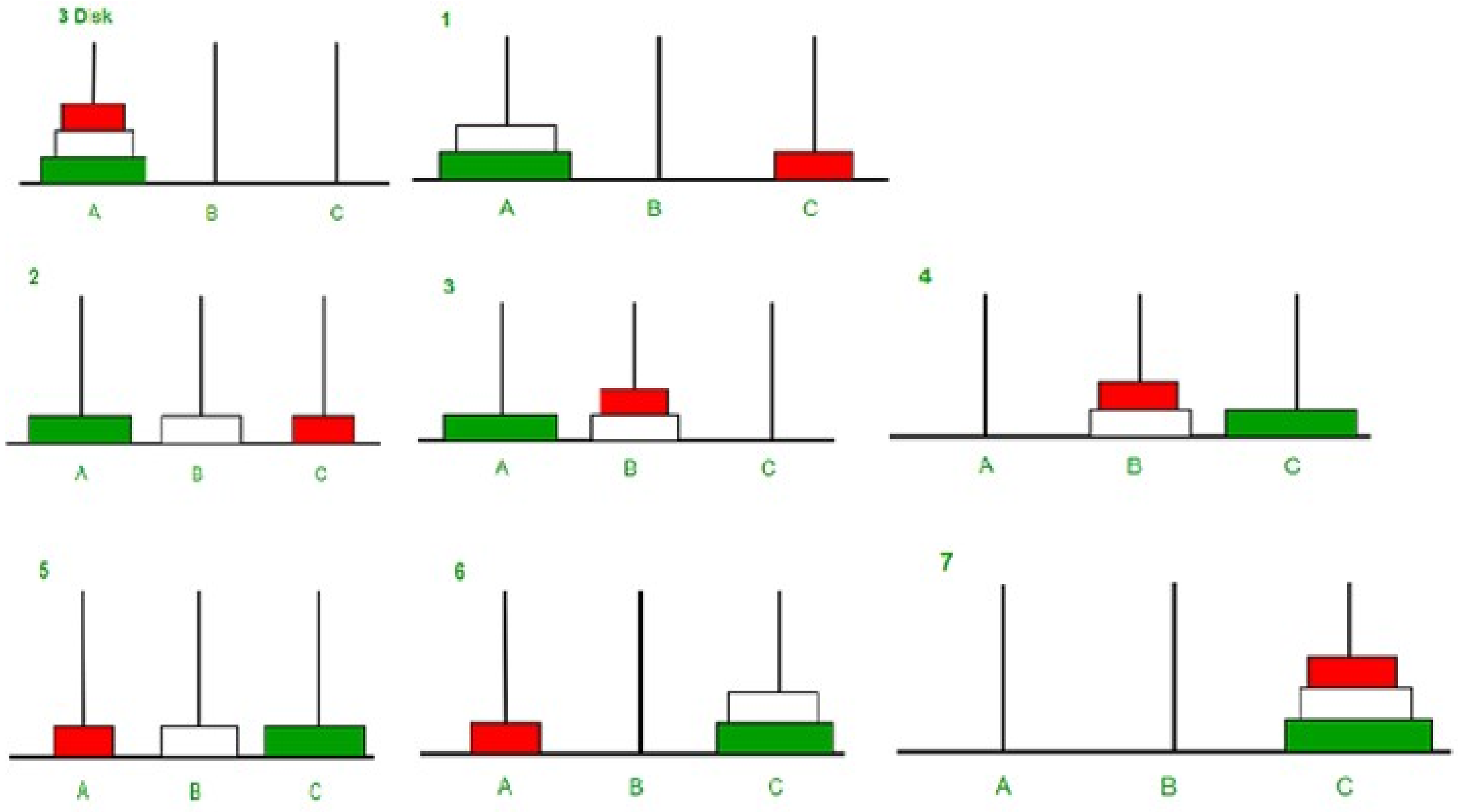
1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3) No disk may be placed on top of a smaller disk.

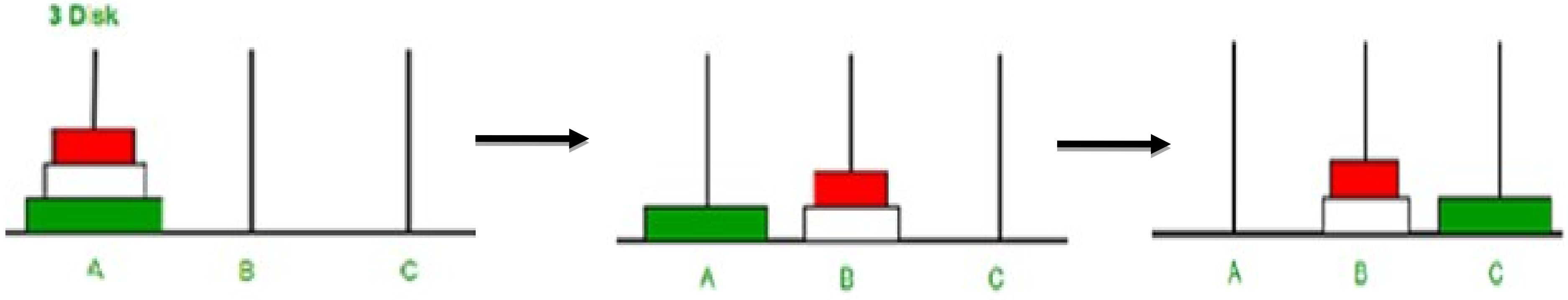
Introduction to Stack

Tower of Hanoi Problem Illustration



Introduction to Stack

Tower of Hanoi Algorithm Concept



Introduction to Stack

A recursive algorithm for Tower of Hanoi can be driven as follows

The steps to follow are −

Step 1 − Move n-1 disks from **source** to **aux**

Step 2 − Move nth disk from **source** to **dest**

Step 3 − Move n-1 disks from **aux** to **dest**

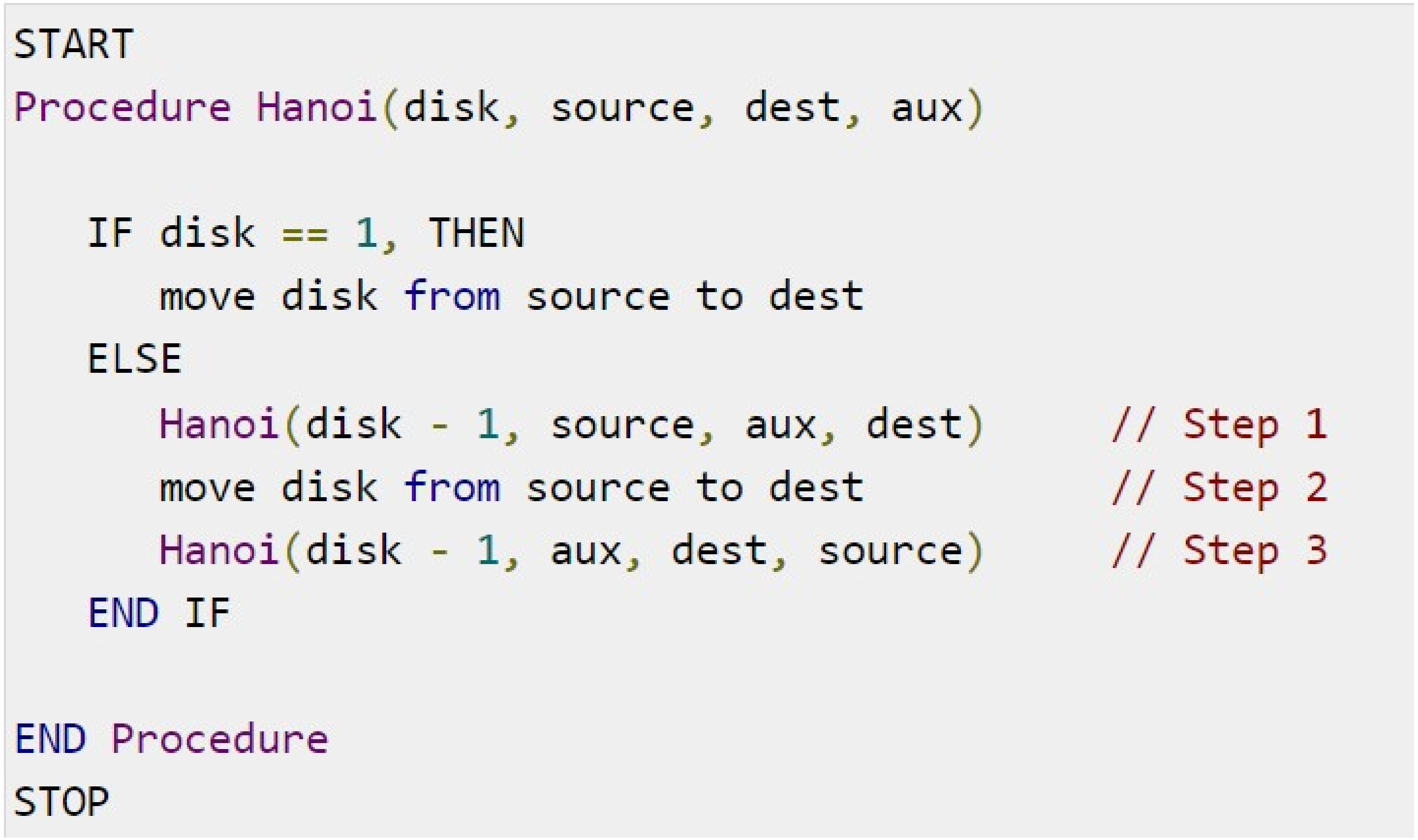
Introduction to Stack

A recursive algorithm for Tower of Hanoi can be driven as follows

```
Input : 2
Output :
       Disk 1 moved from A to B
       Disk 2 moved from A to C
       Disk 1 moved from B to C
Input : 3
Output :
       Disk 1 moved from A to C
       Disk 2 moved from A to B
       Disk 1 moved from C to B
       Disk 3 moved from A to C
       Disk 1 moved from B to A
       Disk 2 moved from B to C
       Disk 1 moved from A to C
```

Introduction to Stack

A recursive algorithm for Tower of Hanoi can be driven as follows

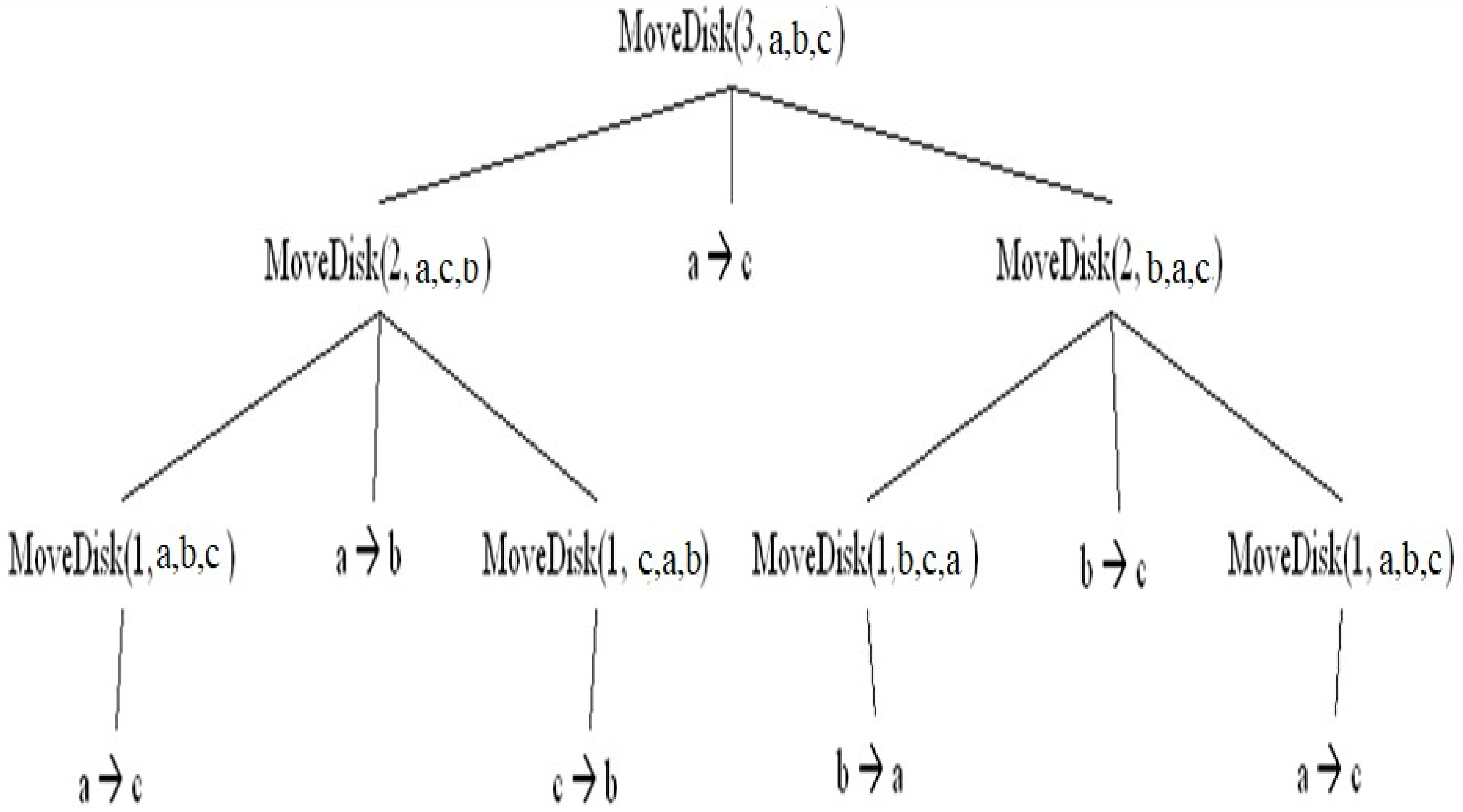


```
START
Procedure Hanoi(disk, source, dest, aux)

   IF disk == 1, THEN
      move disk from source to dest
   ELSE
      Hanoi(disk - 1, source, aux, dest)     // Step 1
      move disk from source to dest          // Step 2
      Hanoi(disk - 1, aux, dest, source)     // Step 3
   END IF

END Procedure
STOP
```

Introduction to Stack



Introduction to Stack

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high	Relatively lower time complexity