Subject: Data Structures

Module Number: 01

Module Name: Stack and its Applications

Syllabus

Stacks: Basic Stack Operations, Representation of a Stack using Static Array and Dynamic Array, Multiple stack implementation using single array.

Stack Applications: Reversing list, Factorial Calculation, Infix to postfix Transformation, Evaluating Arithmetic Expressions and Towers of Hanoi.

Aim:

Understand all basic operations and multiple implementations of stack and its applications.

Objectives:

The objectives of this module are as follows:

- Analyze the basic stack operations to understand their functionality.
- Demonstrate the representation of a stack using both static arrays and dynamic arrays.
- Implement multiple stacks using a single array efficiently.
- Apply stack data structures to practical problem-solving in various domains.
- Evaluate arithmetic expressions using stack-based algorithms.
- Solve complex problems such as the Towers of Hanoi using stack-based approaches.

Outcomes:

At the end of this module, you are expected to:

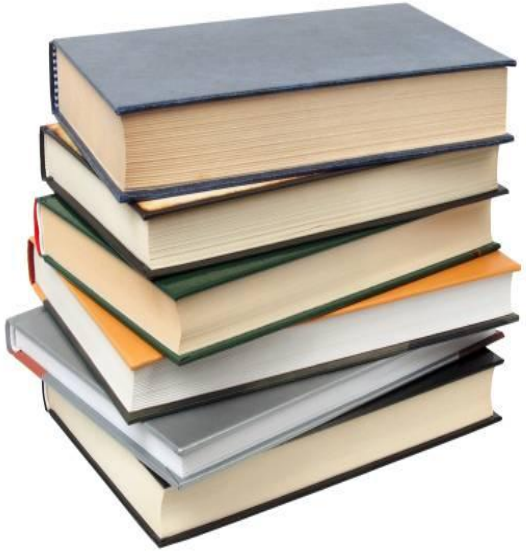
- Analyse and describe the fundamental operations of a stack, including push, pop, and peek.
- Implement stacks with static and dynamic arrays for efficient memory management.
- Demonstrate proficiency in implementing multiple stacks using a single array.
- Apply stack structures to tasks like list reversal, factorial calculation, and infix to postfix transformations.
- Evaluate arithmetic expressions with stack-based algorithms for accurate results..

Table of Contents

Stacks: Basic Stack Operations

Examples from real life

- Stack of Books



- Stack of containers

Figure: Examples of stack from real life

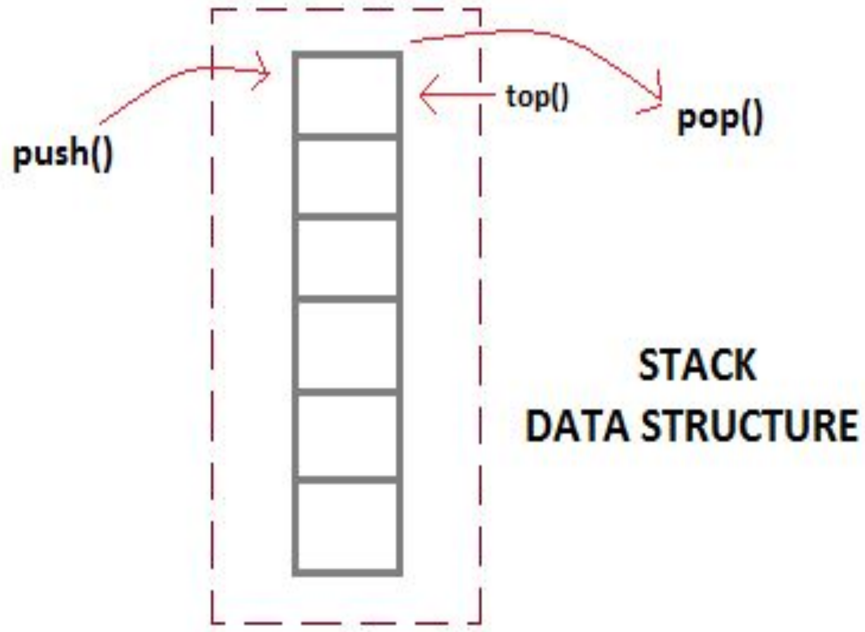
Stacks: Basic Stack Operations (Contd.…)

Stack- Definition

- A stack is a LIFO structure and physically it can be implemented as an array or as a linked list.
- A stack implemented as an array inherits all the properties of an array and if implemented as a linked list.
- An insertion in a stack is called pushing.
- A deletion from a stack is called popping.

Representation of Stack

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing of elements in a particular order.



Representation of Stack (Contd….)

Features of Stack:

1. Stack is an ordered list of similar data type

2. Stack is a LIFO structure. (Last in First out). Here, the element which is placed last, is accessed first

3. push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion 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

Representation of Stack (Contd….)

Stack Specification:

Definitions: (provided by the user)

- MAX_ITEMS: Max number of items that might be on the stack
- ItemType: Data type of the items on the stack

Operations

- MakeEmpty
- Boolean IsEmpty
- Boolean IsFull
- Push (ItemType newItem)
- Pop (ItemType& item)

Representation of Stack (Contd….)

Push (ItemType newItem) :

- Function: Adds newItem to the top of the stack.
- Preconditions: Stack has been initialized and is not full.
- Post-conditions: newItem is at the top of the stack.

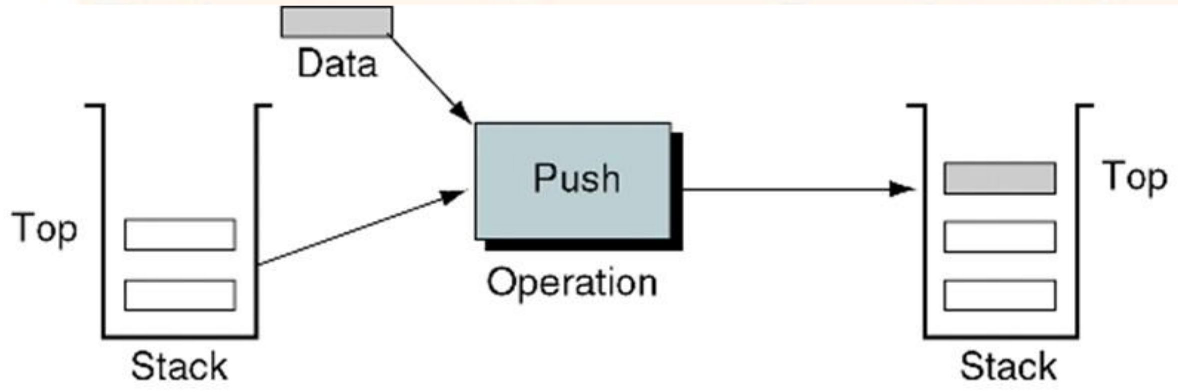


Figure: Push operation on stack

Representation of Stack (Contd….)

Pop (ItemType & item)

- Function: Removes topItem from stack and returns it in item.
- Preconditions: Stack has been initialized and is not empty.
- Post-conditions: Top element has been removed from stack and item is a copy of the removed element.

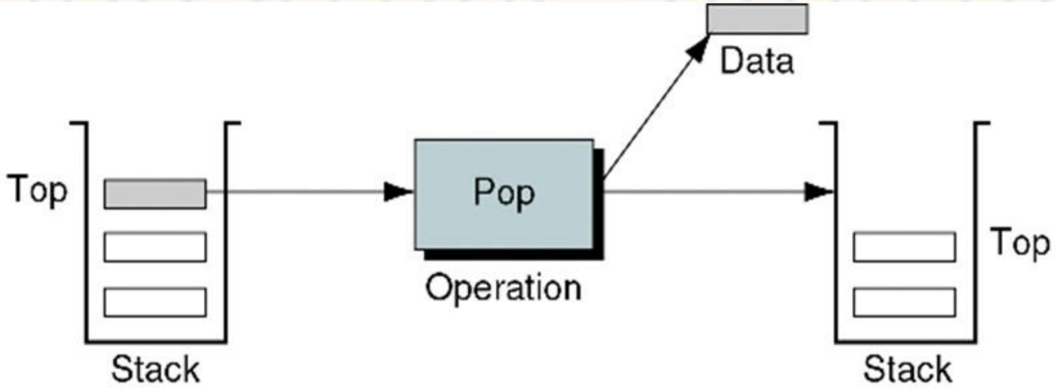


Figure: Pop operation on stack

Representation of Stack (Contd….)

StackTop (ItemType & item)

Function: returns topItem from stack and returns it in item.

Preconditions: None

Post-conditions: No Changes

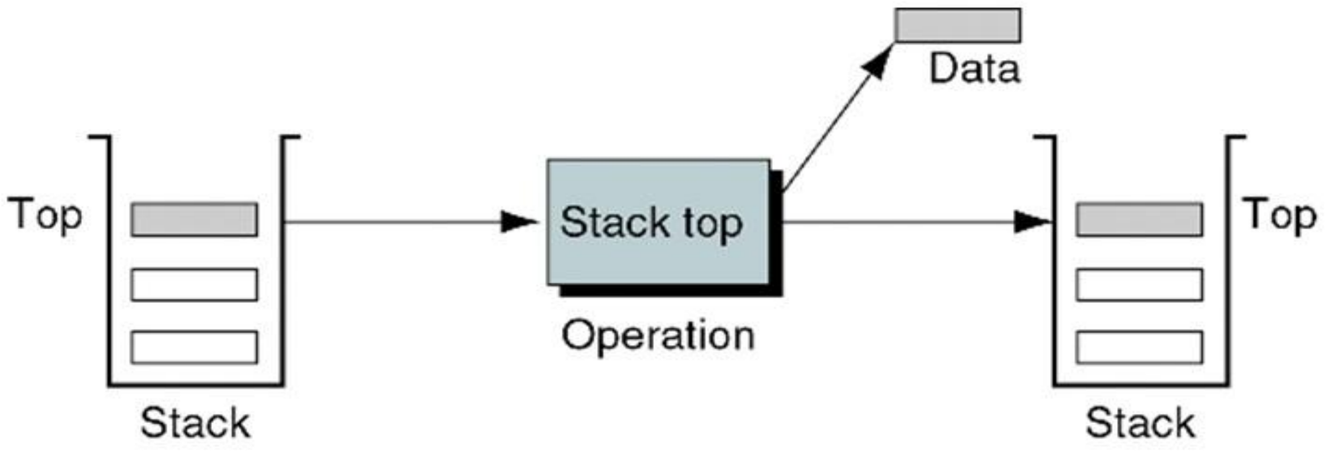


Figure: Pop operation on stack

Representation of Stack (Contd….)

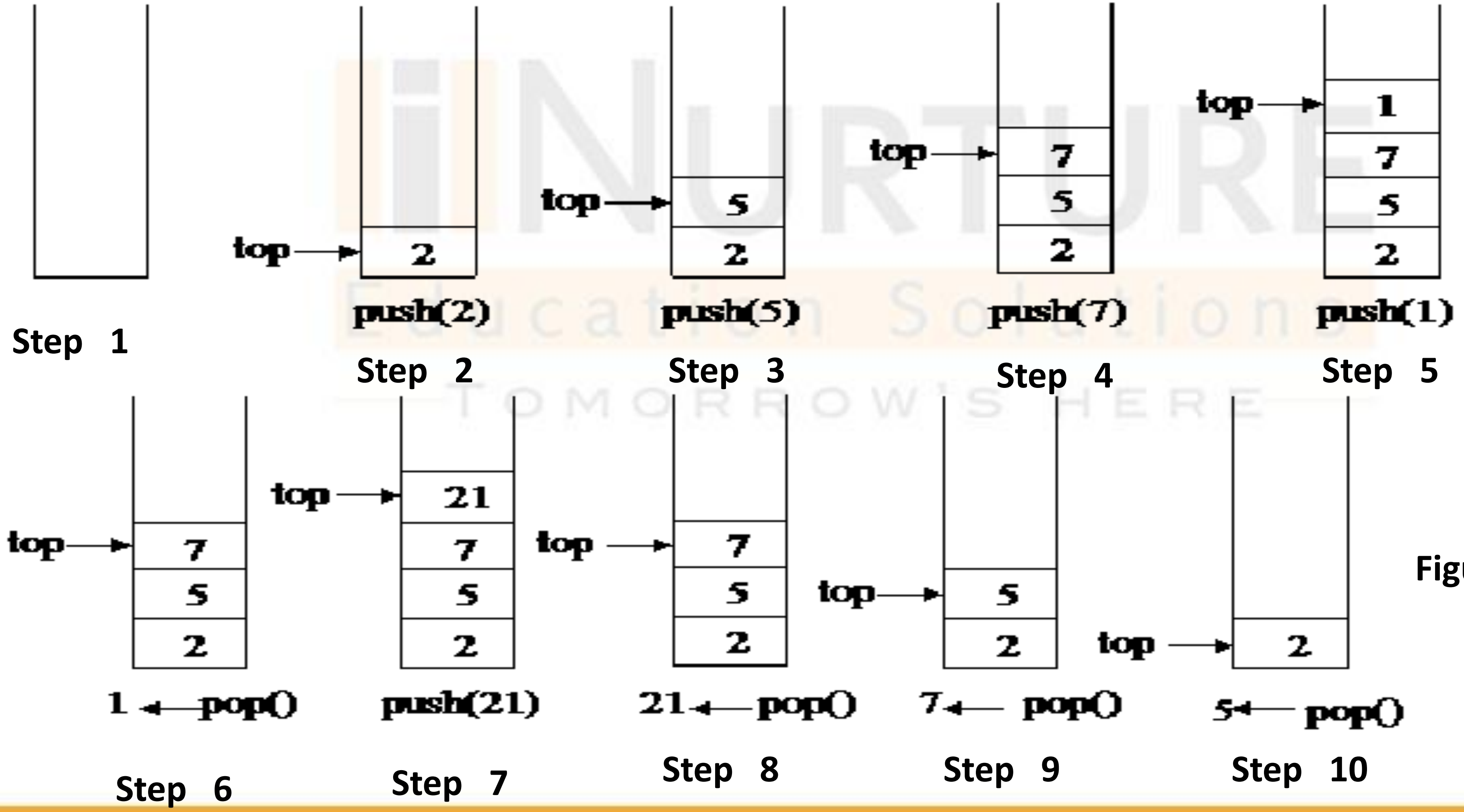


Figure: Stack Operations

Representation of Stack (Contd….)

Stack Implementation (Using C++)

```
#include "ItemType.h"
// Must be provided by the user of the class
// Contains definitions for MAX_ITEMS and ItemType
 class StackType
{ public:
    StackType();
    void MakeEmpty();
            bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
```

Representation of Stack (Contd….)

Stack Implementation (Using C++)

```
void Pop(ItemType&);
private:
        int top;
        ItemType items[MAX_ITEMS];
};
```

Representation of Stack (Contd….)

```
StackType::StackType()
{
 top = -1;
}
void StackType::MakeEmpty()
{
 top = -1;
}
bool StackType::IsEmpty() const
{
 return (top == -1);
}
```

Representation of Stack (Contd….)

```
bool StackType::IsFull() const
{
 return (top == MAX_ITEMS-1);
}
void StackType::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}
void StackType::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Representation of Stack (Contd….)

- **Stack overflow**

The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
stack.Push(item);
```

- **Stack underflow**

The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
stack.Pop(item);
```

Representation of Stack as an ADT

- A stack is an Abstract Data Type that serves as a collection of elements.

Two operations:

- Push - which adds an element to the collection.
- Pop - which removes the most recently added element that was not yet removed.

Representation of a Stack using Static Array and Dynamic Array

Array and Linked List Representation:

- Linked list is a linear collection of data elements, in which linear order is not given by their physical placement in memory.
- Each element points to the next. It is a data structure consisting of a group of nodes which together represent a sequence.

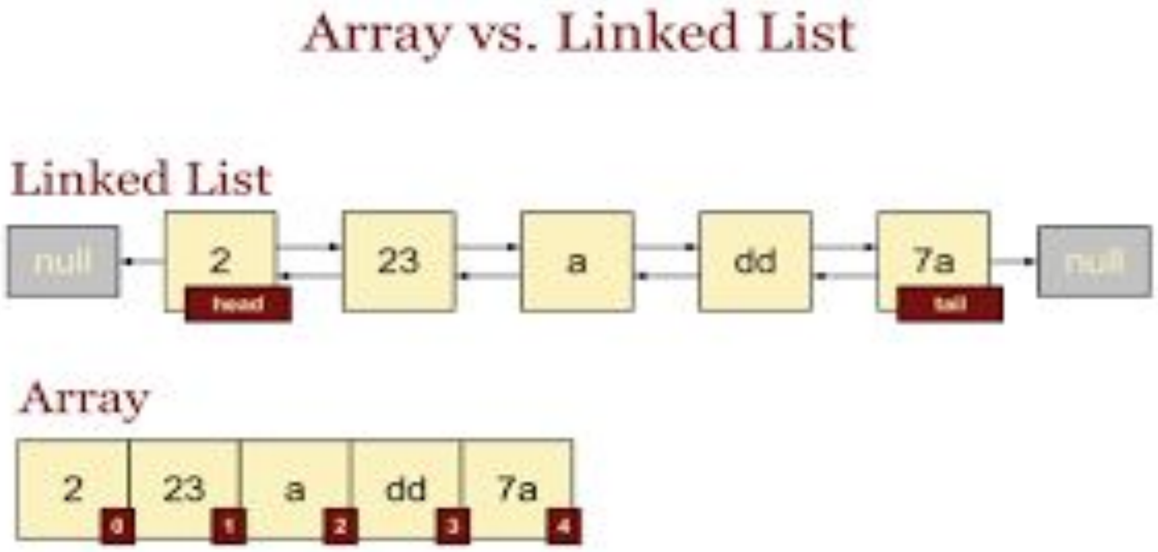


Figure : Array Vs. Linked List

Representation of a Stack using Static Array and Dynamic Array (Contd….)

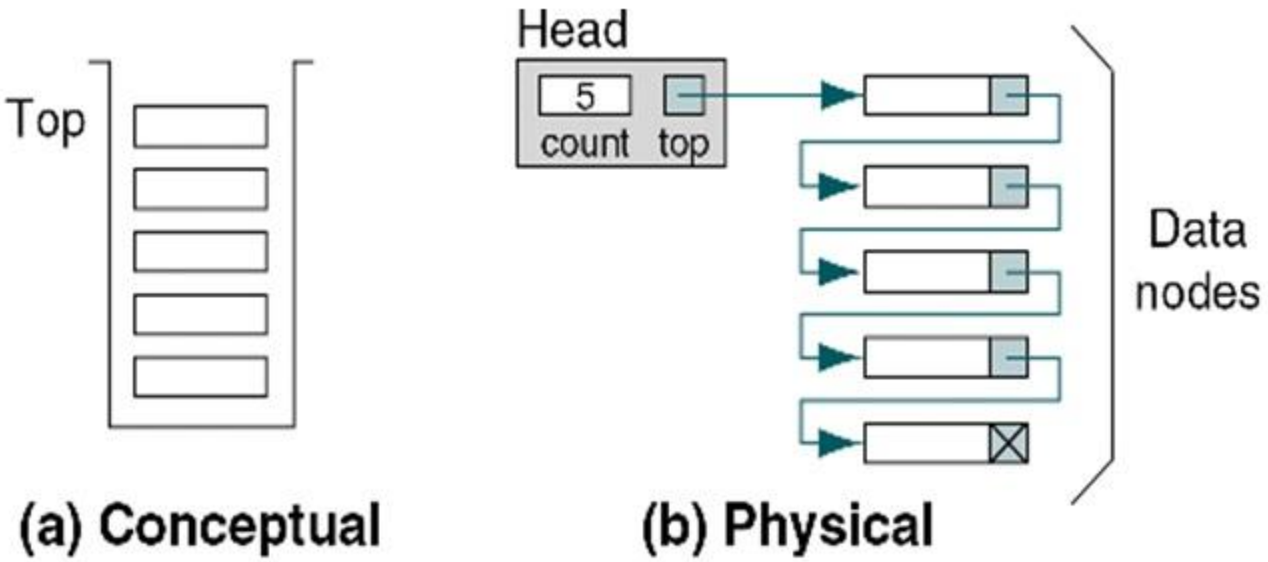
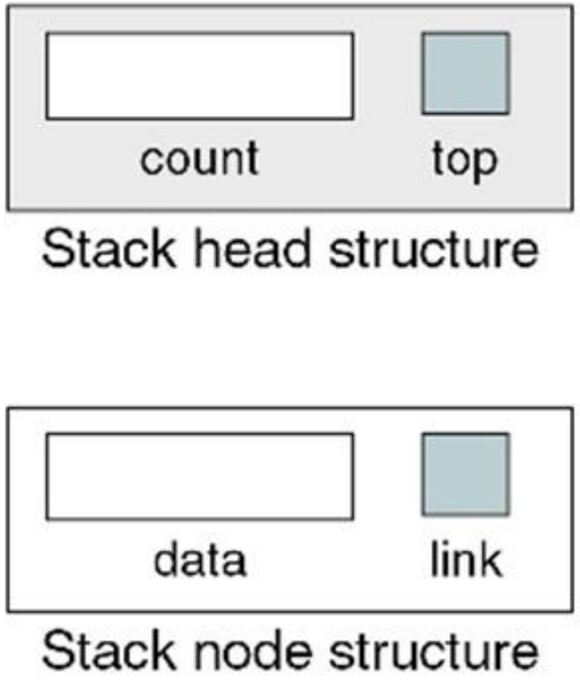


Figure : Conceptual Vs Physical Stack Implementation

Representation of a Stack using Static Array and Dynamic Array (Contd….)




```
stack
   count
   top
end stack


node
   data
   link
end node
```

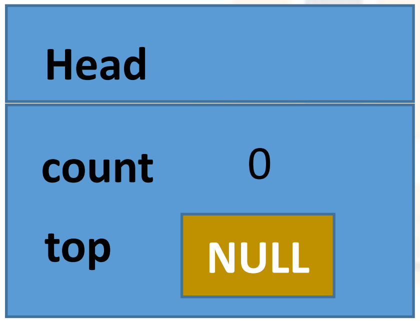
Figure: Data Structure used in Stack implementation using Linked List

Representation of a Stack using Static Array and Dynamic Array (Contd….)

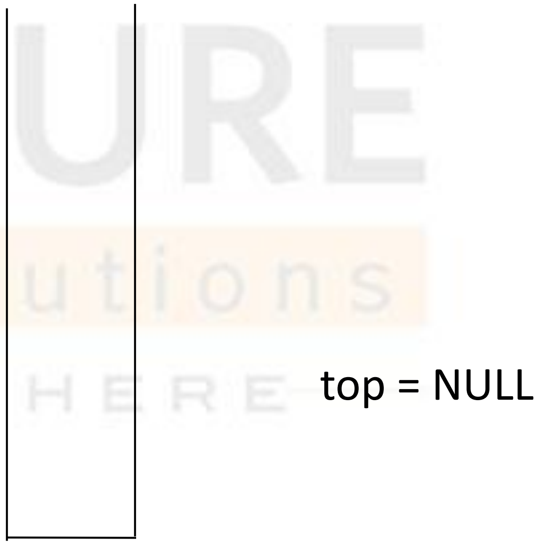
Create Stack

```
Algorithm createStack
Creates and initializes metadata structure.
   Pre     Nothing
   Post    Structure created and initialized
   Return stack head
1  allocate memory for stack head
2  set count to 0
3  set top to null
4  return stack head
end createStack
```

Representation of a Stack using Static Array and Dynamic Array (Contd….)



Physical Implementation



Logical Implementation

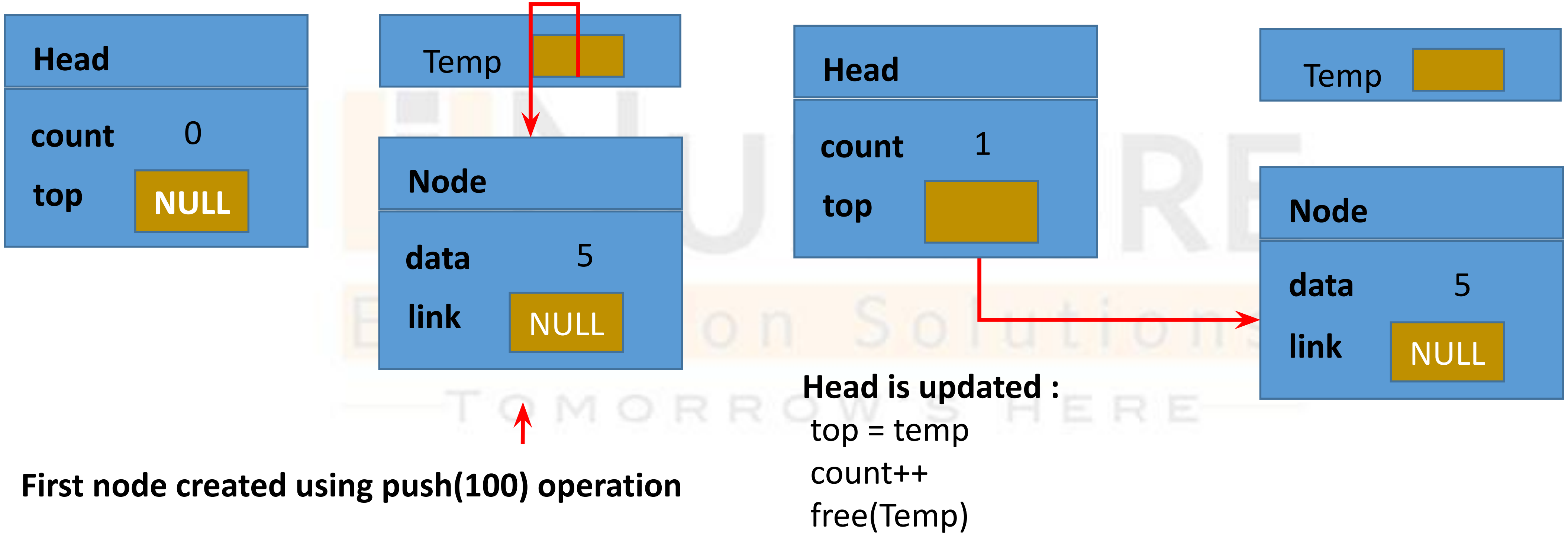
Figure : Logical and Physical implementation of empty stack using linked list

Representation of a Stack using Static Array and Dynamic Array (Contd….)

Push Stack Design

```
Algorithm pushStack (stack, data)
Insert (push) one item into the stack.
   Pre  stack passed by reference
        data contain data to be pushed into stack
   Post data have been pushed in stack
1  allocate new node
2  store data in new node
3  make current top node the second node
4  make new node the top
5  increment stack count
end pushStack
```

Representation of a Stack using Static Array and Dynamic Array (Contd….)



Physical implementation

Figure: Implementation of Push operation on stack using linked list

Representation of a Stack using Static Array and Dynamic Array (Contd….)

Pop Stack Pseudocode

Algorithm: popStack (stack, val)

Removes (pop) one item from top of stack.

Pre: stack passed by reference

Post: data have been removed from stack and top is updated

val contains data removed from top of stack

1. Check whether stack IsEmpty()

   if stack is empty output underflow error

   else goto step 2.

2. Assign the address of head node to Temp pointer

   Temp = top

Representation of a Stack using Static Array and Dynamic Array (Contd….)

3. Update top pointer to next node

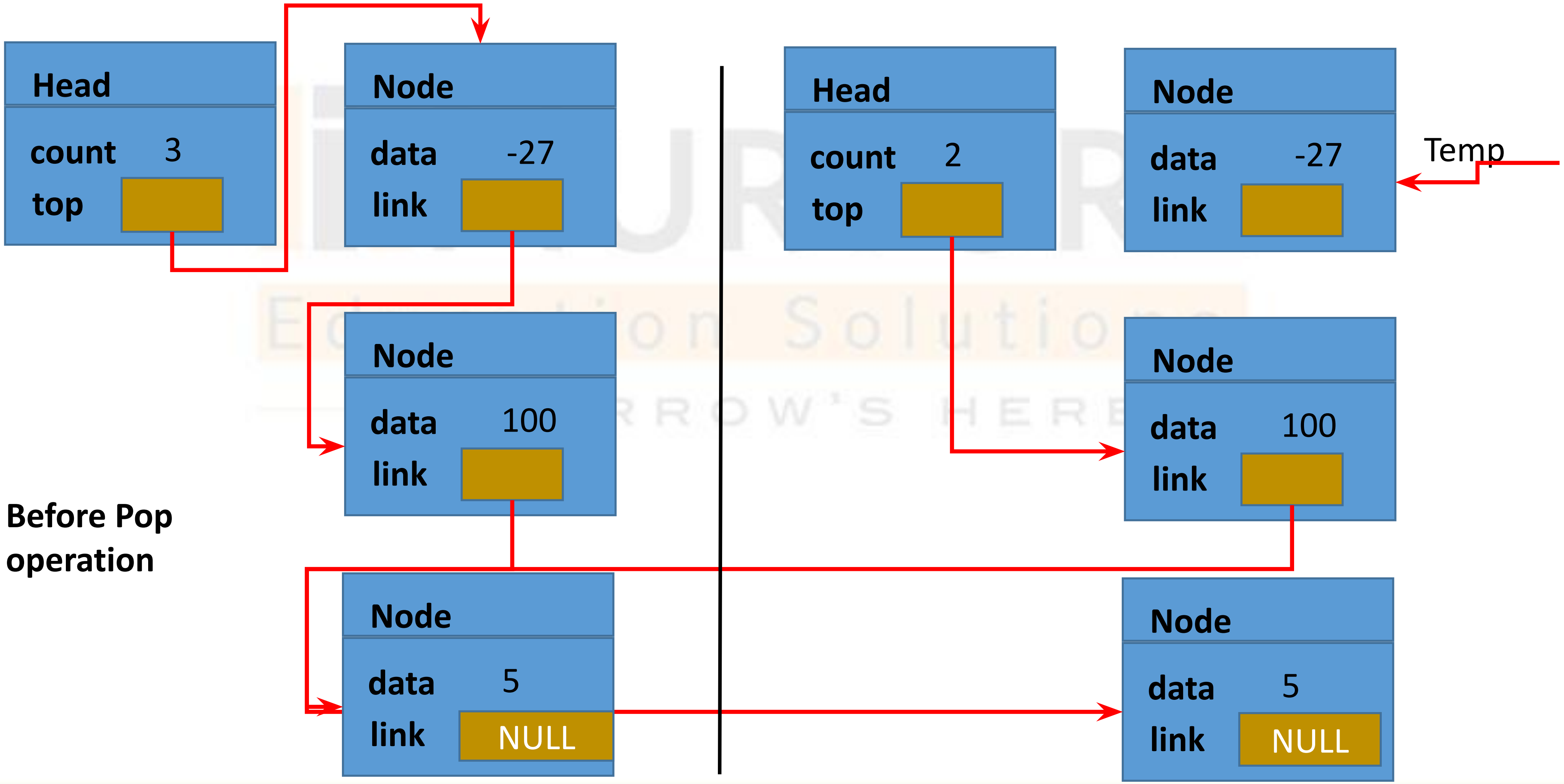
   top= top-> link

4. Decrement count by 1 in head node
5. Assign data in Node pointed by Temp to val

   val = Temp->data

6. Output val
7. Free(Temp)

Representation of a Stack using Static Array and Dynamic Array (Contd….)



Multiple stack implementation using single array

- Multiple stack implementation using a single array is a technique where you can create and manage multiple stacks within a single contiguous block of memory (array). This approach is useful when you want to save memory and need to implement multiple stacks with varying sizes.
- Divide the single array into segments for each stack and keep track of the top position of each stack separately. Each stack's elements are stored within its respective segment, and operations like push and pop are performed based on the top positions of the individual stacks.

Multiple stack implementation using single array

Let's see the step-by-step explanation for a two-stack implementation using a single array:

- We create a **Two Stacks** structure that holds the data array and two top positions, one for each stack.
- We initialize both tops accordingly, and then we can use the **pushStack1, pushStack2, popStack1, and popStack2** functions to perform stack operations on each stack separately.
- The **isStack1Full, isStack2Full, isStack1Empty,** and **isStack2Empty** functions are used to check if a stack is full or empty before performing push or pop operations to avoid stack overflow and underflow.

Multiple stack implementation using single array (Contd….)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
typedef struct
 {
    int data[MAX_SIZE];
    int top1;
    int top2;
} TwoStacks;
```

Multiple stack implementation using single array (Contd….)

```
void initializeTwoStacks(TwoStacks* stacks)
{  stacks->top1 = -1; // Initialize top index for the first stack
   stacks->top2 = MAX_SIZE; // Initialize top index for the second stack at the end of the array
}
int isStack1Empty(TwoStacks* stacks)
{ return stacks->top1 == -1;}
int isStack2Empty(TwoStacks* stacks)
{ return stacks->top2 == MAX_SIZE;}
int isStack1Full(TwoStacks* stacks)
{ return stacks->top1 == stacks->top2 - 1;}
```

Multiple stack implementation using single array (Contd….)

```
int isStack2Full(TwoStacks* stacks)
{    return stacks->top2 == stacks->top1 + 1;}
void pushStack1(TwoStacks* stacks, int value)
{  if (isStack1Full(stacks))
    {  printf("Stack 1 overflow! Cannot push element.\n");
       return;}
   stacks->data[++stacks->top1] = value;}
void pushStack2(TwoStacks* stacks, int value)
{    if (isStack2Full(stacks))
  {  printf("Stack 2 overflow! Cannot push element.\n");
     return; }
```

Multiple stack implementation using single array (Contd….)

```
stacks->data[--stacks->top2] = value;}
int popStack1(TwoStacks* stacks)
{ if (isStack1Empty(stacks))
 { printf("Stack 1 underflow! Cannot pop element.\n");
     return -1; // Return a default value or signal an error in a real scenario. }
   return stacks->data[stacks->top1--];}
int popStack2(TwoStacks* stacks) {
   if (isStack2Empty(stacks)) {
     printf("Stack 2 underflow! Cannot pop element.\n");
     return -1; // Return a default value or signal an error in a real scenario.
   }    return stacks->data[stacks->top2++];}
```

Application of Stacks

Let us understand the use of stack by taking the case of " Backtracking".

Backtracking: This is a process when you need to access the most recent data element in a series of elements

To understand this we take analogy of a Maze.

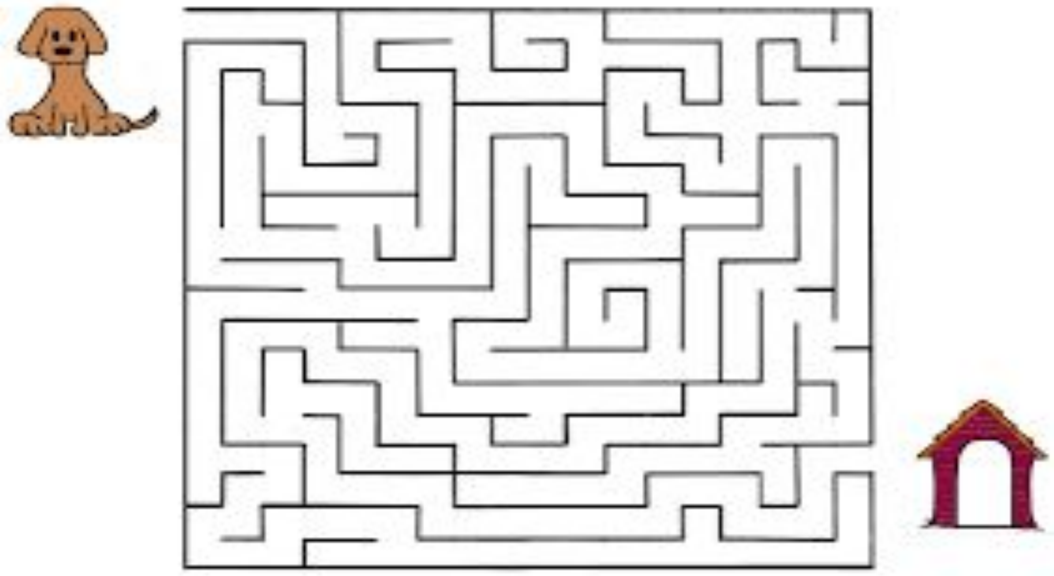


Figure : Example of Backtracking using Maze

Application of Stacks (Contd….)

- Expression evaluation
- Backtracking (game playing, finding paths, exhaustive searching)
- Memory management, run-time environment for nested language features
- Delimiter Matching
- System software and Compiler Design

Stack and its Applications

Application of Stacks (Contd….)

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Reversing list

To reverse the linked list, we need to change the direction of the pointers so that the last node becomes the new head of the list. This process can be done iteratively or recursively.

Iterative Approach:

- We use three pointers, prev, current, and next, to reverse the linked list.
- Initialize prev to NULL, current to the head of the original list, and next to NULL.
- Traverse the list, updating the next pointer of each node to point to its previous node (prev).
- Move prev to current, and current to the next node (next).
- Continue until current becomes NULL.

Reversing list (Contd….)

Recursive Approach:

The recursive approach is an elegant way to reverse the linked list by reversing the rest of the list after the current node.

- The base case is when the current node is NULL or the next node is NULL.
- The current node becomes the new head of the reversed list.
- Recursively reverse the rest of the list and make the next node point to the current node.
- Update the next pointer of the current node to NULL to complete the reversal.

Reversing list (Contd….)

For ex:

Iterative Approach:

```
Node* reverseListIterative(Node* head)
{  Node* prev = NULL;
   Node* current = head;
   Node* next = NULL;
   while (current != NULL)
   {   next = current->next;
       current->next = prev;
       prev = current;
```

Reversing list (Contd….)

```
current = next;   }
  return prev; // 'prev' points to the new head of the reversed list
}
```

Factorial Calculation

Recursion: Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

Stack is not only used to perform recursion but any bunch of nested function calls.

Factorial Calculation (Contd….)

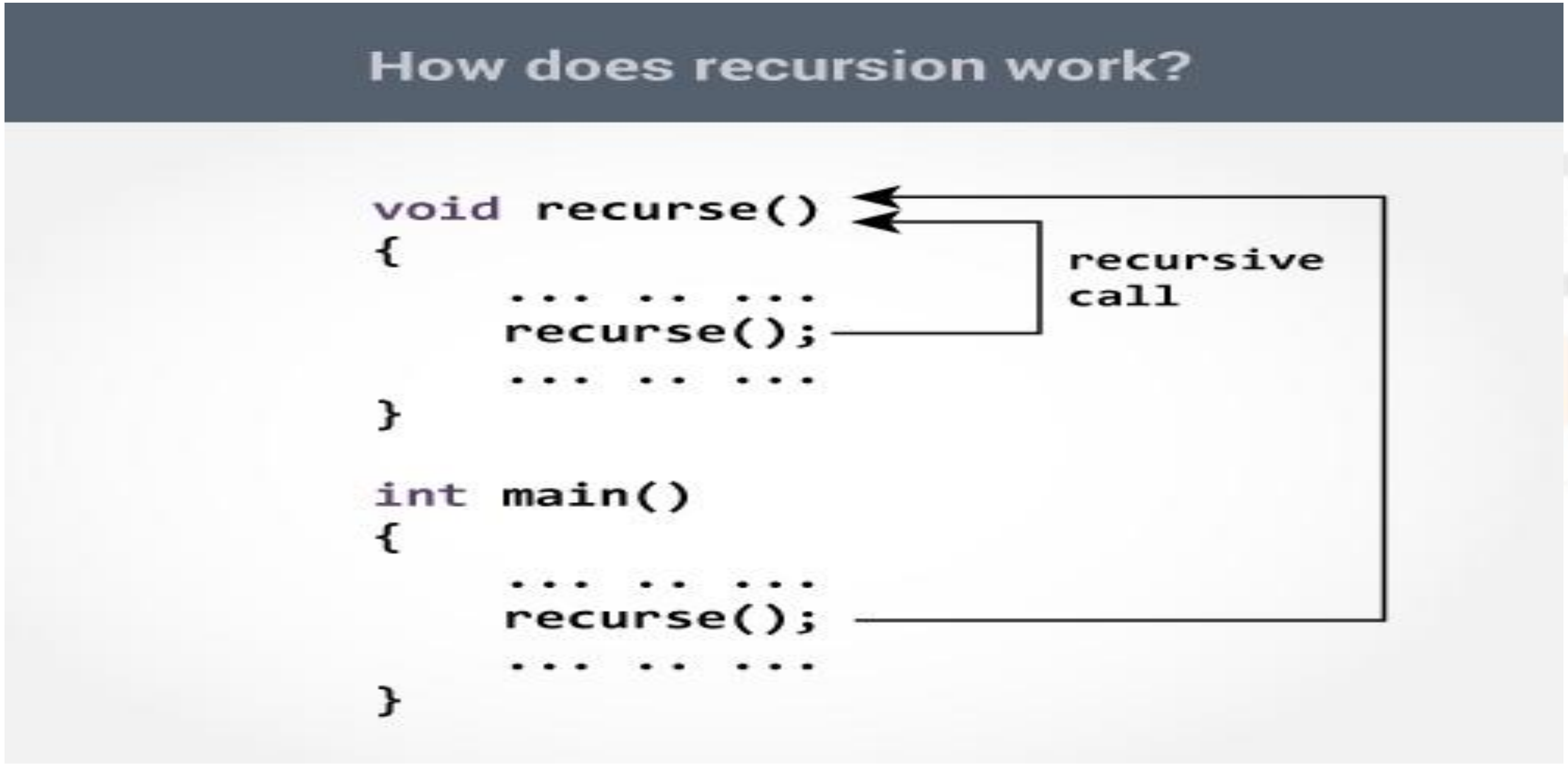


Figure : Working of recursion

Factorial Calculation (Contd….)

Example: Recursion

Let us take example of calculating factorial of a number using recursive approach. Code is implemented using C Language

```
#include <stdio.h>
void main()
{
   int m, res;

   printf("\nEnter the number :");
   scanf("%d",&m);

    res= fact(m);

    printf("\n Factorial of  %d is %d", m,res);  }
```

```
int fact( int n)
{
       if (n <=1)
             return 1;
       else
             return m * fact(n-1);
   }
```

Factorial Calculation (Contd….)

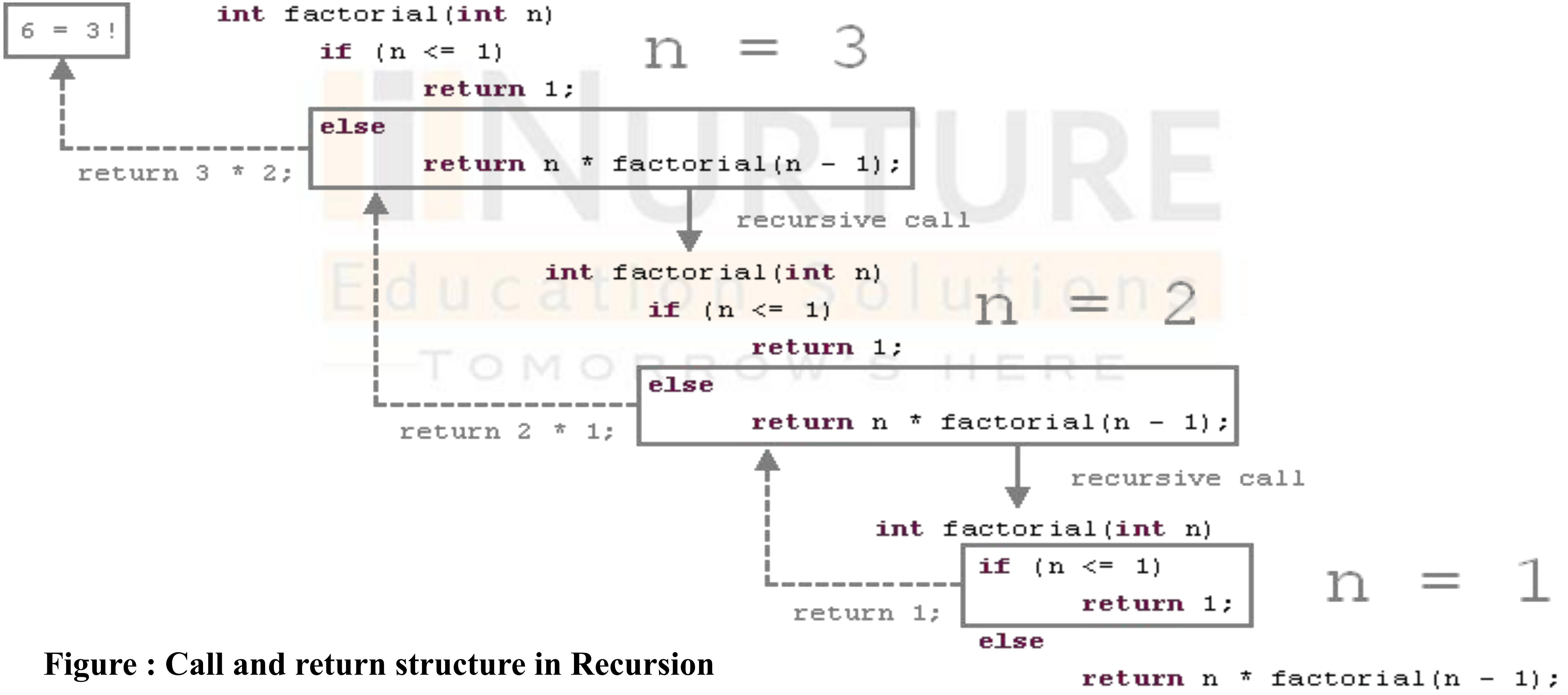


Figure : Call and return structure in Recursion

Factorial Calculation (Contd….)

Stack is used for internal implementation of Recursion

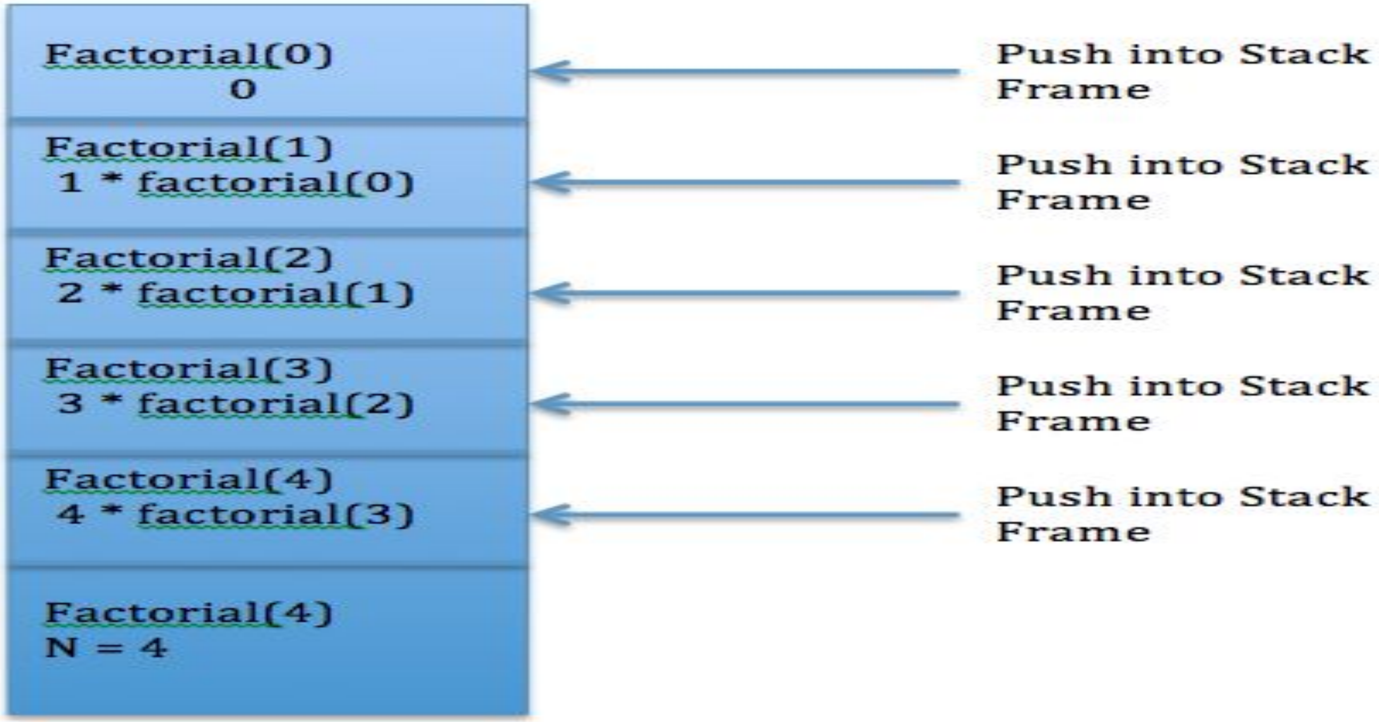


Figure : Use of stack internally to implement Recursion

Infix to postfix Transformation

Conversion of Infix to Postfix Expression:

- Convert the infix form to postfix using a stack to store operators and then pop them in correct order of precedence.
- Evaluate the postfix expression by using a stack to store operands and then pop them when an operator is reached.
- Scan through an expression, getting one token at a time.

Infix to postfix Transformation (Contd….)

Use a loop to read the tokens one by one from a vector infixVect of tokens (strings) representing an infix expression. For each token do the following in the loop:

- When the token is an operand
    - Add it to the end of the vector postfixVect of token (strings) that is used to store the corresponding postfix expression.
- When the token is a left parenthesis "("
    - Push_back the token x to the end of the vector stackVect of token (strings) that simulates a stack.

Infix to postfix Transformation (Contd….)

- When the token is a right parenthesis ")"
  - Repeatedly pop_back a token y from stackVect and push_back that token y to postfixVect until "(" is encountered in stackVect. Then pop_back "(" from stackVect.
  - If stackVect is empty before finding a "(", that expression is not a valid expression.
- When the token x is an operator
  - Write a loop that checks the following conditions:

1. The stack stackVect is not empty
2. The token y currently in the end of stackVect is an operator. In other words, it is not not a lef parenthesis "(" .
3. y is an operator of higher or equal precedence than that of x

Infix to postfix Transformation (Contd….)

As long as all the three conditions above are true, in the loop above repeatedly do the following in the body of the loop :

1. Call push_back to store a copy of the token y into postfixVect
2. Call pop_back to remove the token y from stackVect

Note: The loop above will stops as soon as any of the three conditions is not true. After the loop, push_back the token x into stackVect.

- After the loop (in the previous slide) has processes all the tokens in infixVect and stop.
- Use another loop to repeatedly do the following as long as the stack vector stackVect is not empty yet:

Infix to postfix Transformation (Contd….)

- Call push_back to store a copy of the token on the top of the stack vector stackVect into postfixVect.
- Call pop_back to remove the top token y from the stack vector.

Infix to postfix Transformation (Contd….)

Example: Conversion Infix to Postfix Notation using Stack

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	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 +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Table 2: Conversion Infix to Postfix Notation Example

Evaluation of Arithmetic Expressions

The basic algorithm for handling arithmetic expressions without parentheses makes use of a data structure called a stack.

Polish notation:

Polish notation is a symbolic logic invented by Polish mathematician Jan Lukasiewicz.

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Evaluation of Arithmetic Expressions (Contd….)

Parsing Expression:

- Precedence
- Associativity
- Postfix Evaluation Algorithm using stack

Evaluation of Arithmetic Expressions (Contd….)

Parsing Expression:

Postfix Evaluation Algorithm using stack

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

Towers of Hanoi

The Tower of Hanoi is a mathematical game or puzzle. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

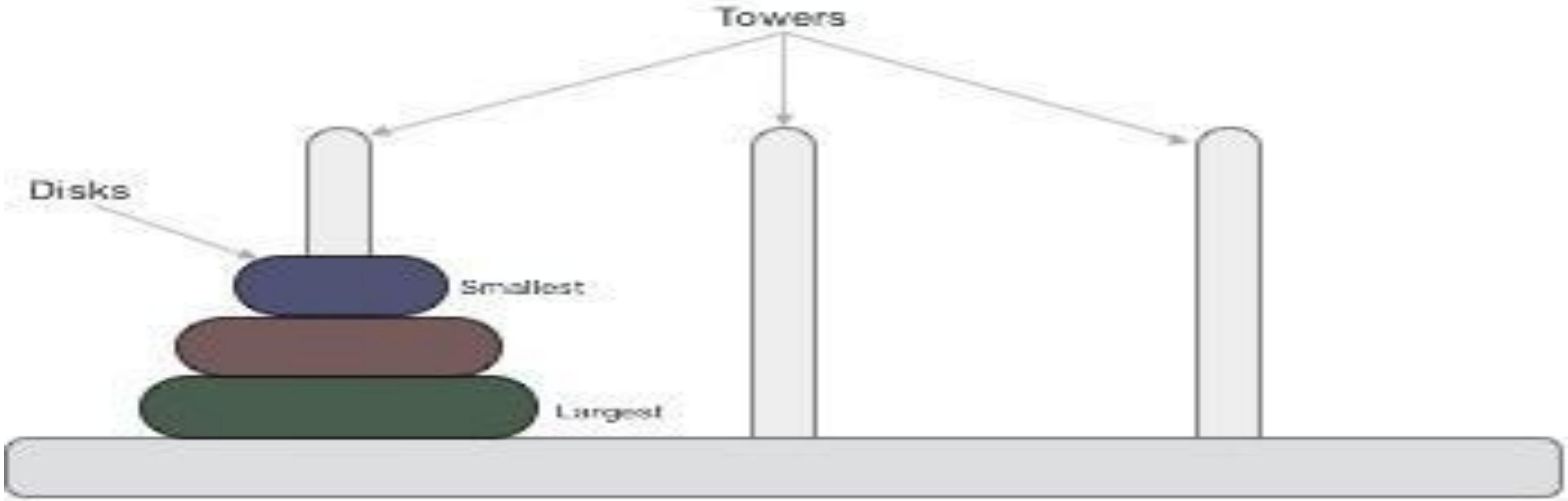


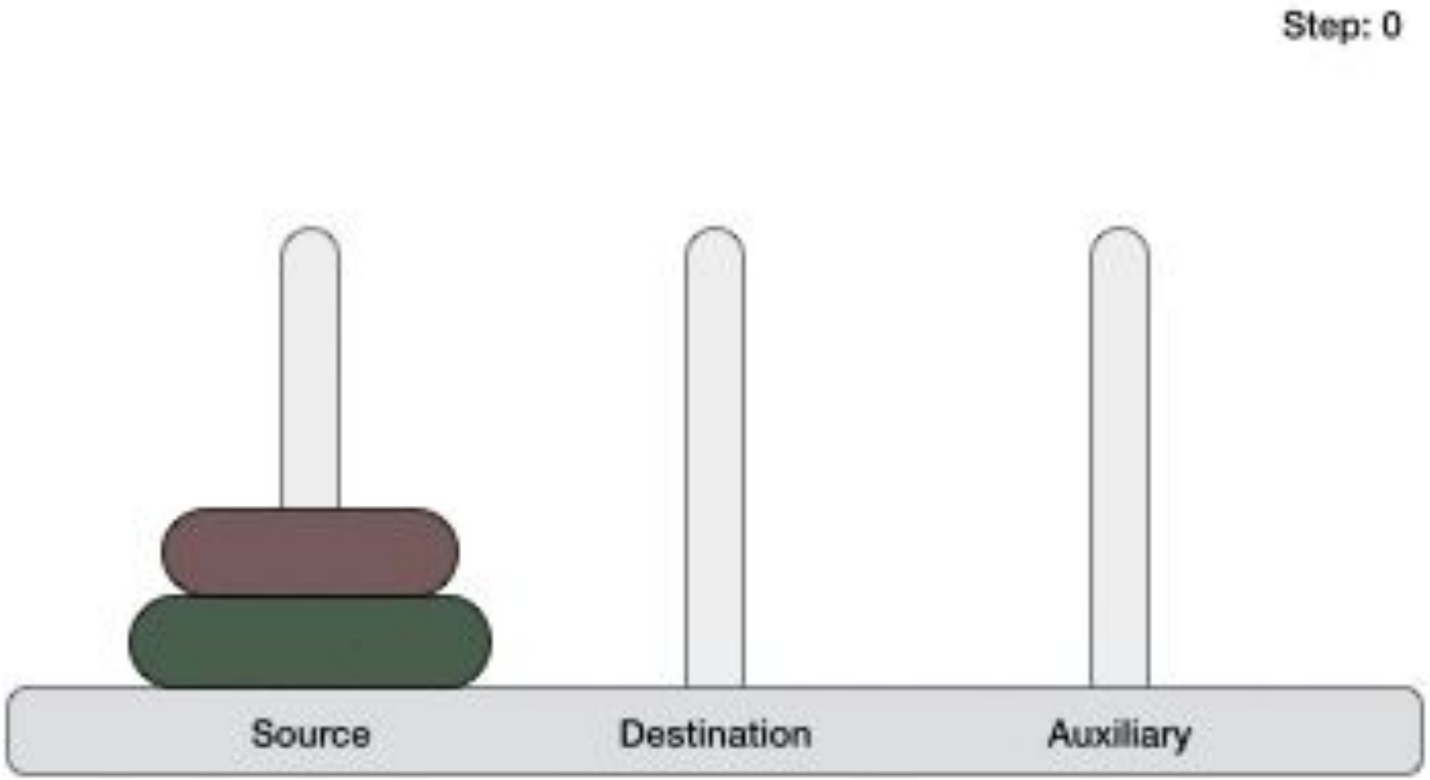
Figure : Schematic representation of Tower of Hanoi

Towers of Hanoi (Contd….)

There are three towers:

- 64 gold disks, with decreasing sizes, placed on the first tower.
- You need to move all of the disks from the first tower to the last tower.
- Larger disks cannot be placed on top of smaller disks.
- The third tower can be used to temporarily hold disks.
- The disks must be moved within one week. Assume one disk can be moved in 1 second. Is this possible?
- To create an algorithm to solve this problem, it is convenient to generalize the problem to the "N-disk" problem, where in our case N = 64.

Towers of Hanoi (Contd….)



Towers of Hanoi (Contd….)

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

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

Towers of Hanoi (Contd….)

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
```

Summary

- Stacks are Last-In-First-Out (LIFO) data structures in which the most recent element inserted in the stack is the first one to be removed.
- The stack can be implemented using an array by creating a stack pointer variable for keeping track of top position.
- Push () and pop () are the primary operations possible on stacks for insertion and deletion of elements along with some support functions.
- Polish notation which is also called as prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra.
- Infix notation is the most common and simplest notation in which an operator is placed between two operands.
- In prefix notation, operator precedes operands and in postfix operands precedes the operator.

Self Assessment Question

1. Which operation removes an element from the top of the stack?

   a. Push

   b. Pop

   c. Peek

   d. Delete

Answer: b

Self Assessment Question

2. Which data structure is typically used to evaluate arithmetic expressions?

   a. Queue
   b. Stack
   c. Linked List
   d. Tree

Answer: b

Self Assessment Question

3. Which stack application involves rearranging the elements in reverse order?

   a. Factorial Calculation

   b. Towers of Hanoi

   c. Reversing a list

   d. Infix to Postfix Transformation

Answer: c

Self Assessment Question

4. Which of the following problems can be solved using recursion?

   a. Factorial of a number
   b. Nth fibonacci number
   c. Length of a string
   d. All of the mentioned

Answer: d

Self Assessment Question

5. How can multiple stacks be implemented using a single array?

   a. Each stack can use a fixed-size section of the array.

   b. Stacks cannot be implemented using a single array.

   c. Only two stacks can be implemented together in a single array.

   d. Each stack can use the entire array as needed.

Answer: a

Assignment

1. Explain Stack in detail with the help of suitable example.
2. Write algorithms for PUSH, POP operations on stack.
3. Write a program in C Language to implement stack using arrays.
4. Write a program in C Language to implement stack using linked list.
5. Explain any one application of stack in detail.
6. Write an algorithm for converting Unparenthesized Infix expression into Postfix expression.
7. Convert the following Infix expression into Postfix expression:

   i) (A- B) * x + y / (F – C * E) + D

   ii) A* (b + c) + (b/d) * a + z* U

Document Link

Topic	URL	Notes
The stack Abstract Data Type	CSCI 2170 Lab 11 - Introduction to Abstract Data Types & Stacks (mtsu.edu)	This link contains the Stack Abstract Data Type
Representation of Stack	Stacks Representation and Operations - Programming and Data Structures - Computer Science Engineering (CSE) PDF Download (edurev.in)	This link contains the representation of stack
Operations of stack and applications	Stack and its basic Operations (afteracademy.com)	This link contains the operations of stack, and its applications

Video Link

Topic	URL	Notes
Representation of stack as an ADT	https://www.youtube.com/watch?v=XSdXSmwb550	This link contains the representation of stack as an ADT
Representation of Stack	https://www.youtube.com/watch?v=Zo6ykuemNLc	This link contains the content of how to represent the stack
Infix to postfix conversion	3.6 Infix to Postfix using Stack | Data Structures Tutorials - YouTube	This link contains the conversion from infix to postfix

E- Book Link

Topic	URL	Notes
Data Structures	https://mu.ac.in/wp-content/uploads/2021/05/Data-Structure-Final-.pdf	This link contains introduction to Data structures and algorithms
Algorithms and its notations	https://www.audisankara.ac.in/has/pdf/DATA%20STRUCTURE.pdf	This link contains the information about algorithms.