Data Structure
Algorithms & Applications

CT-159

Prepared by
Muhammad Kamran

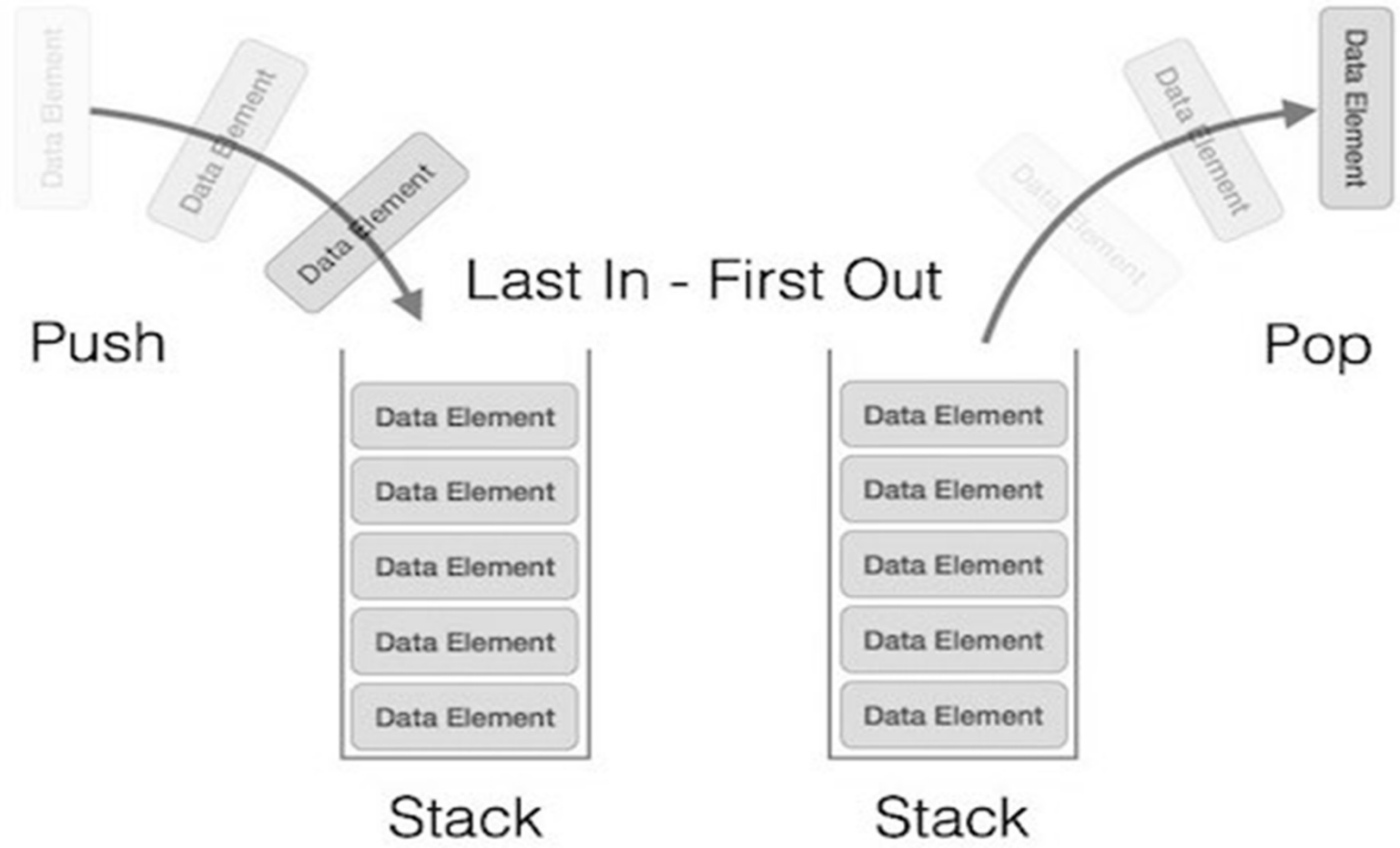
Stack (in real world)

- A real-world stack examples are: place or remove a card or plate from the top of the stack only.

Stack

- Stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- LIFO data structure. LIFO stands for Last-in-first-out.
- **PUSH** operation / **POP** operation.

Stack Representation



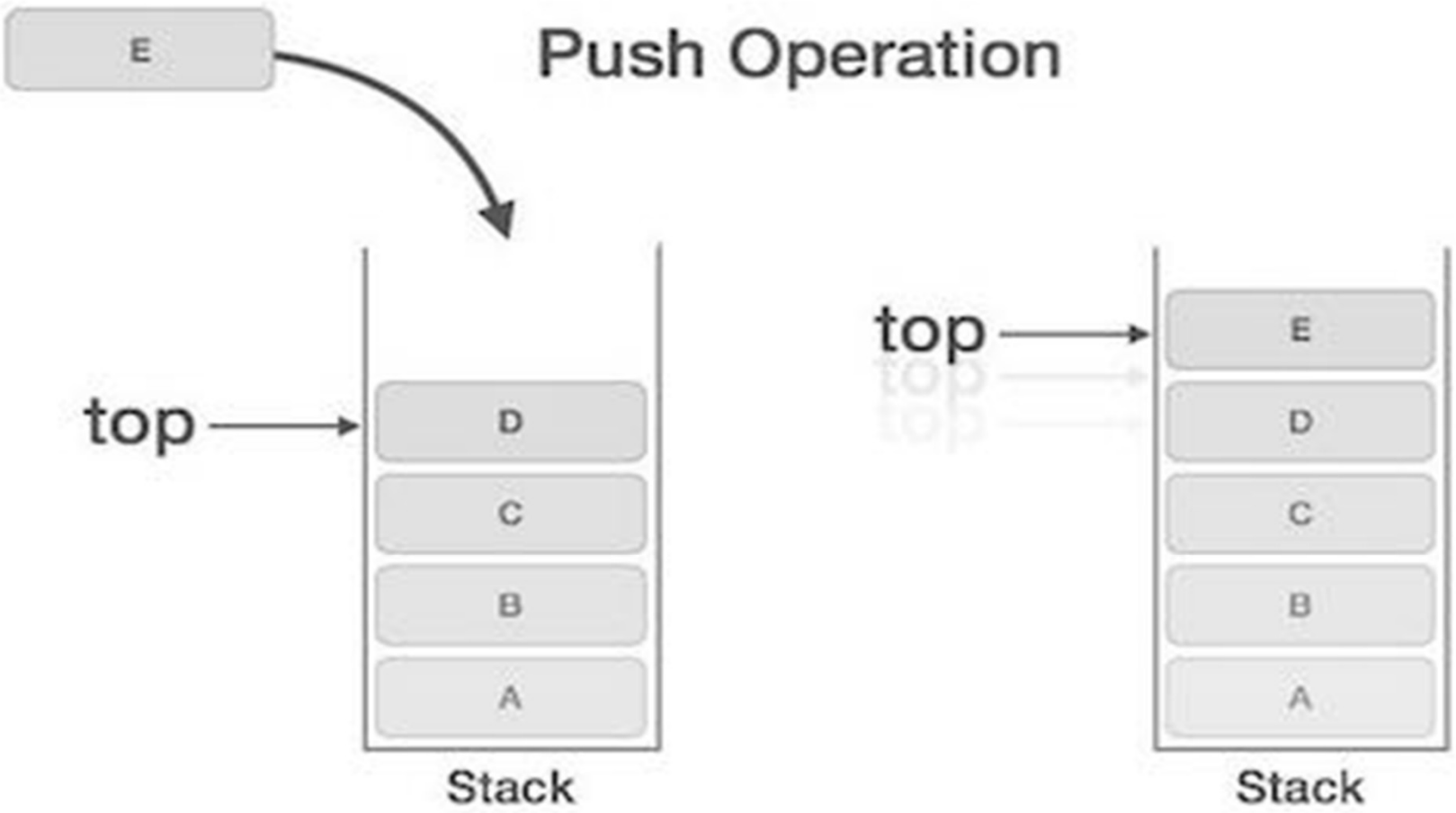
Stack (basic operations)

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.
- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty

Stack (Push Operation)

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.

Push



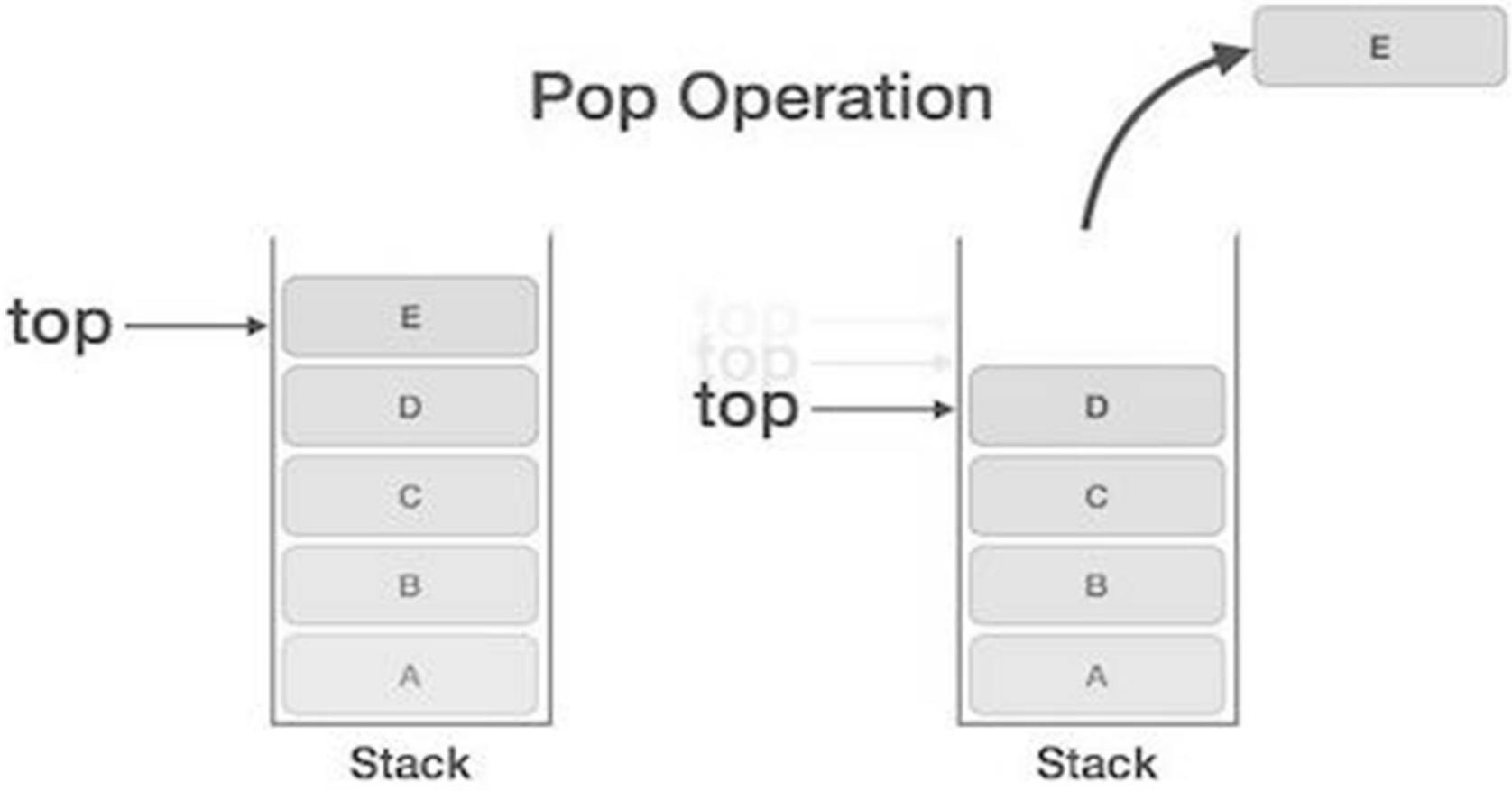
Pseudo code for PUSH Operation

```
begin procedure push: stack, data
        if stack is full
                return null
        endif
        top ← top + 1
        stack[top] ← data
end procedure
```

Stack (Pop Operation)

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.

Pop



Pseudo code for POP Operation

```
begin procedure pop: stack
        if stack is empty
                return null
        endif
        data ← stack[top]
        top ← top - 1
        return data
end procedure
```

Stack Implementation

- **push():** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
- **pop():** Return the top element of the Stack i.e simply delete the first element from the linked list.
- **peek():** Return the top element.
- **display():** Print all elements in Stack.

Stack Implementation

Push Operation

- *Initialize a node*
- *Update the value of that node by data i.e.* ***node->data = data***
- *Now link this node to the top of the linked list*
- *And update top pointer to the current node*

Stack Implementation

Pop Operation

- *First Check whether there is any node present in the linked list or not, if not then return*
- *Otherwise make pointer let say **temp** to the top node and move forward the top node by 1 step*
- *Now free this temp node*

Stack Implementation

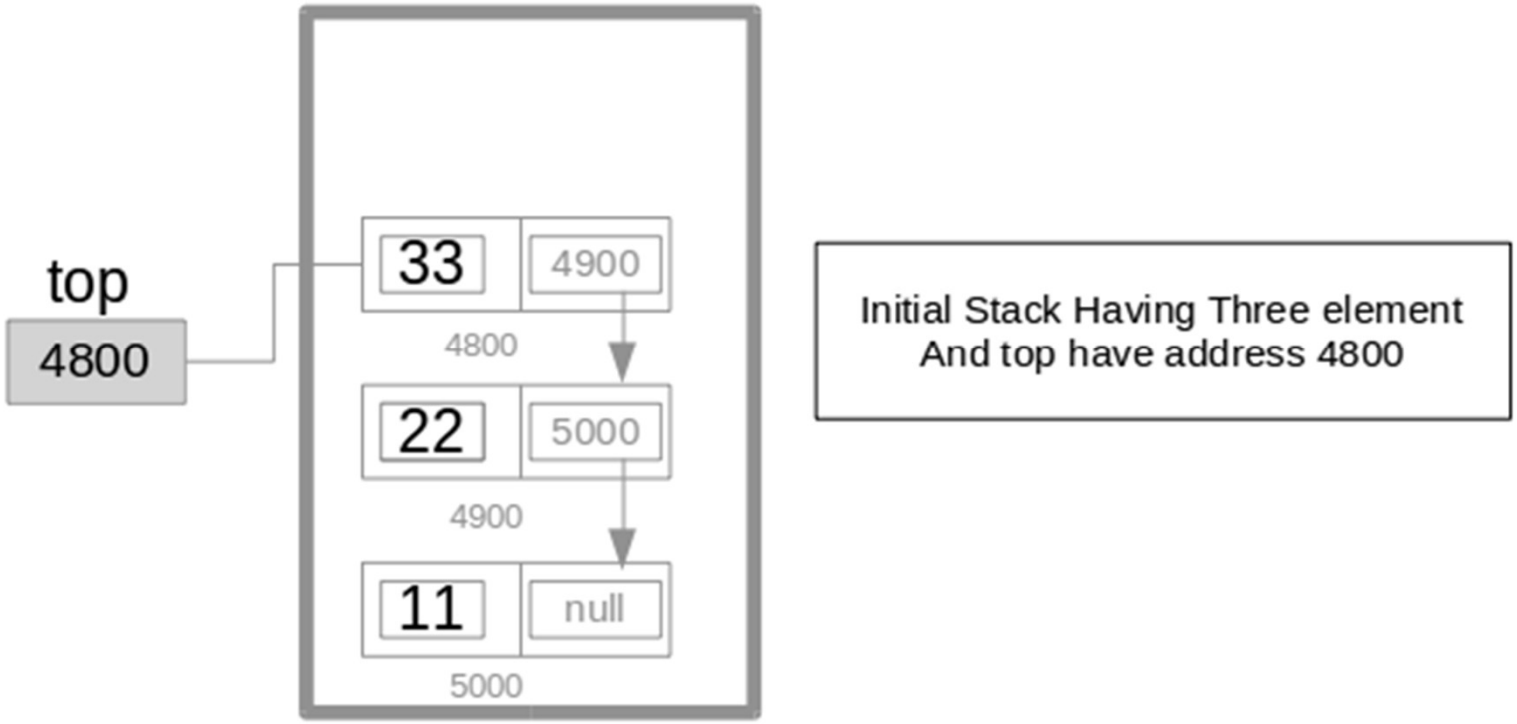
Peek Operation

- *Check if there is any node present or not, if not then return.*
- *Otherwise return the value of top node of the linked list*

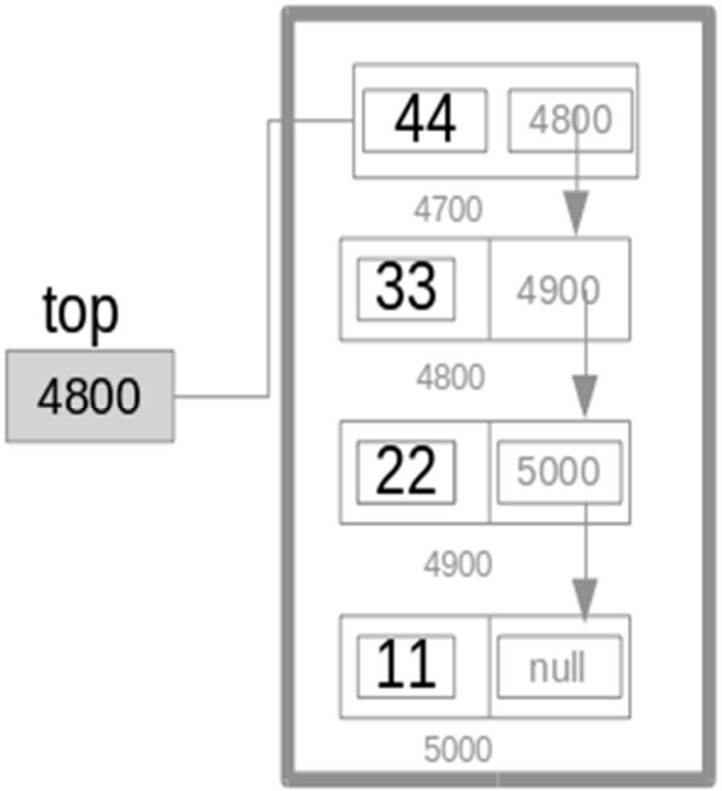
Display Operation

- *Take a **temp** node and initialize it with top pointer*
- *Now start traversing temp till it encounters NULL*
- *Simultaneously print the value of the temp node*

Stack Implementation

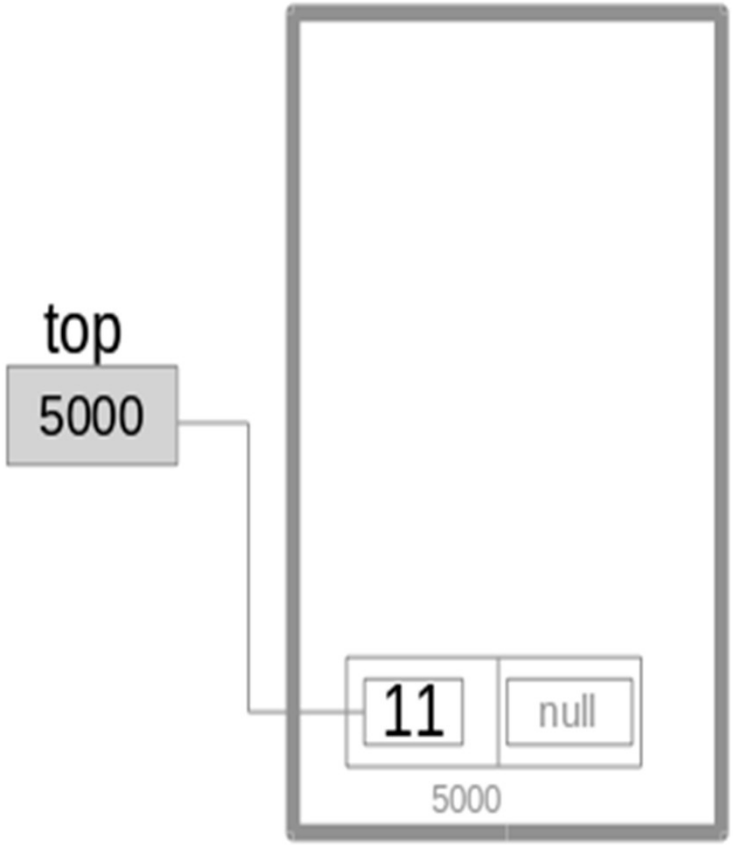


Stack Implementation



First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top

Stack Implementation




```
Pop three  element
      temp = top;
       top = top->link;
       temp->link = NULL;
       free(temp);
```

Application of Stack ??

- Function calls and recursion.
- Undo/Redo operations.
- Expression evaluation.
- Browser history.
- Balanced Parentheses.
- Backtracking Algorithms.

Notation

- Infix Notation (Human Readable)

  e.g. **(a + b)** $*$ **c**

- Prefix (Polish) Notation

  e.g. $*$ **+ a b c**

- Postfix (Reverse-Polish) Notation

  e.g. **a b + c** $*$

Postfix Evaluation Algorithm

- **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

Postfix Evaluation

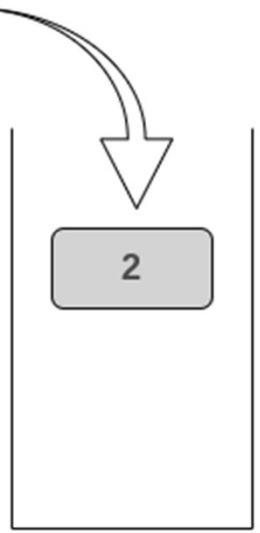
Input**: **str = "2 3 1 * + 9 -"

***Output**: -4*

***Explanation:** If the expression is converted into an infix expression:*

*it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.*

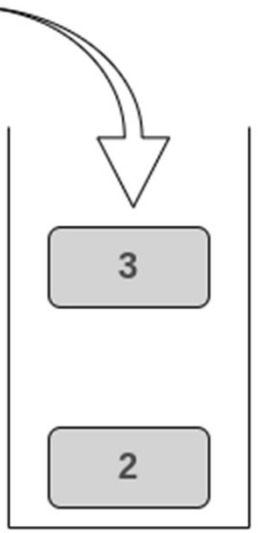
Add 2 in the stack



2 is an operand. Push it in stack

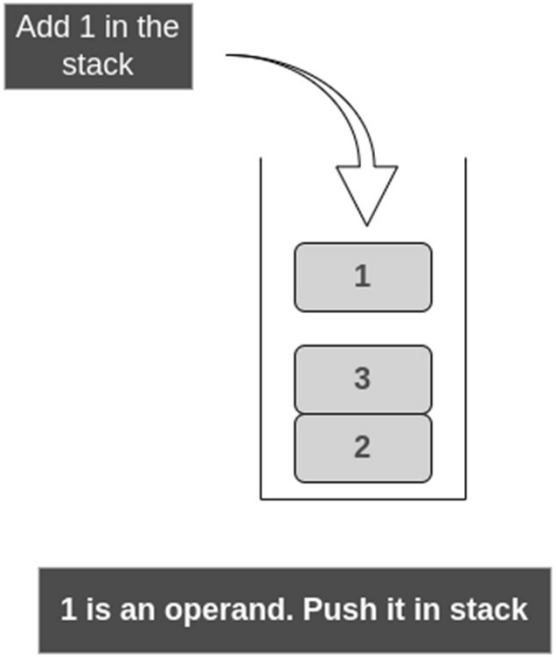
str = “3 1 * + 9 -“

Add 3 in the stack

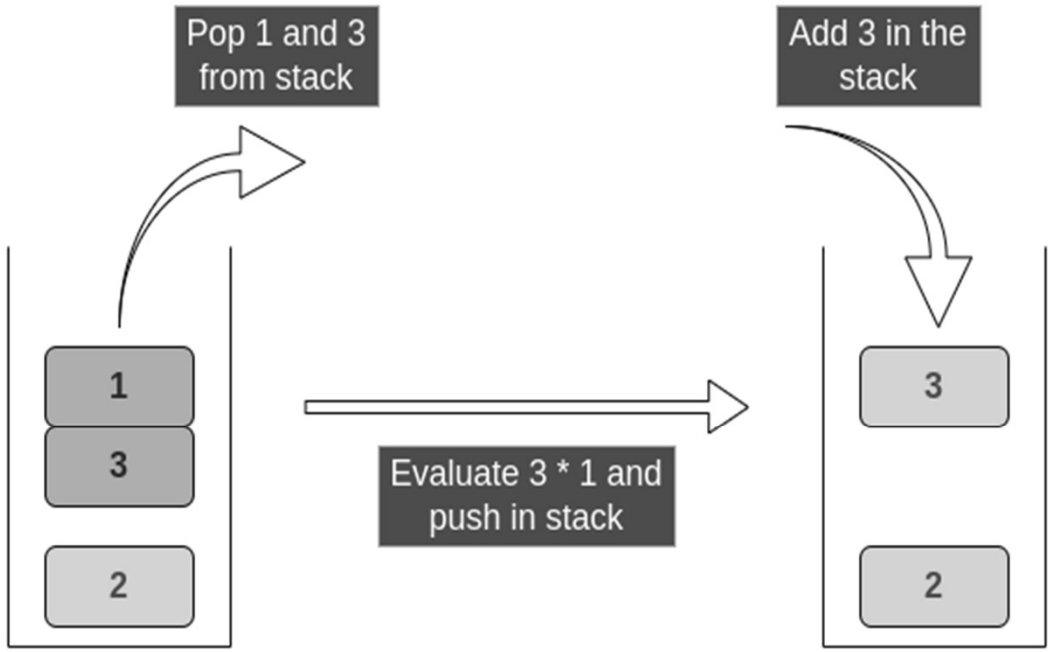


3 is an operand. Push it in stack

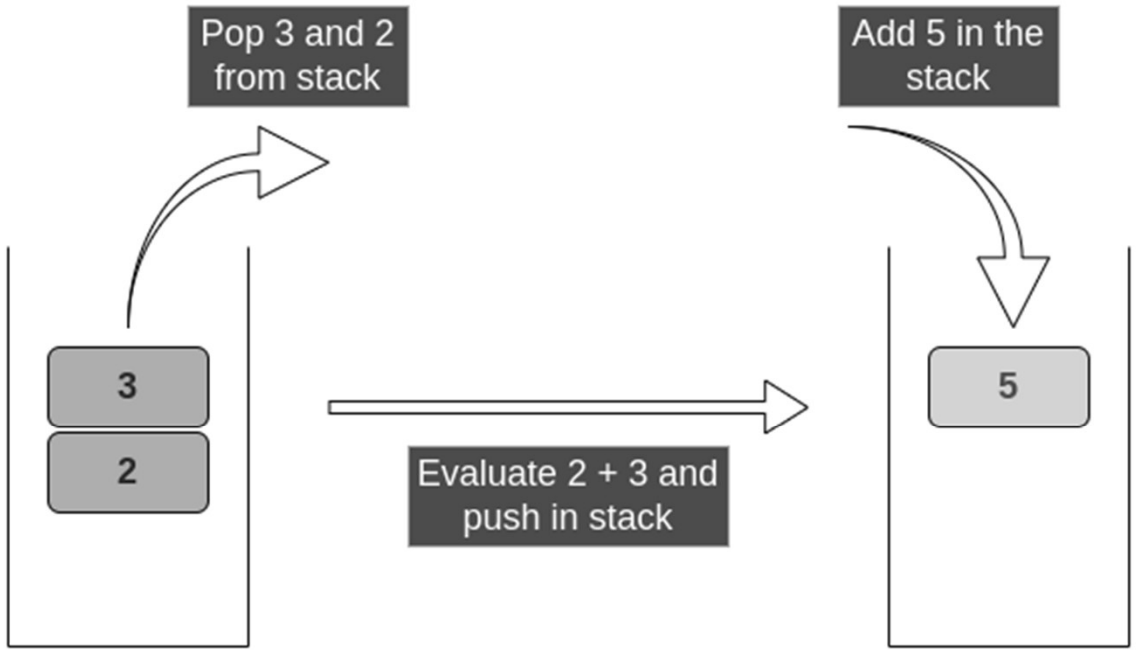
str = “1 * + 9 -“


Add 1 in the stack
1
3
2
1 is an operand. Push it in stack

str = “* + 9 -“

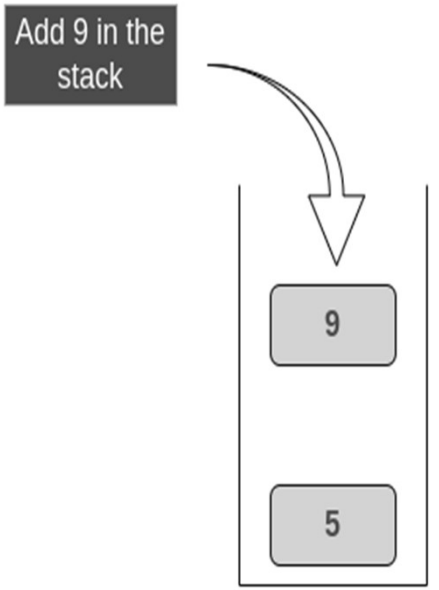

Pop 1 and 3 from stack
Add 3 in the stack
1
3
2
Evaluate 3 * 1 and push in stack
3
2
* is an operator. Evaluate it and push result in stack

str = “+ 9 -“



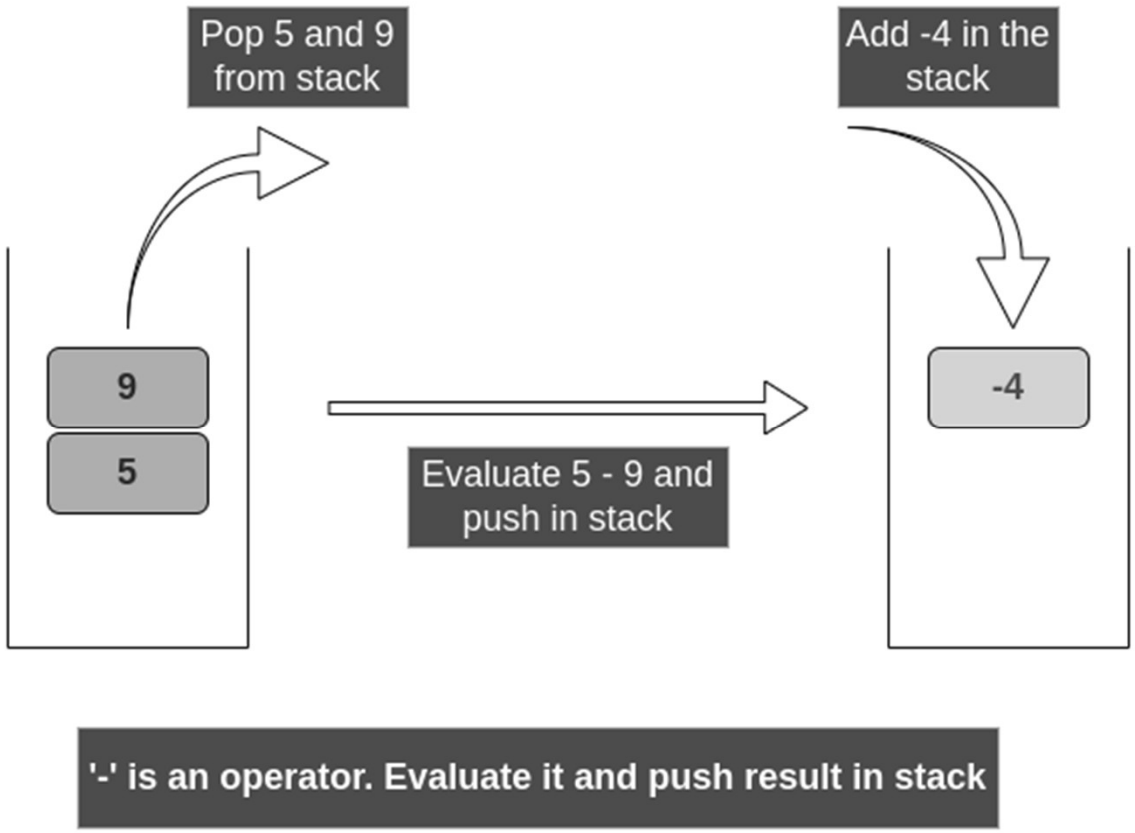
str = "9 -"

+ is an operator. Evaluate it and push result in stack



str = "-"

9 is an operand. Push it in stack


Pop 5 and 9 from stack
Add -4 in the stack
9
5
Evaluate 5 - 9 and push in stack
-4
'-' is an operator. Evaluate it and push result in stack

str = “”

Homework

Write an algorithm for **NEXT Greater Element** using **Stack**

*The **Next greater Element** for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.*

Example:

***Input**: array[] = [3 , 4 , 1 , 24]*
***Output**: 3 –> 4*
4 –> 24
1 –> 24
24 –> -1

Recursion

- Some computer programming languages allow a module or function to call itself. This technique is known as recursion.
- In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

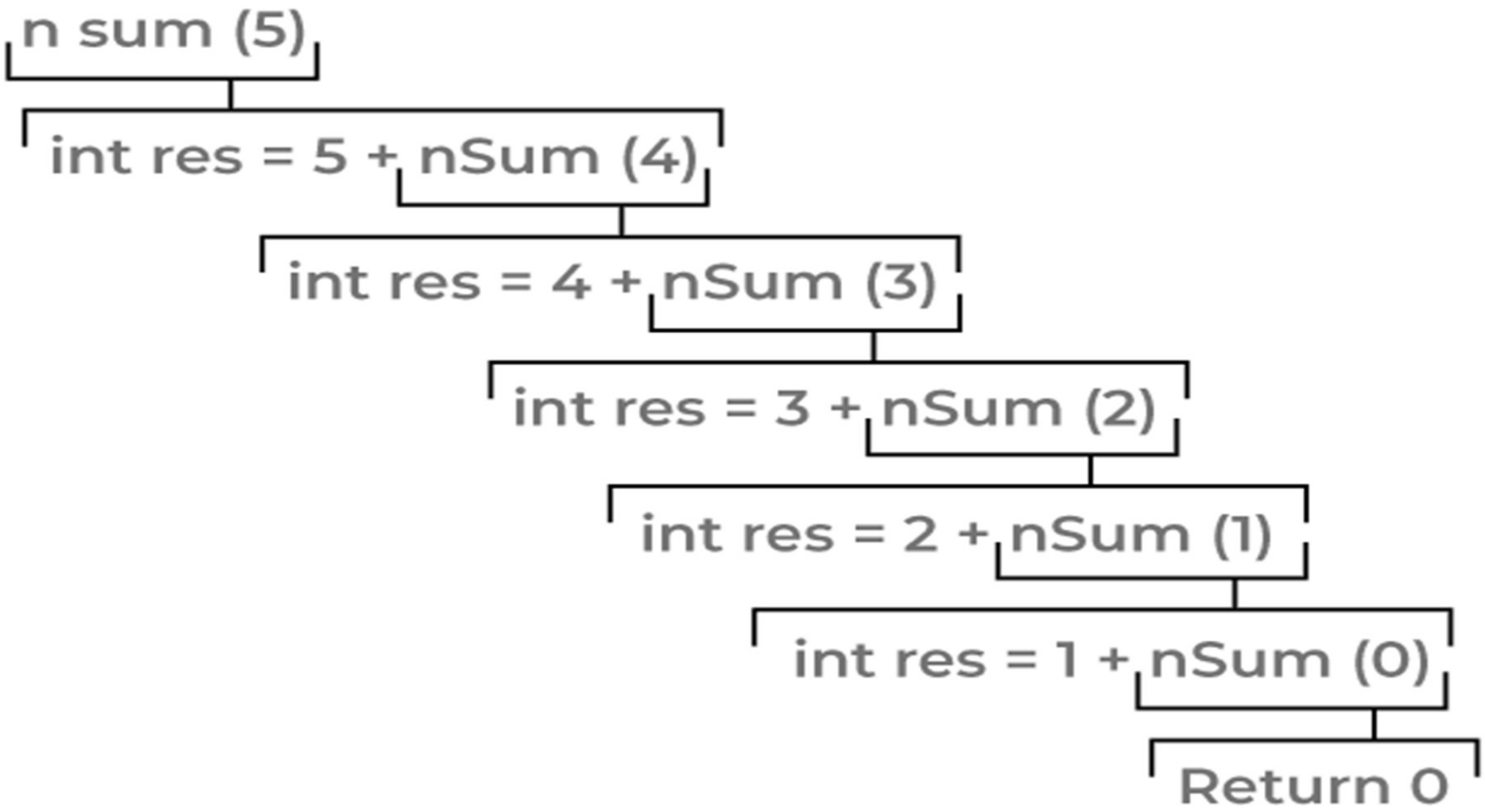
Recursion Example

```
#include <iostream>
using namespace std;


int nSum(int n)
{
if (n == 0) {
      return 0;
   }
   int res = n + nSum(n - 1);
   return res;
}
```

```
int main()
{
   int n = 5;
   int sum = nSum(n);
   cout << "Sum = " <<
sum;
   return 0;
}
```

Recursion Example



Types of Recursion

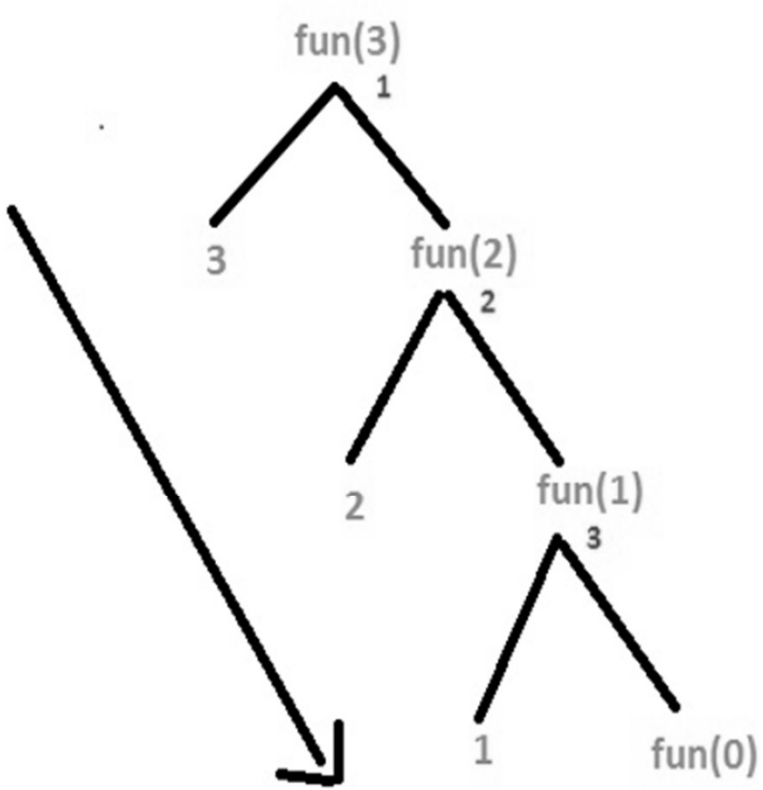
- Recursion are mainly of two types depending on whether a function calls itself from within itself or more than one function call one another mutually.
- Two types are :
    - Direct recursion
    - Indirect recursion.

Direct Recursion

1 - Tail Recursion: If a recursive function calling itself and that recursive call is the last statement in the function then it's known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

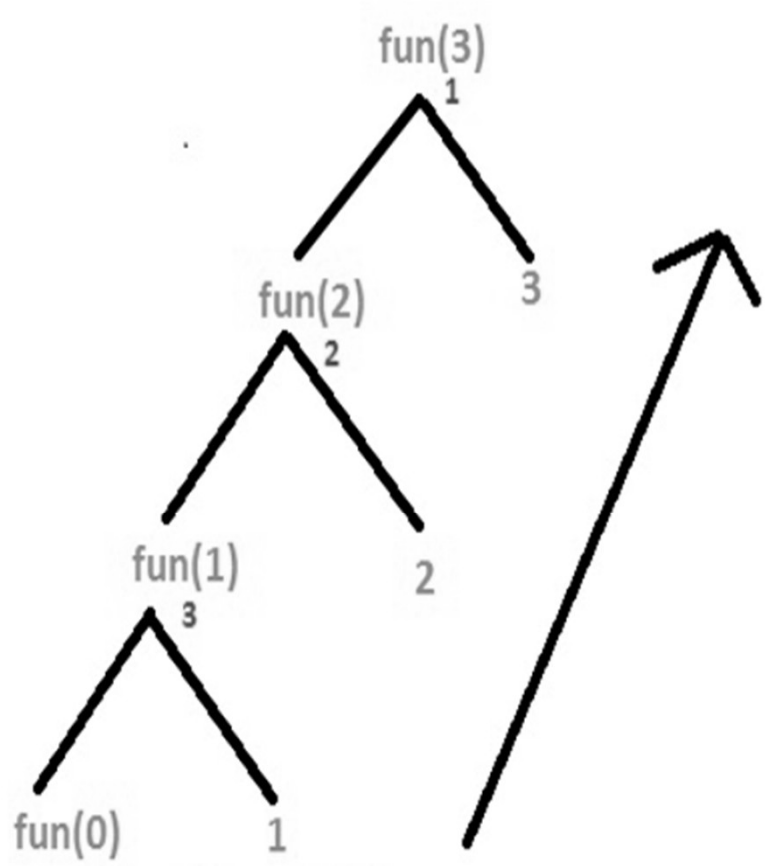
2 - Head Recursion: If a recursive function calling itself and that recursive call is the first statement in the function then it's known as Head Recursion. There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

Tail & Head Recursion



[Tail Recursion]

Output: 3 2 1



[Head Recursion]

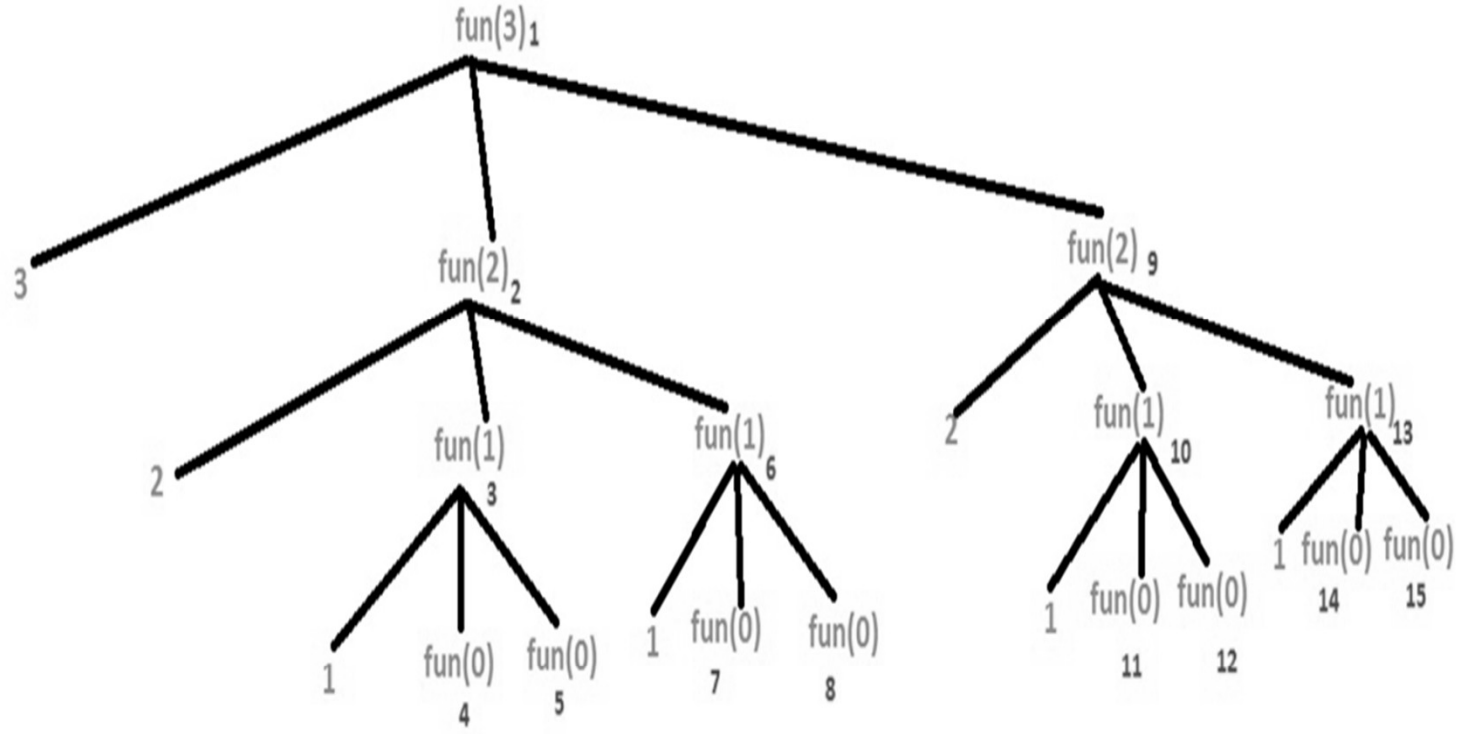
Output: 1 2 3

Direct Recursion

3 - Linear Recursion & Tree Recursion: If a recursive function calling itself for one time then it's known as Linear Recursion. If a recursive function calling itself for more than one time then it's known as Tree Recursion.

4 - Nested Recursion: In this recursion, a recursive function will pass the parameter as a recursive call. That means recursion inside recursion.

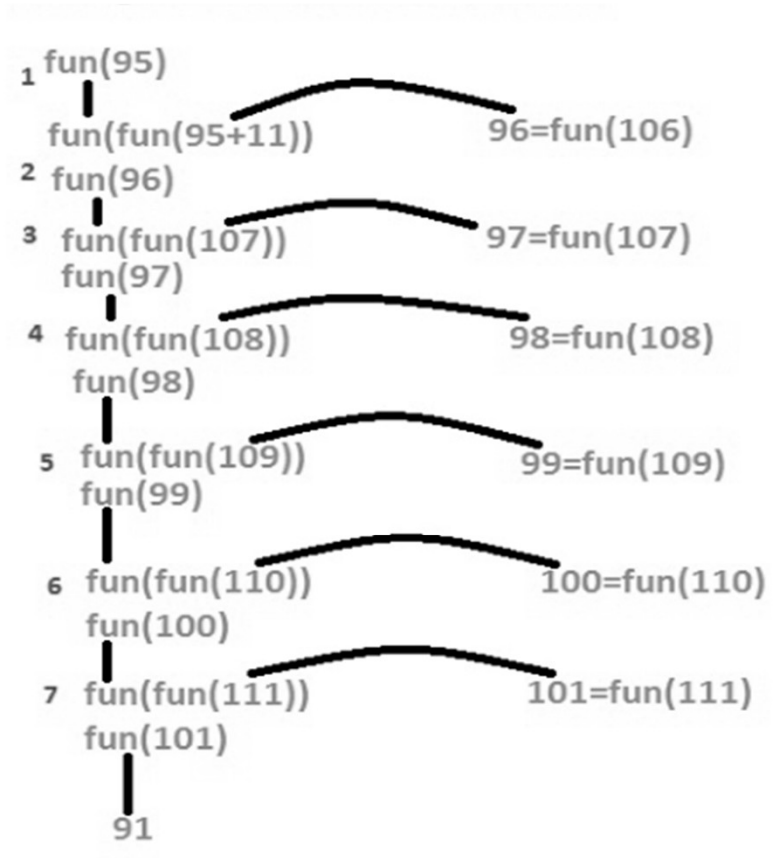
Tree Recursion



[Tree Recursion]

Output: 3 2 1 1 2 1 1

Nested Recursion



[Nested Recursion]

Output: 91

Indirect Recursion

In this recursion, there may be more than one functions and they are calling one another in a circular manner.

For example : in a programe

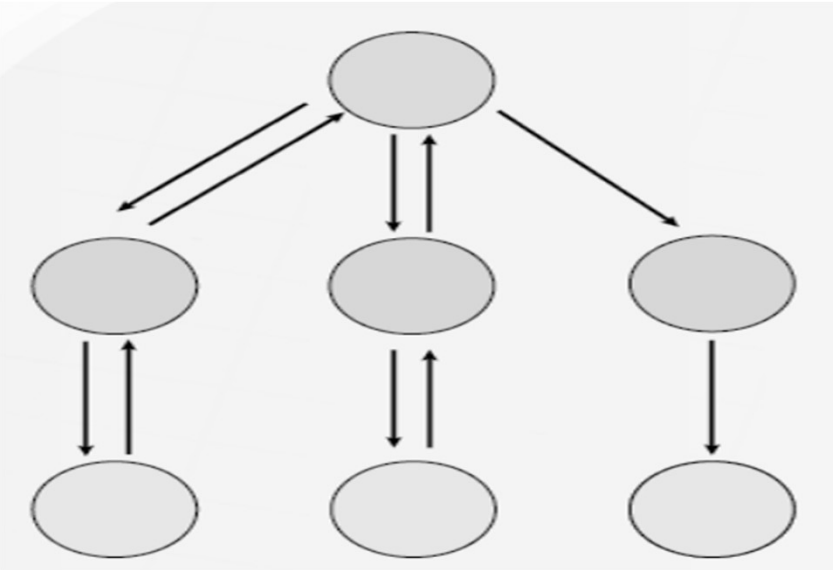
- Func (Z) calls Func (X),

  Func (X) calls Func (Y), and

  Func (Y) calls Func (Z)

Backtracking

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.



General outline of Backtracking algorithm

- Choose an initial solution.
- Explore all possible extensions of the current solution.
- If an extension leads to a solution, return that solution.
- If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
- Repeat steps 2-4 until all possible solutions have been explored.

Example of Backtracking Algorithm

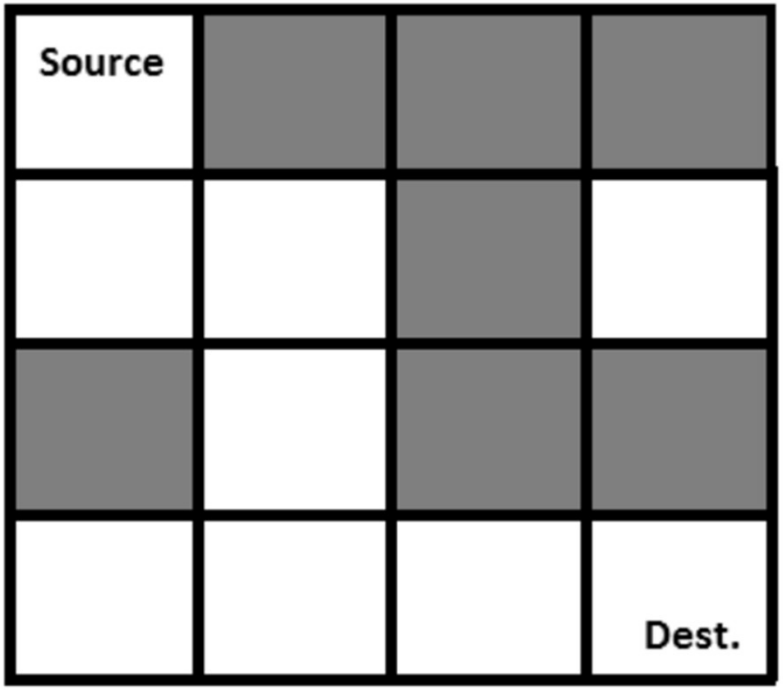
Finding the shortest path through a maze

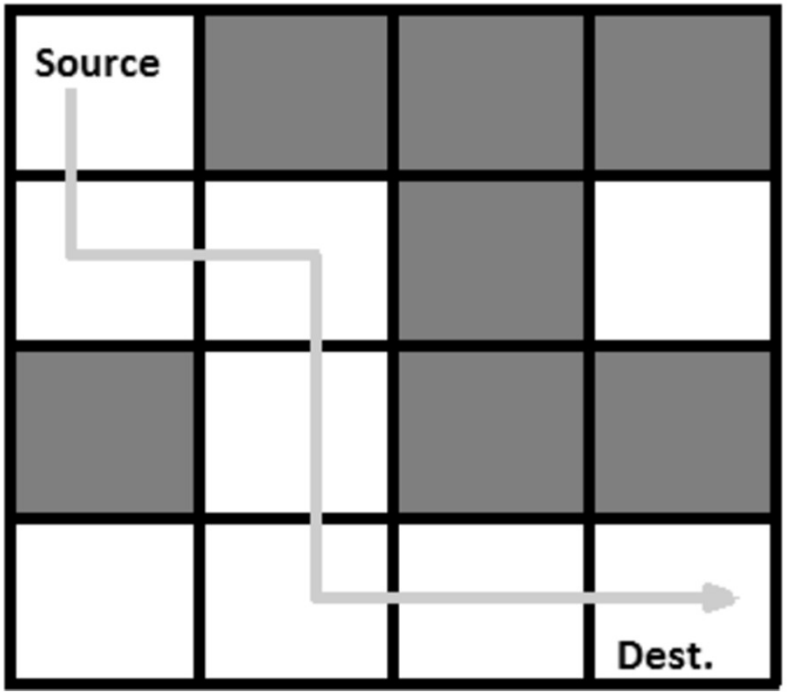
Input: A maze represented as a 2D array, where **0** represents an open space and **1** represents a wall.

Algorithm:

- Start at the starting point.
- For each of the four possible directions (up, down, left, right), try moving in that direction.
- If moving in that direction leads to the ending point, return the path taken.
- If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction.
- Repeat steps 2-4 until the ending point is reached or all possible paths have been explored.

Rat in a Maze





Step-by-step approach

- Create **isValid()** function to check if a cell at position (row, col) is inside the maze and unblocked.
- Create **findPath()** to get all valid paths:
  - **Base case:** If the current position is the bottom-right cell, add the current path to the result and return.
  - Mark the current cell as blocked.
  - Iterate through all possible directions.
    - Calculate the next position based on the current direction.
    - If the next position is valid (i.e, if **isValid()** return **true**), append the direction to the current path and recursively call the **findPath()** function for the next cell.
    - Backtrack by removing the last direction from the current path.
- Mark the current cell as unblocked before returning.

Thank You