Introduction to Stacks

Stack is an abstract data type with a bounded (predefined) capacity.

It is a simple data structure that allows adding and removing elements in a particular order.

Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.

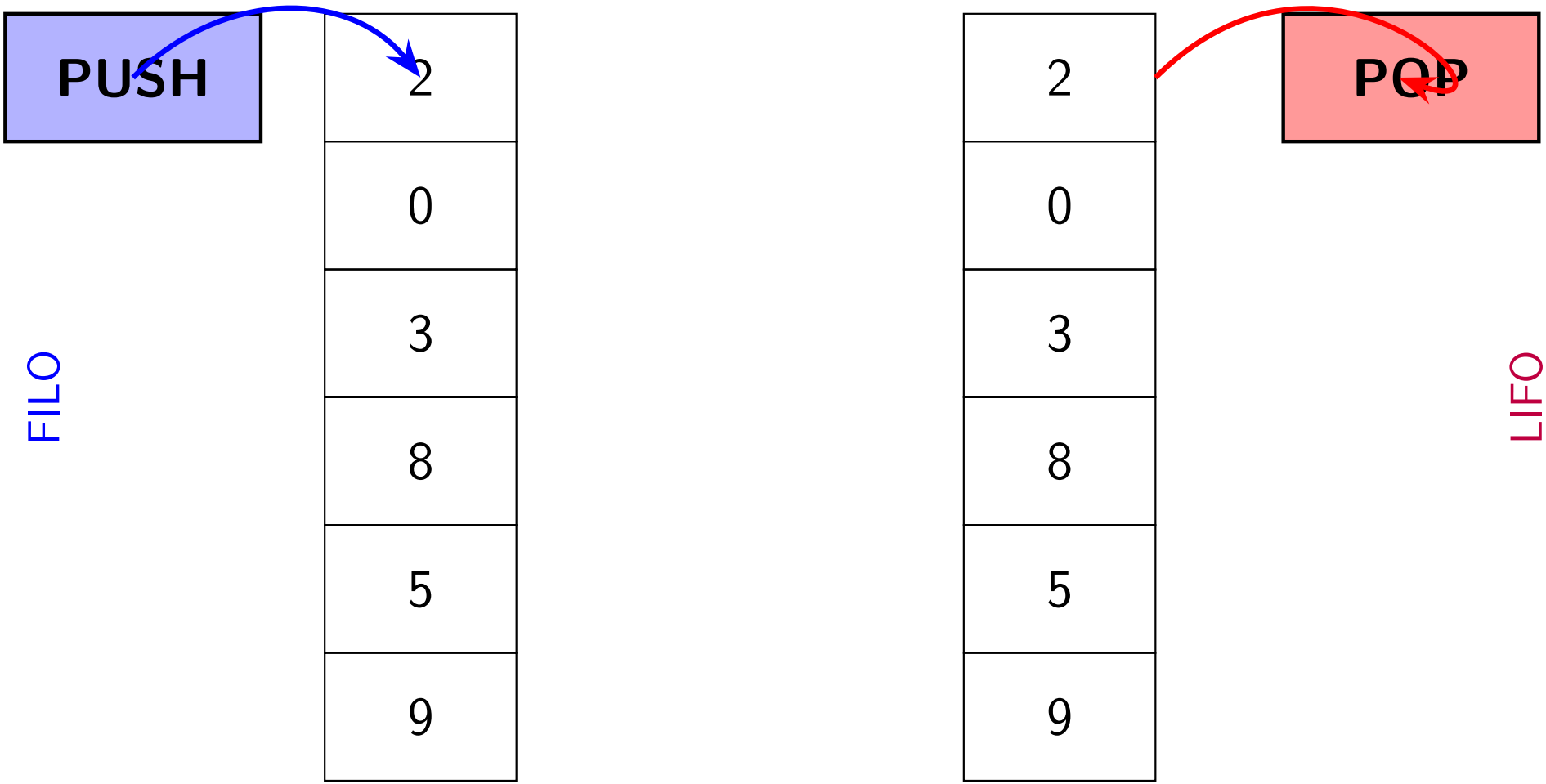
Real-life Examples of Stacks

- **Paper / Documents** — call stacks of paperwork, trays of forms (LIFO: last placed often accessed first).
- **Books / Plates** — stacked plates or books where the top item is easiest to remove.
- **Rings / Coins** — small circular objects stacked vertically for storage or display.

Paper stack

Stack of plates

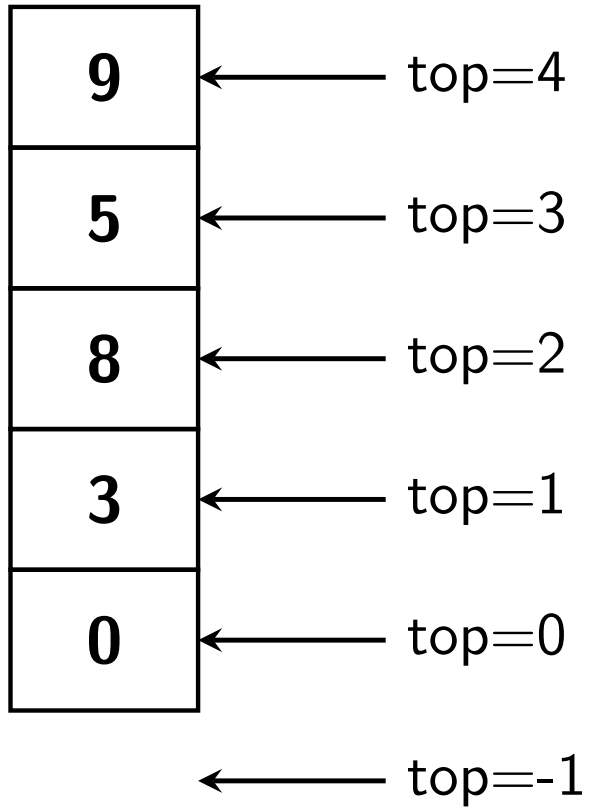
Stack – Quick Overview



The Top of the Stack

We always push an element to the top of the stack

We always pop the element from the top of the stack.



Stack operations

- **Push:** Place an item on top of the stack.
- **Pop:** Remove the item from the top.
- **Peek / Top:** Inspect the top item without removing it.
- These operations make stacks useful for undo mechanisms, parsing, recursion, and more.

Basic Functions

- **Push ( )** - Add an element to the stack (Insertion).
- **Pop ( )** - Remove an element from the stack (Deletion).
- **IsEmpty ( )** - Check if the stack contains no elements.
- **IsFull ( )** - Check if the stack has reached its maximum capacity.
- **Peek ( )** - Return the element at the top of the stack without removing it.

LIFO in Action: Where Stacks are used?

Compiler & Interpreter

- **Syntax Parsing:** Validating balanced parentheses, braces, and brackets (e.g., $(a + [b - c] * \{d\})$).
- **Expression Evaluation:** Converting Infix to Postfix/Prefix (Reverse Polish Notation) and evaluating Postfix expressions.
- **Function Calls:** Managing the *Call Stack* for function execution, local variables, and return addresses.

Web Navigation & Memory

- **Undo/Redo:** Storing state changes in text editors or graphic tools.
- **Backtracking Algorithms:** Used extensively in solving mazes, searching graphs (Depth First Search), and recursive problems (e.g., N-Queens).
- **Browser History:** Implementing the "Back" button functionality; the current page is `PUSH`ed when a new link is clicked.

Principle: Last In, First Out (LIFO)

Stack Implementation using Array (C)

1. Initialization and Global State

- A Stack uses a fixed-size array ('Size') for storage.
- The 'top' pointer tracks the index of the topmost element.

Listing: Global Stack Definitions

```
#define Size 10 // Maximum capacity of the stack
int stack[Size]; // The array to hold stack elements
int top = -1; // Initialize top to -1 (Empty Stack)
```

Stack Implementation (C) - PUSH

2. Inserting an Element (LIFO)

- Check for **Stack Overflow** (‘top == Size - 1‘).
- If space is available, increment ‘top‘ and add the element.

Listing: C Code for PUSH

```
// Global State: int top;
// Global Array: int stack[Size];

void push(int data) {
    if (top ==Size - 1) {
        printf("Stack Overflow! Cannot push %d.\n", data);
    } else {
        top++;
        stack[top] = data;
        printf("%d pushed to stack.\n", data);
    }
}
```

Stack Implementation (C) - POP

3. Removing the Top Element

- Check for **Stack Underflow** (‘top == -1‘).
- If not empty, retrieve the element at ‘stack[top]‘ and then decrement ‘top‘.

Listing: C Code for POP

```c
// Global State: int top;
// Global Array: int stack[Size];

int pop() {
    if (top == -1) {
        printf("Stack Underflow! Cannot pop.\n");
        return -1; // Return error code
    } else {
        int popped_item = stack[top];
        top--;
        return popped_item;
    }
}
```

Stack Implementation (C) - PEEK

4. Viewing the Top Element (Non-destructive)

- Check if the stack is **Empty** (‘top == -1‘).
- If not empty, return the element at ‘stack[top]‘ without changing ‘top‘.

```
// Global State: int top;
// Global Array: int stack[Size];

int peek() {
    if (top == -1) {
        printf("Stack is Empty.\n");
        return -1;
    } else {
        return stack[top]; // Returns the top element
    }
}
```

Stack Implementation (C) - DISPLAY

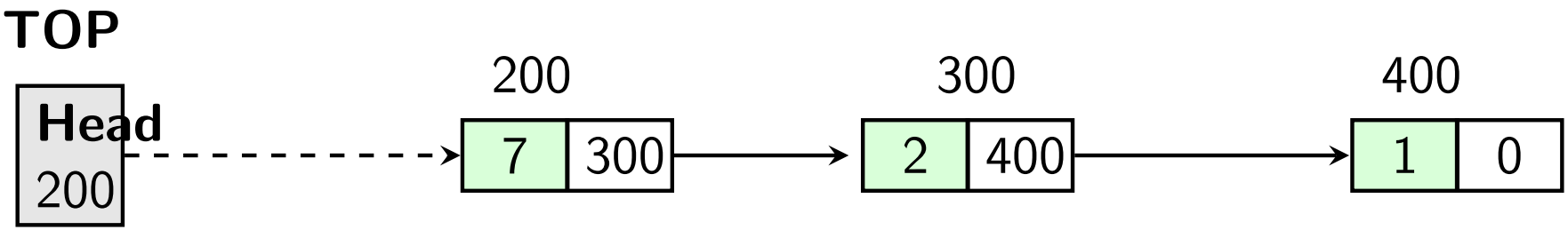
5. Printing All Stack Elements

- Checks if the stack is **Empty**.
- Iterates from ‘top‘ index down to 0 to print elements in LIFO order.

```c
void display() {
    if (top == -1) {
        printf("Stack is Empty.\n");
        return;
    }
    printf("Stack elements (Top to Bottom): \n");
    // Iterate from the current top down to the base (index
        0)
    for (int i = top; i >= 0; i--) {
        printf("| %d |\n", stack[i]);
    }
    printf("-----\n");
}
```

Linked List Representation

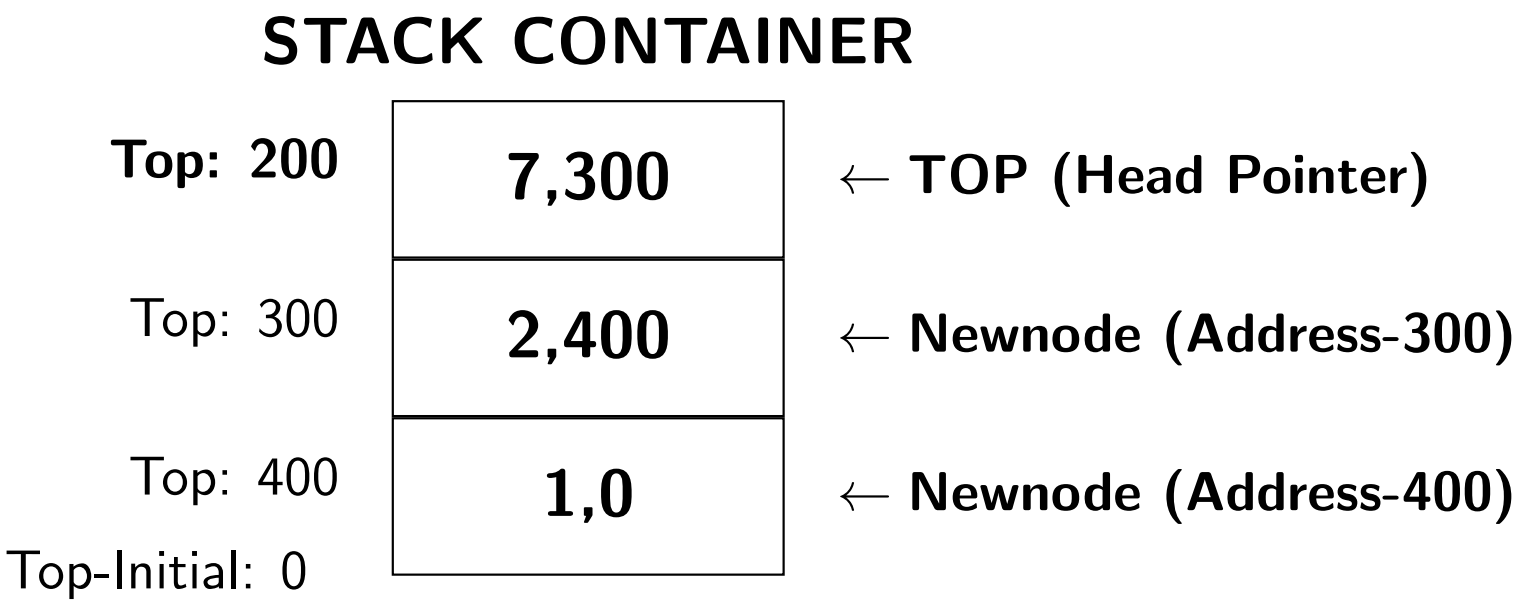
- **Fundamental Principle:** Stack follows the **LIFO** (Last-In, First-Out) principle.
- **Structure:** The linked list is a collection where the primary operations (`Insertion/Deletion`) are restricted to be performed exclusively on one end, known as the **TOP**.



Key Takeaway

The `Head` of the linked list is always treated as the **TOP** of the stack, enforcing the LIFO order for operations.

Stack Container Diagram (Linked List Implementation)



Achieving $O(1)$ Time Complexity

Efficiency (The Linked List Advantage)

- **Linked List Constraint:** Standard `Insertion` in a Linked List is often done at the tail (takes $O(N)$).
- **Stack Rule:** To achieve $O(1)$ complexity, both `PUSH` and `POP` **must** be performed at the beginning (Head) of the linked list.

Operation Complexity

- `PUSH(element)`: Insertion at the Head. Takes $O(1)$ (Constant Time).
- `POP()`: Deletion at the Head. Takes $O(1)$ (Constant Time).
- **Why $O(1)$?:** Only one pointer (`Head/Top`) needs to be modified, regardless of the list size.

Contrast with Array Implementation

A stack implemented using a dynamic array may require $O(N)$ time for expansion (reallocation and copying) when the array capacity is exceeded. The linked list guarantees $O(1)$ for `PUSH` and `POP` consistently.

Stack-Functions

- Each push adds a new node on top of the stack.
- The stack grows dynamically without overflow (until memory is full).
- Pop operation deletes the top node and moves the top pointer.

Push Function

Listing: Node Structure

```
struct Node {
    int data;
    struct Node *next;
};
struct Node *top = NULL;
```

Push Function

```
void push(int value) {

    struct Node *newNode = (struct Node *)malloc(sizeof(
        struct Node));

    if (newNode == NULL) {
        printf("Stack Overflow! \n");
        return;
    }

    // 2. Set new node's data
    newNode->data = value;

    // 3. Link new node to the current top
    newNode->next = top;

    // 4. Update top to the new node
    top = newNode;

    printf("Pushed %d to stack.\n", value);}
```

Display Function

Note: No need to check overflow condition.

```
void display()
{
    struct node *temp;
    temp = top;

    if (top == NULL)
    {
        printf("Empty");
    }
    else
    {
        while (temp != NULL)
        {
            printf("%d", temp->data);
            temp = temp->next;
        }
    }
}
```

Peek Function

Purpose: View the topmost element.

```
void peek()
{
    if (top == NULL)
    {
        printf("Stack empty");
    }
    else
    {
        printf("Top element is %d", top->data);
    }
}
```

Pop Function

Purpose: Remove the top element.

```
void pop()
{
    struct node *temp;

    if (top == NULL)
    {
        printf("Underflow");
    }
    else
    {
        temp = top;
        top = top->next;
        free(temp);
    }
}
```