HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INFORMATION TECHNOLOGY


# REPORT ON STACK

*Course Code: DASA230179*

**Supervisor:** MSc. Vu Dinh Bao

**Students:**

| | |
|---|---|
| Nguyen Hung Dung | Student ID: 22134002 |
| Tran Nhu Hoang | Student ID: 22134006 |
| Tran Nguyen Phuong Binh | Student ID: 24133006 |
| Pham Ngoc Phuc | Student ID: 22134010 |
| Vo Hong Quan | Student ID: 22134012 |

**Academic Year 2025-2026**

# Team Task Assignment

| No. | Name | Role | Assigned Tasks |
|---|---|---|---|
| 1 | Vo Hong Quan | Leader | Content development<br>Presentation<br>Slide design<br>Implement web demo |
| 2 | Nguyen Hung Dung | Member | Content development<br>Presentation |
| 3 | Tran Nhu Hoang | Member | Content development<br>Slide design |
| 4 | Tran Nguyan Phuong Binh | Member | Content development<br>Slide design |
| 5 | Pham Ngoc Phuc | Member | Content development<br>Slide design |

# ACKNOWLEDGMENTS

To complete this report, we have received invaluable support from many individuals and organizations. We would like to express our sincere gratitude to:

- **The Board of Directors of Ho Chi Minh City University of Technology and Education** and the faculty members of the Faculty of Information Technology, who have provided us with favorable and effective learning and research conditions.
- Our supervisor, **Mr. Vu Dinh Bao**, whose dedicated guidance played an important role throughout every stage of this project.

Although we have made our best efforts to conduct this report in a serious and thorough manner, shortcomings are inevitable. Therefore, we sincerely look forward to receiving comments and suggestions from our supervisor so that we can continue to improve our work in the future.

# ABSTRACT

This report focuses on the study of the stack data structure, an abstract data type operating under the LIFO (Last In, First Out) principle. The content presents fundamental concepts, typical operations, implementation methods using arrays and linked lists, as well as complexity analysis and performance evaluation through experiments. In addition, the report explores various practical applications of stacks such as recursive call management, expression conversion and evaluation, DFS and Backtracking algorithms, as well as software features like Undo/Redo, web browsing history, and expression validation. Moreover, stacks play an important role in advanced areas such as natural language processing, machine learning, and computer vision. The findings show that stacks are a foundational data structure with wide applicability in both theory and practice, while also opening directions for optimization and integration with other data structures to solve modern computational problems.

Keywords: **Stack, LIFO**.

# Contents

# INTRODUCTION

This chapter provides an overview of the **Stack**, including background, objectives, research scope, and methodology.

## 1.1 Background of the Topic

In the era of digital technology and Industry 4.0, programming has become a core skill that contributes to many fields such as finance, healthcare, education, transportation, and entertainment, since most modern system software and services require programming. However, in order to develop software applications such as Artificial Intelligence (AI), Big Data analytics, Blockchain, or the Internet of Things (IoT), learners first need solid technical skills and a strong foundation.

The most crucial foundation that every programmer must master is **Data Structures and Algorithms**. These are considered the "backbone"—the building blocks for creating complex and efficient software. The relationship between **Data Structures and Algorithms** is two-way. An algorithm, no matter how optimal, is inefficient, hard to scale, and resource-intensive if paired with an unsuitable data structure. Conversely, a well-designed data structure coupled with a poor algorithm may lead to incorrect results. This demonstrates that **Data Structures and Algorithms** are inseparably linked, complementing each other. Data structures enable fast data storage and retrieval, while algorithms are the sets of steps or rules to solve problems and optimize processes. Together, they provide systematic ways to organize, store, and manage data, thereby improving program performance. Especially in today's context, where massive amounts of data are generated every second, choosing appropriate **Data Structures and Algorithms** has become increasingly critical.

Furthermore, by studying deeper, programmers gain knowledge of fundamental Abstract Data Types (ADT) commonly used in program development, as well as their implementation and applications. ADTs describe possible operations (behavior) without specifying how they are implemented. Common ADTs include: List, **Stack**, Queue, Deque, Priority Queue, Set, Map/Dictionary, and Disjoint Set. These ADTs can be implemented using various data structures such as: dynamic arrays and linked lists (for List), heaps (for Priority Queue), hash

tables or balanced search trees like AVL/Red–Black/B-tree (for Map/Set), Union–Find/DSU (for Disjoint Set), or graphs with adjacency lists/matrices.

Among them, the **Stack** is a fundamental ADT that operates under the LIFO principle and can be implemented simply using arrays or linked lists. Despite its early introduction, the **Stack** remains a core element in modern algorithm design and systems (call stack, expression evaluation, DFS, undo/redo).

## 1.2 Objectives of the Report

This report provides a comprehensive, systematic, and scientific view of the characteristics, operating principles, and applications of the **Stack** in computer science and software development. Specifically, it aims to clarify fundamental concepts and basic operations, analyze performance and implementation methods, and demonstrate the wide applicability of the **Stack** through practical examples and multi-language source code illustrations. Through this, readers not only strengthen their theoretical knowledge but also develop the ability to apply the **Stack** as an effective tool to solve specific programming problems.

## 1.3 Research Scope

This report limits its research scope to the fundamental aspects of the **Stack** data structure, focusing on the LIFO principle and basic operations such as Push, Pop, Peek, and isEmpty. It examines two main implementation methods: arrays and linked lists. Selected applications are analyzed in typical fields such as Call Stack in computer systems, expression evaluation, DFS algorithms, and the Undo/Redo mechanism in user interfaces. Example code is implemented in five popular programming languages: C++, Java, Python, C#, and TypeScript, with the goal of demonstrating the universality of the **Stack** without delving into advanced optimization techniques or libraries. Additionally, the report does not extend to comparisons with other data structures beyond the basic Queue and does not explore hardware-level aspects.

## 1.4 Methodology

To ensure that this report is conducted scientifically, systematically, and meets its stated objectives, the group applied the following main research methods:

1. Literature review: collecting information and materials from reliable sources, focusing on studying textbooks and references on **Data Structures and Algorithms**.

2. Theoretical analysis and synthesis: comparing and evaluating information, identifying strengths and weaknesses, and drawing valuable conclusions. The theories studied were synthesized clearly, with a focus on the principles, operations, and applications of the **Stack**. The role of the **Stack** in specific applications (e.g., Call Stack, expression evaluation) was analyzed to explain why it is more suitable than other data structures.

3. Experimentation: simulating code in multiple programming languages to compare differences in syntax and approaches, while emphasizing the universality of the **Stack**.

4. Group work methodology: to ensure progress and quality, the team created a clear work plan, assigned tasks based on each member's strengths, divided the timeline into milestones, and strictly followed the schedule. Members frequently exchanged ideas, reviewed, and provided feedback on each other's work to maintain consistency and accuracy throughout the project.

# THEORICAL BACKGROUND

## 2.1 Stack

### 2.1.1 Definition and Basic Characteristics of Stack

A stack (**Stack**) is a type of list implemented to serve applications that require reverse-order processing. In a stack data structure, all operations of inserting or deleting an element must be performed at only one end of the list, called the top of the stack.

A stack is an abstract data structure with the property: Last In, First Out (LIFO). A stack can be visualized through the image of a pile of plates placed on a table: if you want to add a plate, you must place it on the top; if you want to take a plate, you must also take the top one.

The basic operations include **push**, **pop**, **peek** (view the top element without removing it), and **isEmpty** (check if empty); with a fixed-size array implementation, we also have **isFull** (check if full).

Since operations are only performed at the top, the **push/pop/peek/isEmpty** operations all have time complexity $O(1)$. With a dynamic array, this complexity is guaranteed at an average level (amortized $O(1)$).

### 2.1.2 Comparison with Queue

Stack and Queue, although both are linear data structures, operate in completely opposite ways: stack follows LIFO (Last In, First Out), in which the element added last will be the first removed, while queue applies FIFO (First In, First Out) – the element added first will be removed first. This difference is clearly illustrated in Figure 2.1.

In terms of implementation and management, a stack only needs a **top** pointer to manage the top, allowing push and pop operations to always be performed at the same position, while a queue requires two pointers **front** and **rear** to distinguish between enqueue (insert) and dequeue (remove).

Stack is widely used in managing recursive function calls, backtracking algorithms, expression evaluation, and undo/redo functionality, thanks to efficient locality of reference and

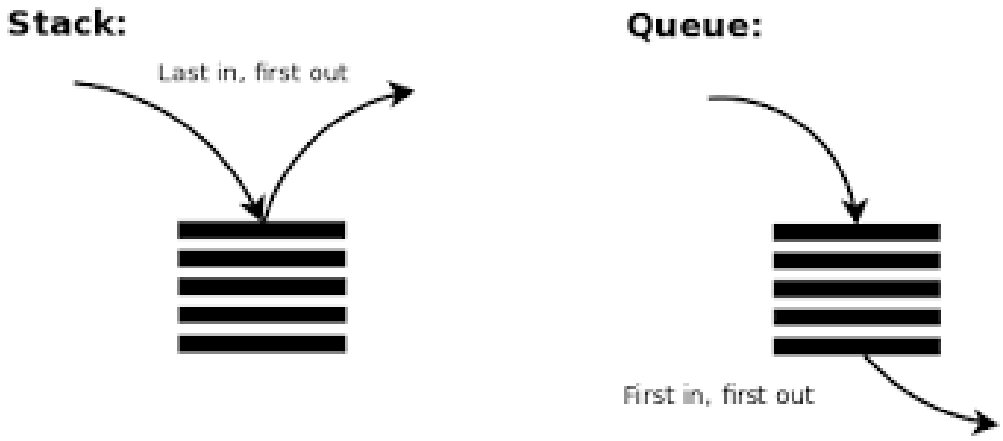


Figure 2.1: Comparison of stack (LIFO) and queue (FIFO) mechanisms

automatic memory management through stack frames. Conversely, queue is often applied in CPU scheduling, print task management, breadth-first search (BFS), I/O buffering, and producer–consumer models, ensuring fair processing order and suitable for continuous data streams.

The difference between LIFO and FIFO not only determines how data is accessed but also affects memory efficiency, cache performance, and the practical applications of each structure in computer science.

### 2.1.3 Basic Operations

#### 2.1.3.1 Push

Push is used to insert a new element onto the top of the stack. After pushing, the new element becomes the new top of the stack. The size of the stack increases by 1 after each successful push. In the case of a fixed-size array implementation, the push operation may fail if the stack is already full (overflow state).
Example: if the stack currently contains [A, B] (with B being the top), after performing Push(C), the stack becomes [A, B, C] (with C as the new top).

#### 2.1.3.2 Pop

Pop is used to retrieve and simultaneously remove the element currently at the top of the stack. The element immediately below (if any) becomes the new top. The size of the stack decreases by 1 after each successful pop.
Example: if the stack is [A, B, C] (C is the top), after performing Pop(C), the stack becomes [A, B] (with B as the new top) and the value C is returned.

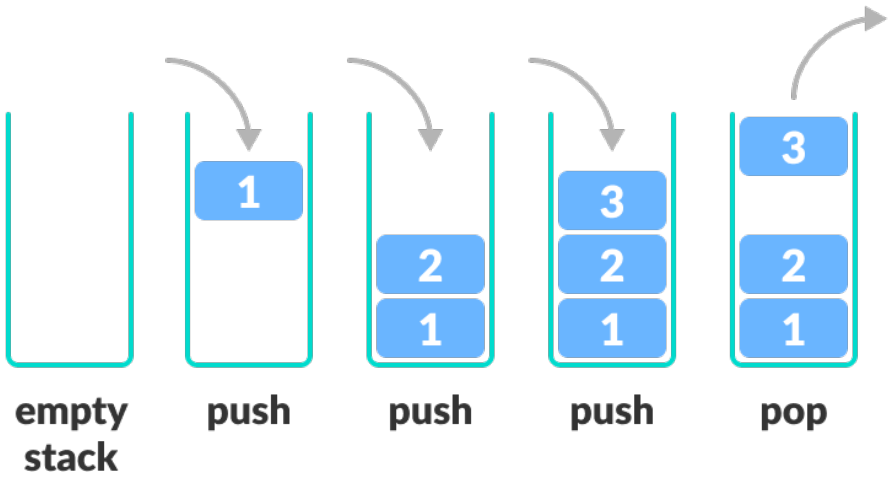


Figure 2.2: Basic operations of a stack

#### 2.1.3.3 Peek/Top

Peek (Top) allows us to view the value of the element currently at the top of the stack without removing it. This operation is useful when you want to know which element will be processed next without changing the current state of the stack.
Example: if the stack is [A, B, C] (C is the top), then performing Peek() will return the value C, but the stack remains [A, B, C].

#### 2.1.3.4 isEmpty and isFull

isEmpty(): Checks whether the stack is empty. Returns true if the stack is empty, and false if it still contains elements. This operation is particularly important to avoid a **Stack Underflow** error when performing pop or peek on an empty stack.

isFull(): Checks whether the stack is full (only applicable when the stack is implemented with a fixed-size array). Returns true if the stack is full, and false if there is still room. This operation helps prevent a **Stack Overflow** error when attempting to push onto a full stack.

#### 2.1.3.5 size

Returns the number of elements currently in the stack. This function is useful for monitoring the capacity of the **Stack** at a given time, for example, to check whether the stack is empty, full, or how many elements it contains.

### 2.1.4 Stack Implementation Methods

#### 2.1.4.1 Array-based Implementation

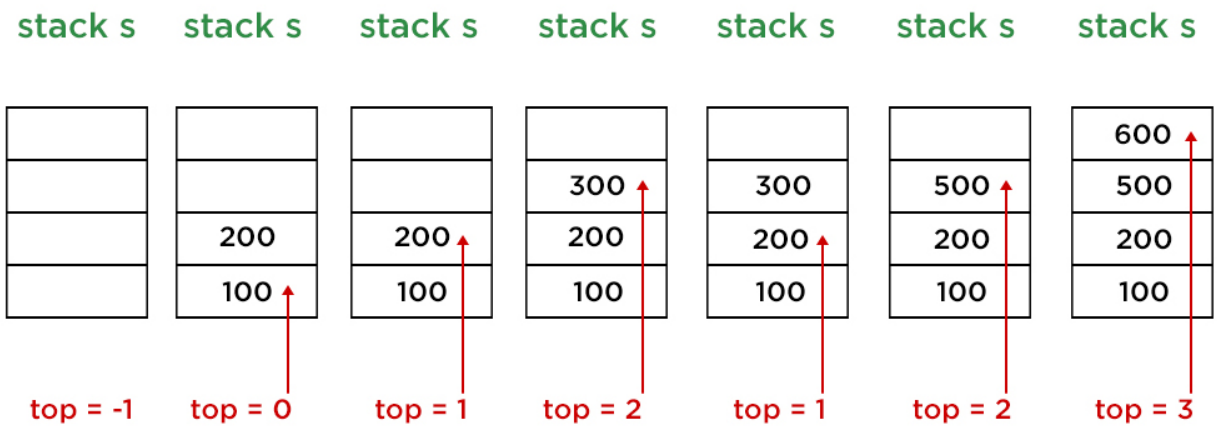


Figure 2.3: Illustration of stack operations using array implementation

This is a simple and common method of implementing a stack, using a one-dimensional array to store the elements. A special variable, usually called **top**, is used to track the index of the element located at the top of the stack. The array-based implementation steps include:

- **Initialization:** First, allocate an array with a fixed size defined as **maxSize**. At the same time, initialize the **top** variable to **-1** to indicate that the stack is currently empty, with no elements inserted.
- **Push(element):** When adding a new element to the stack, first check whether the stack is full using the **isFull** function. If **top == maxSize - 1**, meaning the array is already full, this operation cannot be performed because it would cause a **Stack Overflow**. If the stack is not full, increase the **top** variable by one and assign the new element's value to **array[top]**.
- **Pop():** When retrieving an element from the stack, first check whether the stack is empty using the **isEmpty** function. If **top == -1**, meaning the stack is empty, this operation cannot be performed because it would cause a **Stack Underflow**. If the stack contains elements, retrieve the value at **array[top]**, return this value, and then decrease the **top** variable by one to update the new top of the stack.

- **Peek()/Top():** This operation allows observing the value of the element at the top of the stack without removing it. Similar to **Pop()**, before performing it, you must check whether the stack is empty using **isEmpty**. If the stack is not empty, the result is the value at **array[top]**, while the **top** variable remains unchanged.

***Note***: When using *dynamic arrays* (arrays that automatically resize when adding elements beyond the current capacity) such as **std::vector** in C++ or **list** in Python, the **push**, **pop**, **peek**, and **isEmpty** operations generally have a time complexity of $O(1)$ (in amortized average cases); therefore, checking for **isFull** is usually unnecessary.

#### 2.1.4.2 Linked List-based Implementation

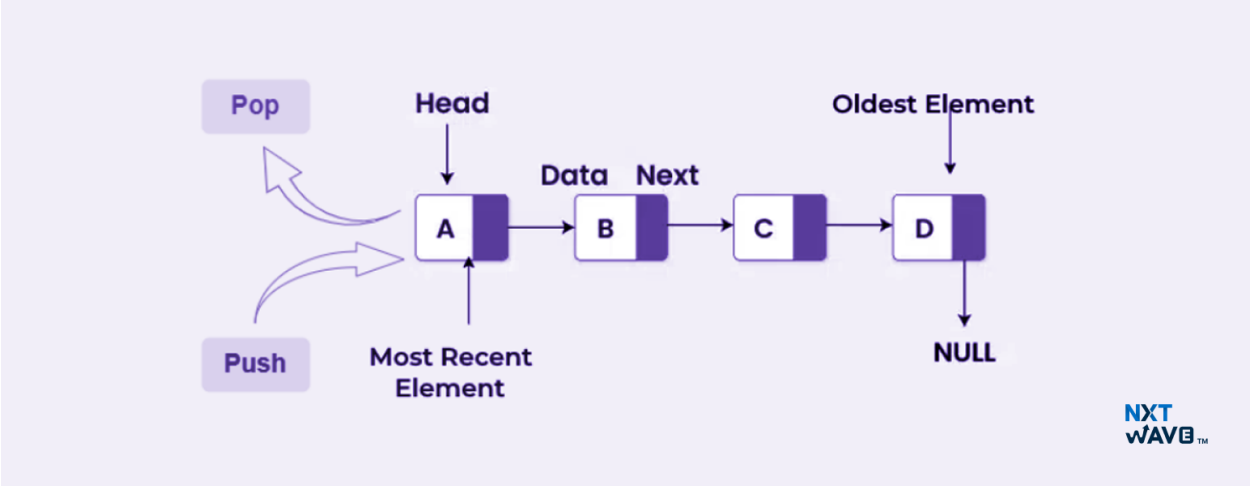


Figure 2.4: Illustration of stack operations using linked list implementation

In this method, the stack is not implemented on a fixed-size array but is built based on a singly linked list. Each element in the stack is represented by a *node*, in which each node consists of two main components: (i) the data of the element, and (ii) a pointer (**next**) pointing to the node immediately below it in the stack. The **top** variable plays an important role, always pointing to the node currently at the top of the stack. Thanks to this, insertion or removal of elements can be performed quickly at this position.

- **Initialization:** At the beginning, set **top = NULL** to indicate that the stack is empty. This means no nodes have been created yet, and any **pop** or **peek** operation must check the empty condition before execution.

- **Push(element):** To insert a new element into the stack, first create a new node and assign data to it. Next, set the **next** pointer of the new node to point to the node

currently referenced by **top**. Finally, update **top** to point to this new node. In this way, the new node becomes the top of the stack and will be processed first in subsequent operations.

- **Pop():** To remove the element at the top of the stack, first check whether the stack is empty (**top == NULL**). If the stack is empty, this operation is invalid and usually causes a **Stack Underflow** error. If the stack is not empty, save the node currently pointed to by **top** into a temporary variable, then update **top = top → next** to move the top pointer down to the next element. The data in the temporary node is returned to the user, and the node is freed from memory to avoid a memory leak.

- **Peek():** Similar to **Pop()**, you must first check whether the stack is empty. If it is not empty, the **peek** operation returns the data of the node that **top** points to, without changing the position of **top**. This allows users to observe the value at the top of the stack while keeping the data structure unchanged.

***Note***: The linked list-based implementation has the outstanding advantage of dynamic size, allowing the stack to expand or shrink depending on the actual number of elements, limited only by system memory. Therefore, there is no concept of a "full stack" as in the case of a fixed-size array. However, the disadvantage is that each node requires additional memory for the **next** pointer, and since nodes are not contiguous in memory, memory access efficiency (cache locality) is often lower compared to the array-based implementation.

#### 2.1.4.3 Comparison of Two Stack Implementation Methods

| Criteria | Array | Linked List |
|---|---|---|
| **Advantages** | Simple to implement.<br>Fast access to the top (good cache locality). | Dynamic size, flexible.<br>No concern about stack overflow unless system memory is exhausted. |
| **Disadvantages** | Fixed size, prone to overflow or waste.<br>Resizing is costly. | Requires extra memory for pointers.<br>Poor locality, may be slower. |
| **Suitable Applications** | When the maximum number of elements is known in advance, and high access speed is required. | When the number of elements varies significantly, requiring flexibility, while accepting extra memory overhead. |

Table 2.1: Comparison of array-based and linked list-based stack implementations

### 2.1.5 Complexity Analysis

#### 2.1.5.1 Time Complexity of Basic Operations

Since all main operations on a stack (`push`, `pop`, `peek`, `isEmpty`) are performed directly at the top, their theoretical time complexity is $O(1)$. However, differences arise depending on the specific implementation (fixed array, dynamic array, or linked list):

| Operation | Array (fixed) | Array (dynamic)[1] | Linked List |
|---|---|---|---|
| `push` | $O(1)$ | $O(1)$ (*Worst case* $O(n)$) | $O(1)$ |
| `pop` | $O(1)$ | $O(1)$ | $O(1)$ |
| `peek/top` | $O(1)$ | $O(1)$ | $O(1)$ |
| `isEmpty` | $O(1)$ | $O(1)$ | $O(1)$ |

Table 2.2: Time complexity of main stack operations

[1]Examples: `std::vector` in C++, `ArrayList` or `list` in Python.

***Note:*** For **dynamic arrays**, the `push` operation is usually $O(1)$; only when resizing is required does it cost $O(n)$. Thus, amortized complexity is still considered $O(1)$. Meanwhile, the **linked list**, although also $O(1)$ for `push`/`pop`, requires extra memory for pointers and nodes scattered across RAM, making real performance often slower than arrays. It is only truly suitable for large datasets where elements are frequently added/removed. If `size()` needs to run in $O(1)$, we simply maintain an additional counter variable updated during `push`/`pop`.

#### 2.1.5.2 Memory Usage

| Characteristic | Fixed Array | Dynamic Array | Linked List |
|---|---|---|---|
| **Allocation method** | Pre-allocate an array of size `maxSize` | Allocated proportional to the number of elements, with some extra capacity | Each element is a node, consisting of data + `next` pointer |
| **Storage complexity** | $\Theta$(`maxSize`) (occupies memory upfront) | $\Theta(n)$, resizing when full | $\Theta(n)$ data + $\Theta(n)$ pointer overhead |
| **Capacity limit** | Fixed by `maxSize` | Flexible, no need for `isFull` | Unlimited, only stops when RAM is exhausted |

Table 2.3: Comparison of memory complexity for stack implementation methods

***Note***: Depending on the actual application case, we must choose different stack implementation methods. Each method has its own limitations that must be considered before use. If using a fixed array, predefining the size `maxSize` is the biggest drawback: choosing too small may cause overflow, while choosing too large leads to memory waste. With dynamic arrays, although more flexible, expanding capacity (resize) when full requires copying all data to new memory, costing $O(n)$ time in the worst case; moreover, over-allocation to reduce resize frequency may waste unused memory. Meanwhile, linked lists solve the fixed-size problem but incur memory overhead because each node must store a `next` pointer, and nodes scattered in RAM reduce cache locality, making real access speed often slower than arrays.

#### 2.1.5.3 Evaluation

There are two main methods to evaluate algorithm execution time, compared in the following table:

| Method | Approach | Advantages | Disadvantages |
|---|---|---|---|
| **Experimental** | Implement and run the program, measuring execution time on different datasets. | Concrete, real results. Reflects actual runtime. | Dependent on hardware, language. Difficult to select fully representative datasets. Time-consuming and costly. |
| **Asymptotic Analysis** | Theoretical analysis, using Big-O notation to express complexity when input size is large. | Independent of hardware and language. Easy to compare algorithms. No implementation required. Suitable for large datasets. | Does not provide exact runtime. Ignores constants. Less accurate for small datasets. |

Table 2.4: Comparison of two methods for algorithm evaluation

From the comparison table, it can be seen that in the case of studying the **Stack** data structure, the experimental approach is more intuitive and easier to implement. The chosen programming languages are **Python** (simple syntax, built-in `list` supporting `append`/`pop`) and **C++** (high performance, using `std::stack` based on `vector`). The experiment was conducted on an **Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz**.

**Benchmark with Python**

```
import time
stack = []
N_1 = 1000000
N_2 = 10000000
def benchmark(num_elements):
    # Benchmark push
    start = time.time()
    for i in range(num_elements):
        stack.append(i)
    end = time.time()
    print(f"Push {num_elements} elements took: {end - start:.4f} 
```

```
        seconds")
    # Benchmark pop
    start = time.time()
    for i in range(num_elements):
        stack.pop()
    end = time.time()
    print(f"Pop {num_elements} elements took: {end - start:.4f}
        seconds")
benchmark(N_1)
benchmark(N_2)
```

*Program output*:

```
Push 1000000 elements took: 0.0530 seconds
Pop 1000000 elements took: 0.0520 seconds
Push 10000000 elements took: 0.5318 seconds
Pop 10000000 elements took: 0.5179 seconds
```

**Remarks:**

- Both `push` and `pop` are very fast (approximately 0.05s for 1 million elements and approximately 0.5s for 10 million elements).
- The results are consistent with theoretical analysis: amortized $O(1)$ for both operations.
- Advantage of experimental evaluation: provides concrete numbers, easy to visualize.

**Benchmark with C++**

```
#include <iostream>
#include <stack>
#include <vector>
#include <chrono>
using namespace std;
using namespace std::chrono;
void benchmark(int N) {
    stack<int, vector<int>> s;
    // Push benchmark
```

```
    auto start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) s.push(i);
    auto end = high_resolution_clock::now();
    cout << "Push " << N << " elements took: "
         << duration<double>(end - start).count() << " seconds\n";
    // Pop benchmark
    start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) s.pop();
    end = high_resolution_clock::now();
    cout << "Pop " << N << " elements took: "
         << duration<double>(end - start).count() << " seconds\n";
}
int main() {
    benchmark(1000000);   // 1 million elements
    benchmark(10000000);  // 10 million elements
    return 0;
}
```

*Program output*:

```
Push 1000000 elements took: 0.0090237 seconds
Pop 1000000 elements took: 0.017885 seconds
Push 10000000 elements took: 0.0955453 seconds
Pop 10000000 elements took: 0.161032 seconds
```

**Remarks:**

- C++ produces results many times faster than Python thanks to optimized compilation and efficient static memory management (Python pushing 10 million elements took: 0.5318s, while C++ took only: 0.0955453s).
- Depending on the project goals (performance or ease of use, etc.), the choice of programming language may vary.

## 2.2 Practical Applications

Although the stack data structure has a simple implementation, its range of applications is extremely diverse and efficient thanks to the LIFO (Last In, First Out) principle along with

basic operations such as push and pop. In this report, we illustrated the typical applications of the stack through a demo web application developed using Python. Access the demo web at: https://demo-stack.streamlit.app/.

### 2.2.1 Managing Recursive Functions

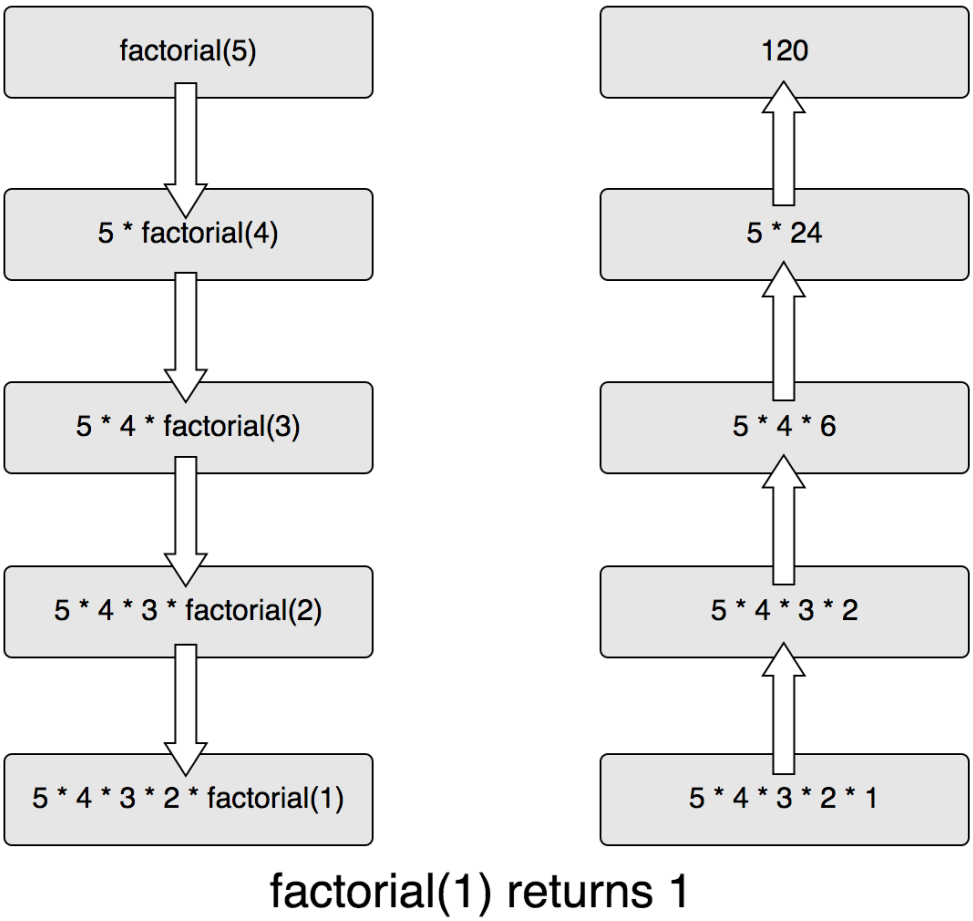


Figure 2.5: Calculating factorial using recursion

During program execution, the runtime system uses the stack data structure to manage the states of functions currently being executed. This mechanism plays a key role in enabling recursive functions to work effectively.

When a function is called, the system creates a new *frame* (stack frame) and pushes it onto the Call Stack. Each frame contains the necessary information such as:

- The function parameters,
- The function's local variables,
- The return address, so that after the function finishes, the program can continue at the calling point.

After the function completes its execution, the corresponding frame is popped from the Call Stack, and the program resumes execution at the stored return address. With this mechanism, each recursive call is stored independently in its own frame, preventing state conflicts among nested calls. However, if the recursion depth is too large and exceeds the Call Stack limit, the system will raise a **Stack Overflow** error, causing the program to crash.

The process of calculating factorial using recursion is clearly illustrated in Figure 2.5. As shown, the process takes place in two main stages: **descent** and **ascent**. During the descent phase, nested recursive calls are made in succession:

- Starting from the call `factorial(5)`, the program recursively calls down to smaller values: `factorial(4)` → `factorial(3)` → `factorial(2)` → `factorial(1)`.
- Each new call is added to the Call Stack, forming a chain of stacked frames reflecting the LIFO nature of the stack.

When the base case is reached (`factorial(1)` = 1), the system begins the ascent phase. The results are returned successively from the bottom up:

- First, `factorial(1)` returns 1. This result is then used to compute the next values: `factorial(2)` = 2×1 = 2 →`factorial(3)` = 3 ×2 = 6 →`factorial(4)` = 4 ×6 = 24.
- Finally, `factorial(5)` = 5 × 24 = 120.

This process highlights the role of the stack in storing temporary states and ensuring the correct return order, thanks to the LIFO principle. Each time a function completes, it is **popped** from the stack, and the result is passed back to the previous caller.

### 2.2.2 Expression Conversion and Evaluation

#### Concepts

- **Infix Expression**: The common notation, where the operator is written *between* operands. Example:

$$A + B * C$$

- **Postfix Expression (Reverse Polish Notation - RPN)**: The operator is placed *after* the operands. Example:

$$A\ B\ C * +$$

- **Prefix Expression**: The operator is placed *before* the operands. Example:

$$+A * BC$$

**Problem:** In infix expressions, the computation order depends on **operator precedence** and **parentheses**.

- $A + B * C \Rightarrow$ must compute $B * C$ first.
- $(A + B) * C \Rightarrow$ must compute $A + B$ first.

Computers cannot directly interpret infix expressions like humans, so we need a way to convert infix expressions into postfix or prefix for consistent and simpler evaluation.

**Shunting-yard Algorithm:** To solve this, mathematician **Edsger Dijkstra** proposed the main idea of using a **stack** to handle operators.

**Working principle:**

1. Read the infix expression from left to right.
2. If an **operand** (A, B, C, number, variable...) is encountered $\Rightarrow$ put it directly into the output.
3. If an **operator** (+, -, *, /...) is encountered:
    - Compare its precedence with the operator at the top of the stack.
    - If the operator on the stack has higher or equal precedence $\Rightarrow$ pop it into the output first, then push the new operator.
    - If the operator on the stack has lower precedence $\Rightarrow$ push the new operator onto the stack.
4. If an **opening parenthesis "("** is encountered $\Rightarrow$ push it onto the stack.
5. If a **closing parenthesis ")"** is encountered $\Rightarrow$ pop operators from the stack into the output until an opening parenthesis is found.
6. After reading the entire expression $\Rightarrow$ pop all remaining operators from the stack into the output.

**Illustrative example:**

$$A + B * C$$

1. Read $A \Rightarrow$ Output: $A$
2. Read $+ \Rightarrow$ Stack: $\{+\}$
3. Read $B \Rightarrow$ Output: $A\ B$
4. Read $*$: since $*$ has higher precedence than $+$, push it onto the stack $\Rightarrow$ Stack: $\{+, *\}$
5. Read $C \Rightarrow$ Output: $A\ B\ C$
6. End of expression $\Rightarrow$ Pop all operators from the stack: first $*$, then $+$.

   $\rightarrow$ **Postfix Result:** A B C * +

   $\rightarrow$ **Prefix Result:** + A * B C

### 2.2.3 DFS Algorithm and Backtracking

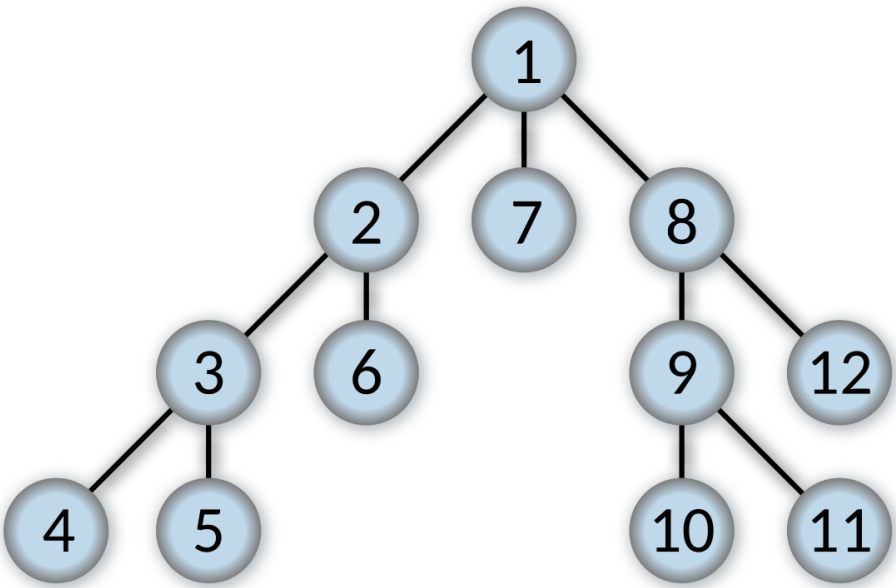


Figure 2.6: Illustration of DFS algorithm on a tree

**Concepts:**

- **DFS (Depth-First Search):** This is a graph or tree traversal algorithm that starts from an initial vertex (or node), then explores as deep as possible along each branch before backtracking to continue exploring other branches.

- DFS can be implemented using *recursion*, in which case the program utilizes the system **Call Stack** (or implemented with an explicit *Stack* defined by the programmer).
- At each step, DFS always selects the next vertex from the top element of the stack to expand.

- **Backtracking:** This is a technique used to search for solutions in the space of possible configurations, by trying step by step, and if it detects that the current path cannot lead to a solution, it backtracks to the previous step to try another direction.
  - Backtracking can be considered a generalized form of DFS, as it also explores depth-first but includes constraint checking.
  - When encountering a dead end (invalid path), the program will **pop** the nearest state from the stack and backtrack to try another branch.
  - This technique is commonly used in problems such as: solving Sudoku, finding paths in a maze, generating combinations/permutations under constraints.

**Comparison between DFS and Backtracking**

- **DFS:** Mainly used for traversal or search in graphs/trees, without necessarily checking constraints.
- **Backtracking:** Based on DFS but adds constraint checking and pruning of invalid branches, enabling the solving of more complex constraint problems.

### 2.2.4 Applications in Software and Daily Life

The stack not only appears in theory but is also widely applied in many common programs and tools used in everyday life:

- **Undo/Redo Functionality:** In word processors or image editing software, the system often maintains two stacks.
  - Each new action is `pushed` onto the **Undo** stack.
  - When the user selects *Undo*, the top action is `popped` from Undo and simultaneously `pushed` onto the **Redo** stack.
  - When selecting *Redo*, the system performs the reverse process.

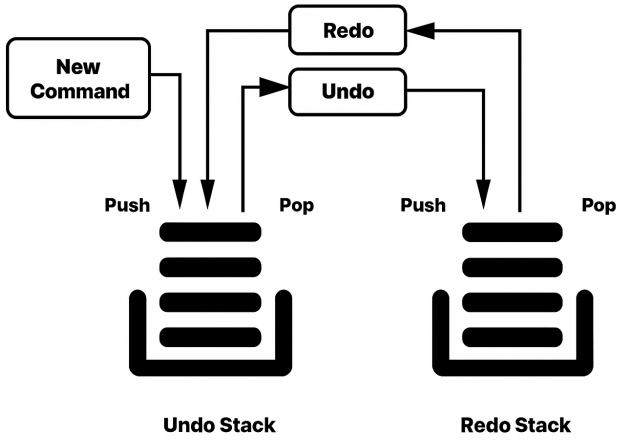


Figure 2.7: Undo / Redo using Stacks

- **Web Browsing History:** Browsers use two stacks to manage the *Back* and *Forward* buttons.
  - When opening a new page, the current URL is `pushed` onto the "Back" stack.
  - When clicking *Back*, the URL is `popped` and simultaneously `pushed` onto the "Forward" stack.
  - This allows users to easily navigate backward or forward.
- **Parentheses Checking in Expressions:** When scanning a string of characters:
  - Encounter an opening parenthesis "(" ⇒ `push` onto the stack.
  - Encounter a closing parenthesis ")" ⇒ `pop` to check if it matches the nearest opening parenthesis.
  - The LIFO mechanism ensures correct nesting order.

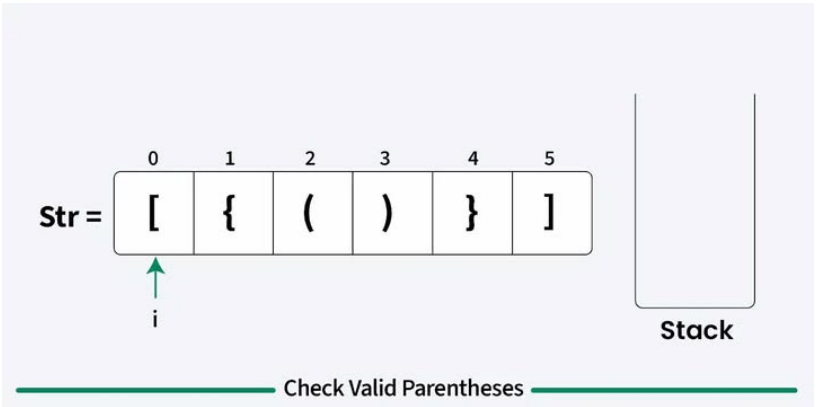


Figure 2.8: Parentheses checking using Stack

- **Base Conversion:** When converting a decimal number into another base:
  - Repeatedly divide the number by the new base and `push` the remainders onto the stack.
  - Then, `pop` the remainders in reverse order to obtain the final result.

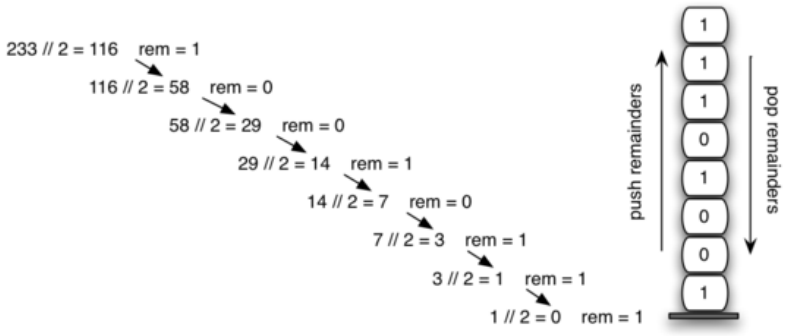


Figure 2.9: Converting from base 10 to base 2

### 2.2.5 Advanced Applications

In addition to familiar applications, stacks are also used in many modern fields:

- **Natural Language Processing (NLP):** The *shift-reduce parsing* algorithm relies on stacks to construct syntax trees for sentences. Some deep learning models such as **Stack-LSTM** and **Neural Stack** combine stack mechanisms with neural networks to learn structured language.
- **Machine Learning:** In reverse-mode automatic differentiation (autodiff) and the backpropagation algorithm, stacks (or *tapes*) are used to store computations. The stack is then traversed in reverse to compute derivatives and gradients for the model.
- **Computer Vision:** Image processing algorithms such as *flood-fill*, *region-growing*, or connected component labeling can all be implemented using DFS with stacks to expand image regions.

### 2.2.6 Illustrations in Popular Programming Languages

The code snippets below demonstrate how stacks can be implemented and used in C++, Python, and C#.

#### 2.2.6.1 C++

**C++ with arrays and the `std::stack` library**

```cpp
#include <iostream>
#include <stack> // stack in STL
using namespace std;
int main() {
    stack<int> s;
    s.push(10); // push elements
    s.push(20);
    s.push(30);
    cout << s.top() << " popped\n"; // show and pop top
    s.pop();
    cout << "Top element is:" << s.top() << "\n"; // peek top
    cout << "Elements present in stack: ";
    while (!s.empty()) {
        cout << s.top() << " "; // print and pop
        s.pop();
    }
    cout << "\n";
}
```

*Program output*:

```
30 popped from stack
Top element is: 20
Elements present in stack: 20 10
```

**Notes:**

- Three elements 10, 20, 30 are sequentially `pushed` onto the stack (following LIFO order).
- The command `s.top()` returns 30, then `s.pop()` removes it.
- When calling `s.top()` again, the value 20 appears because it is now on top.
- The loop `while (!s.empty())` iterates through the stack: printing 20 and then 10, consistent with the LIFO principle.

#### 2.2.6.2 Python

**Python with `list`**

```
# Stack implementation using list
def create_stack():
    return []
def is_empty(stack):
    return len(stack) == 0
def push(stack, item):
    stack.append(item)
    print("pushed item:", item)
def pop(stack):
    if is_empty(stack):
        return "stack is empty"
    return stack.pop()
# Test
stack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)
print("popped item:", pop(stack))
print("stack after popping:", stack)
```

*Program output:*

```
pushed item: 1
pushed item: 2
pushed item: 3
popped item: 3
stack after popping: [1, 2]
```

**Python with `collections.deque`**

```
# Stack implementation using collections.deque
from collections import deque
def create_stack():
    return deque()
def is_empty(stack):
    return len(stack) == 0
```

```
def push(stack, item):
    stack.append(item)  # append to end
    print("pushed item:", item)
def pop(stack):
    if is_empty(stack):
        return "stack is empty"
    return stack.pop()  # pop from end

# Test
stack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)
print("popped item:", pop(stack))
print("stack after popping:", stack)
```

*Program output:*

```
pushed item: 1
pushed item: 2
pushed item: 3
popped item: 3
stack after popping: deque([1, 2])
```

**Notes:**

- With `list`, the `append` and `pop` operations correspond to stack push and pop, but when the number of elements grows large, resizing can be costly.
- `deque` is optimized for adding/removing at both ends with stable $O(1)$ complexity, making it suitable for large stacks/queues.
- In practice, `deque` is often recommended over `list` when performance is critical.

#### 2.2.6.3 C#

**C# with `System.Collections.Generic.Stack`**

```
using System;
```

```
using System.Collections.Generic;
class Program {
    static void Main() {
        // Declare stack of integers
        Stack<int> stack = new Stack<int>();
        // Push objects
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);
        // Pop object
        Console.WriteLine($"{stack.Pop()} popped from stack");
        // Peek top object
        Console.WriteLine("Top element is: " + stack.Peek());
        // Iterate stack
        Console.Write("Elements present in stack: ");
        foreach (int item in stack) {
            Console.Write(item + " ");
        }
        Console.WriteLine();
    }
}
```

**Program output**:

```
30 popped from stack
Top element is: 20
Elements present in stack: 20 10
```

**Notes:**

- `Stack<int>` is declared from the `System.Collections.Generic` namespace.
- The elements 10, 20, and 30 are pushed into the stack in sequence.
- The `Pop()` command returns 30 and removes it; `Peek()` then shows 20.
- The `foreach` loop iterates through the stack following the LIFO principle: printing 20, then 10.

# CONCLUSION

## 3.1 Achieved Results

Through systematic research and analysis, this report has clarified the core concepts of the stack data structure, including the LIFO (Last In First Out) principle, its key characteristics, the set of basic operations, and common implementation methods. Based on this theoretical foundation, the study further analyzed algorithmic complexity and explored practical applications such as managing recursive function call stacks, number base conversion algorithms, mathematical expression evaluation, along with several widely used applications in computer science.

In addition to the content contributions, the process of completing this report also provided team members with opportunities to expand their professional knowledge, develop collaborative research skills, and improve the quality of individual contributions to the collective outcome.

## 3.2 Future Development Directions

Stack is an important data structure that plays a fundamental role in programming and computer science. With the LIFO principle, stack allows data management such that the most recently inserted element is the first to be removed. This mechanism is suitable for managing function calls, enabling programmers to implement recursion, nested function calls, and efficient local variable management. Furthermore, stack is widely applied in mathematical expression processing, traversal algorithms such as DFS, and the implementation of utility features in software applications, such as undo/redo. It can be said that stack is not only a basic data structure but also an indispensable tool, contributing to the operational mechanisms of both computer systems and modern applications.

In the future, research can focus on optimizing stacks in specific environments, such as embedded systems where memory resources are limited. Moreover, exploration can be extended to stack variants such as deque (double-ended queue) or dynamic stacks to solve more complex problems. Another important direction is combining stacks with other data struc-

tures (queues, trees, graphs) to create flexible solutions that effectively support applications in artificial intelligence, big data processing, and parallel programming.

# REFERENCES


[1] Wikibooks. (n.d.). *Data structures/Stacks and queues*. Wikimedia Foundation. https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues

[2] CMU 15-122 Course Staff. (n.d.). *Review: Stacks and queues* [Lecture slides]. Carnegie Mellon University. https://www.cs.cmu.edu/~15122/handouts/slides/review/09-stackqueue.pdf

[3] Quantrimang.com. (n.d.). *Stack data structure*. https://quantrimang.com/cong-nghe/cau-truc-du-lieu-ngan-xep-stack-156375

[4] Vinh, L. V. (2013). *Textbook of Data Structures and Algorithms*. Vietnam National University – Ho Chi Minh City.

[5] The Algorithms. (n.d.). *The Algorithms - Python* [Source code]. GitHub. https://github.com/TheAlgorithms/Python

# APPENDIX

| Abbreviation | Explanation |
|---|---|
| **AI** | Artificial Intelligence |
| **IoT** | Internet of Things |
| **ADT** | Abstract Data Type |
| **DSU** | Disjoint Set Union |
| **LIFO** | Last In, First Out |
| **FIFO** | First In, First Out |
| **DFS** | Depth First Search |
| **BFS** | Breadth First Search |
| **CPU** | Central Processing Unit |
| **RAM** | Random Access Memory |
| **NLP** | Natural Language Processing |
| **LSTM** | Long Short-Term Memory |

# List of Figures

# List of Tables