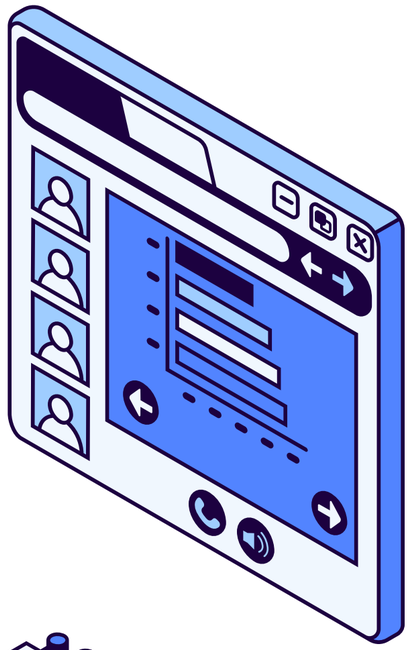
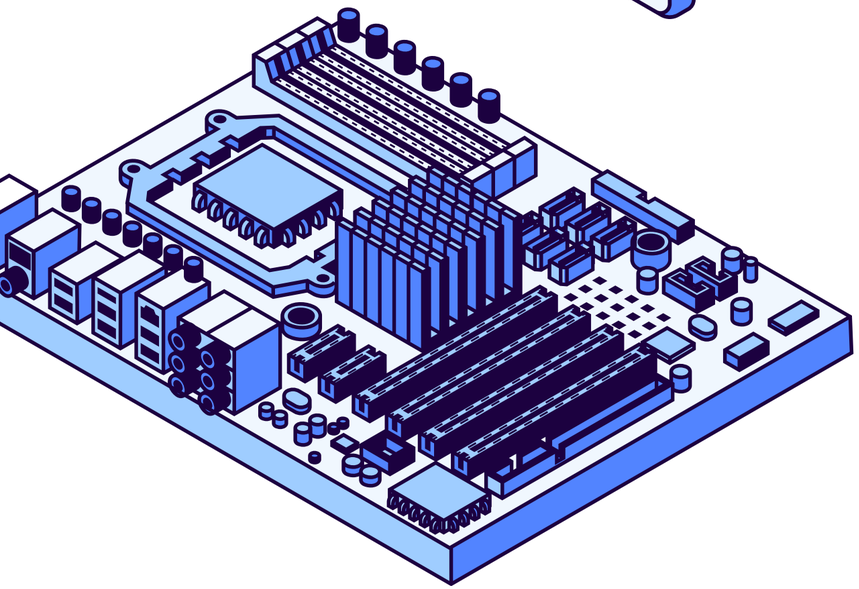




STACK

GROUP 9

Instructor: Th.S Vũ Đình Bảo

Students:

Nguyễn Hùng Dũng	22134002
Trần Như Hoàng	22134006
Trần Nguyên Phương Bình	24133006
Phạm Ngọc Phúc	22134010
Võ Hồng Quân	22134012

Content

01 Introduction to Stack

PART

02 Theoretical Background

PART

- Basic concepts of Stack
- Implementation methods
- Complexity analysis
- Practical applications

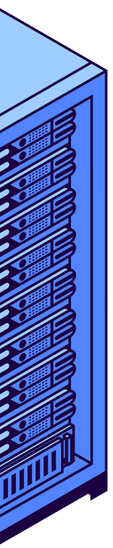
03 Conclusion

PART

- Results
- Future Trends

INTRODUCTION

1.1 Context of the Topic



Programming

It is a key field that supports and benefits many other tasks.

Data Structures and Algorithms

It is the “backbone” and “foundation” for building complex and efficient software:

Abstract Data Type (ADT)

Includes: List, Stack, Queue, Deque, Priority Queue, Set, Map/Dictionary, Disjoint Set.

Stack

Plays a core role in algorithm design and modern systems.

1.2 & 1.3 Objectives and Scope of Study

Clarify Fundamental Concepts: Stack (LIFO principle, basic operations).

Performance Analysis & Implementation: Array-based and linked-list-based implementations; source code in multiple programming languages.

Illustrate Practical Applications: Call Stack, expression evaluation, DFS, Undo/Redo.

Scope Limitations: Excludes other data structures (except basic Queue), Does not focus on hardware-level stack implementation.

1.4 Research Methods

Literature review method

Theoretical analysis & synthesis

Experimental method

Group collaboration method

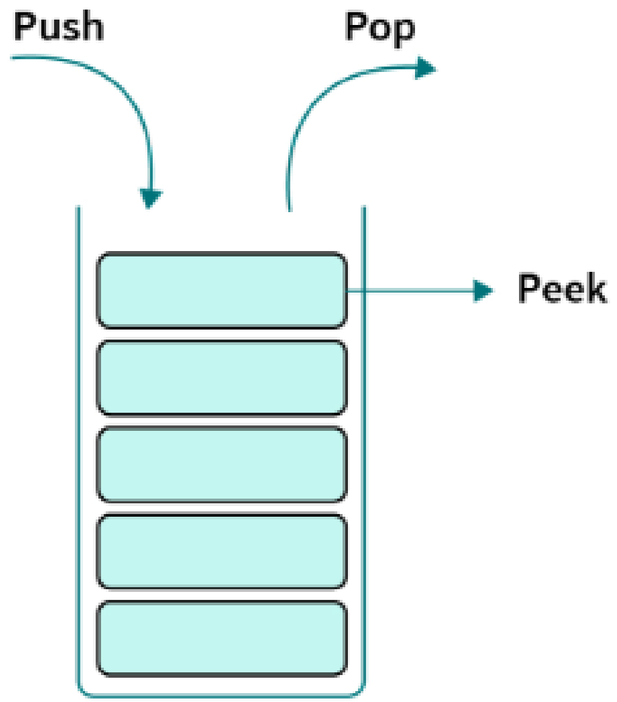
Theoretical Background

2.1.1 Definition and Basic Characteristics of Stack

1. A stack is a data structure that follows the LIFO (Last In, First Out) principle: the last element inserted is the first to be removed.

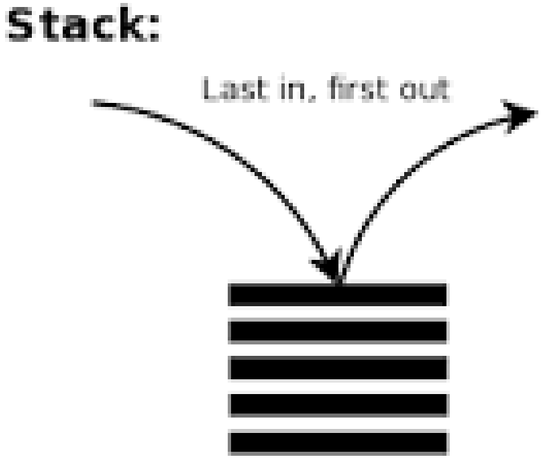
2. Main operations: push, pop, peek...

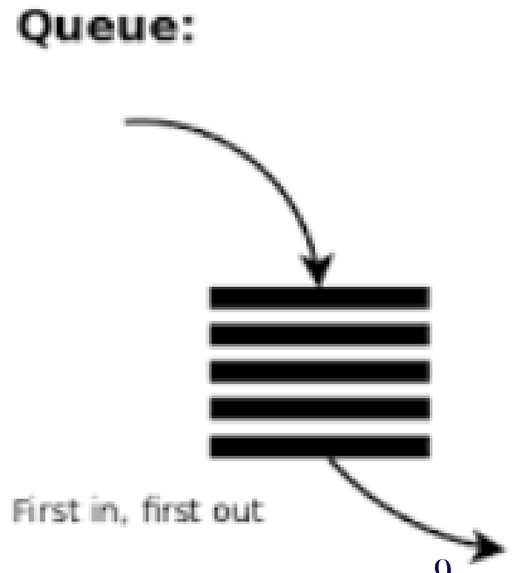
3. Time complexity: O(1) for these operations.



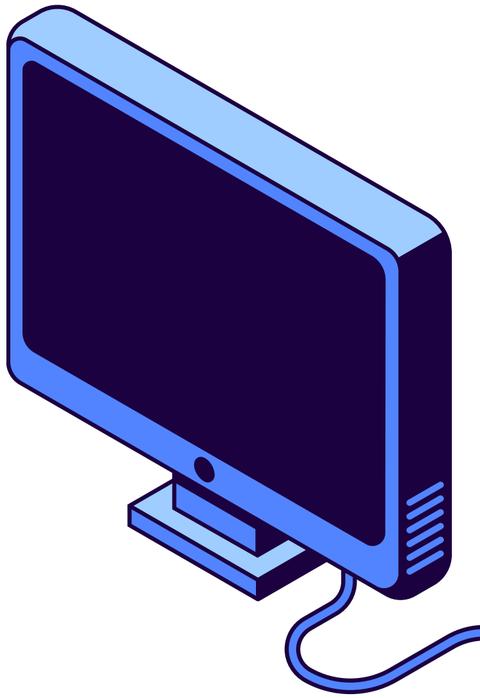
2.1.2 Comparison with Queue

Feature	Stack	Queue
Operating Principle	LIFO (Last In, First Out)	FIFO (First In, First Out)
Operations	push, pop, peek, isEmpty	enqueue, dequeue, front, isEmpty
Management Pointer	One pointer: top	Two pointers: front (head), rear (tail)
Applications	Recursive calls, backtracking, undo/redo, stack frame	CPU scheduling, BFS, I/O buffering, producer-consumer





2.1.3 Basic Operations with Stack



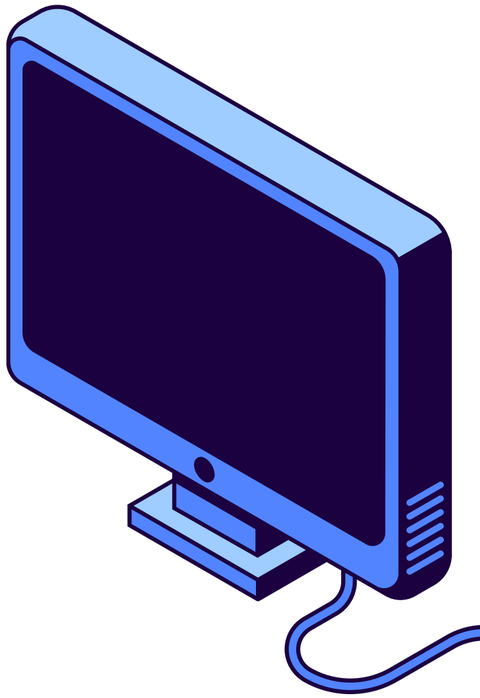
Push (Insert)

- **Purpose:** Add a new element to the top of the stack.
- **Action:** The stack size increases by 1.
- **Note:** May cause Stack Overflow if the stack is implemented using a fixed-size array and is already full.

Example:

- Before: [A, B] (Top = B)
- Push(C)
- After: [A, B, C] (Top = C)

2.1.3 Basic Operations with Stack

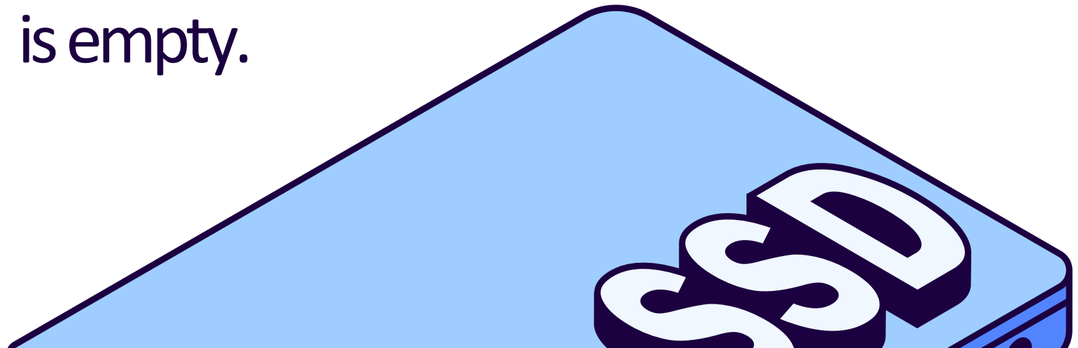


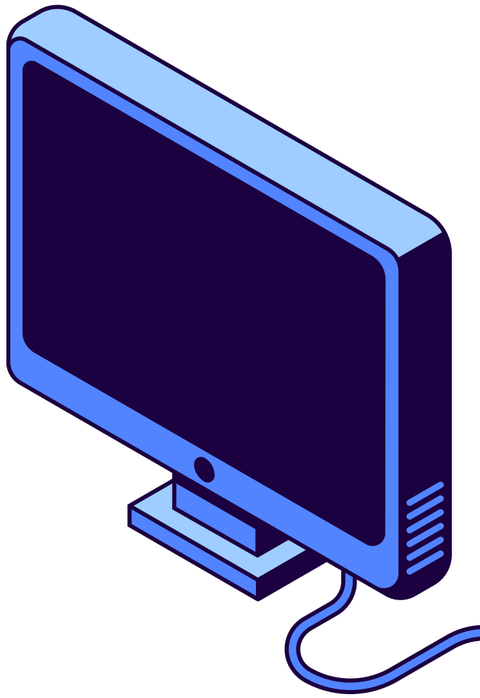
Pop (Remove)

- **Purpose:** Retrieve and remove the top element of the stack.
- **Action:** The stack size decreases by 1.
- **Note:** May cause Stack Underflow if the stack is empty.

Example:

- Before: [A, B, C] (Top = C)
- Pop() → Returns C
- After: [A, B] (Top = B)





2.1.3 Basic Operations with Stack

Peek / Top (View Top)

- **Purpose:** View the value of the top element without removing it.
- **Action:** Does not modify the stack.

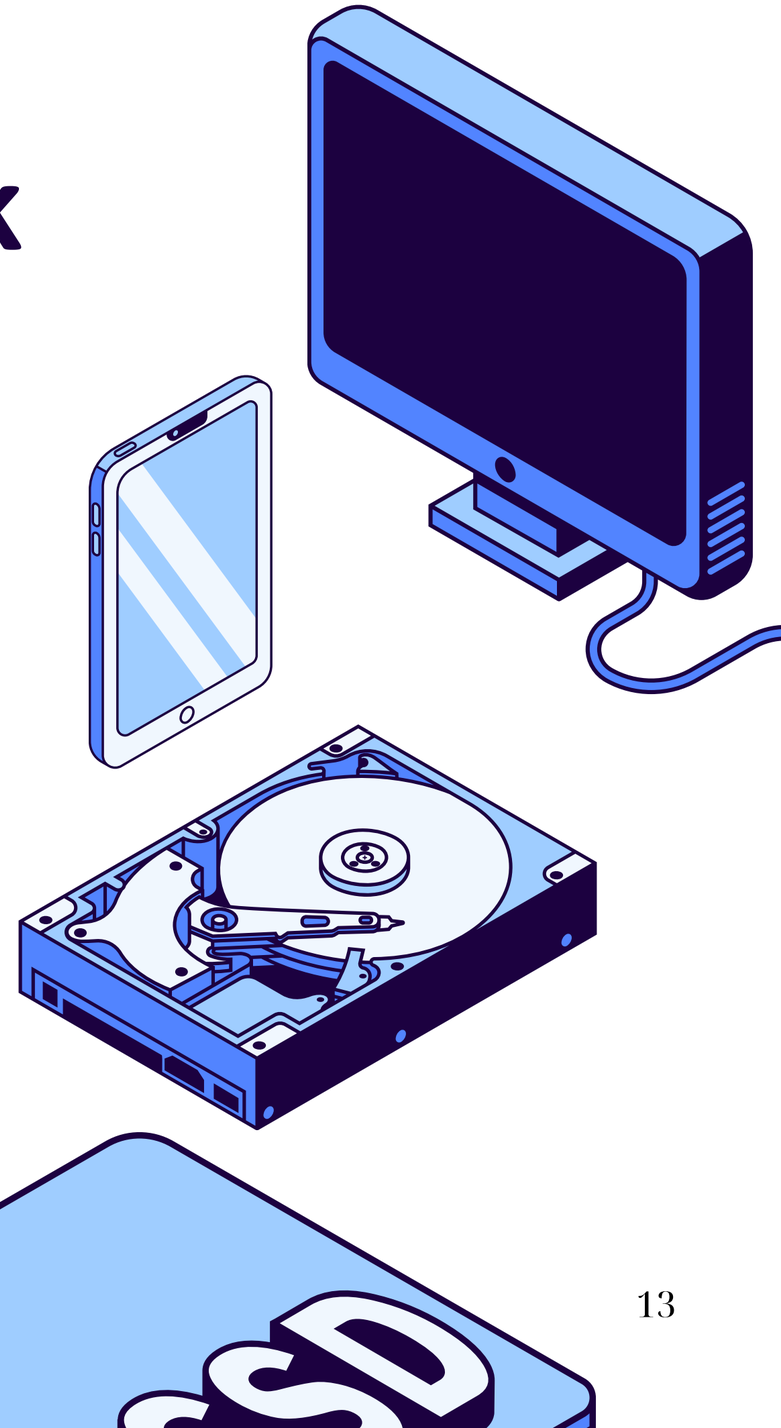
Example:

- Stack: [A, B, C] (Top = C)
- Peek() → Returns C
- Stack remains: [A, B, C]

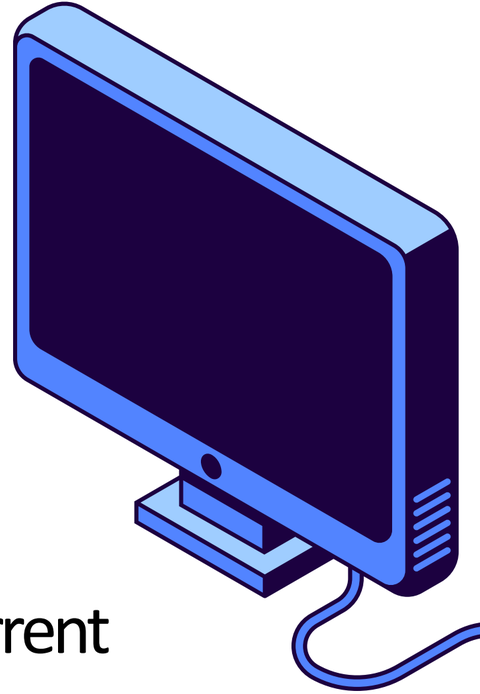
2.1.3 Basic Operations with Stack

isEmpty (Check Empty)

- **Purpose:** Check whether the stack has no elements.
- **Application:** Must be checked before calling Pop() or Peek() to avoid errors.



2.1.3 Basic Operations with Stack

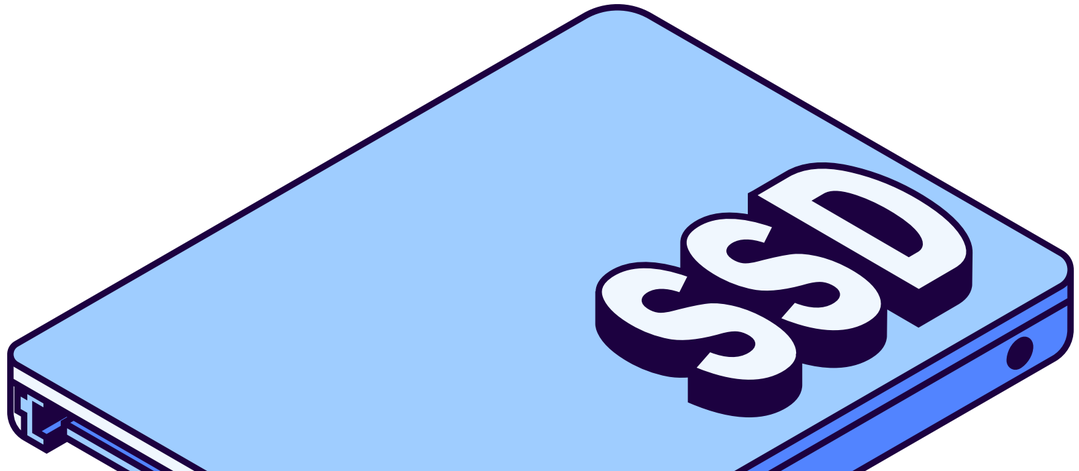


isFull (Check Full)

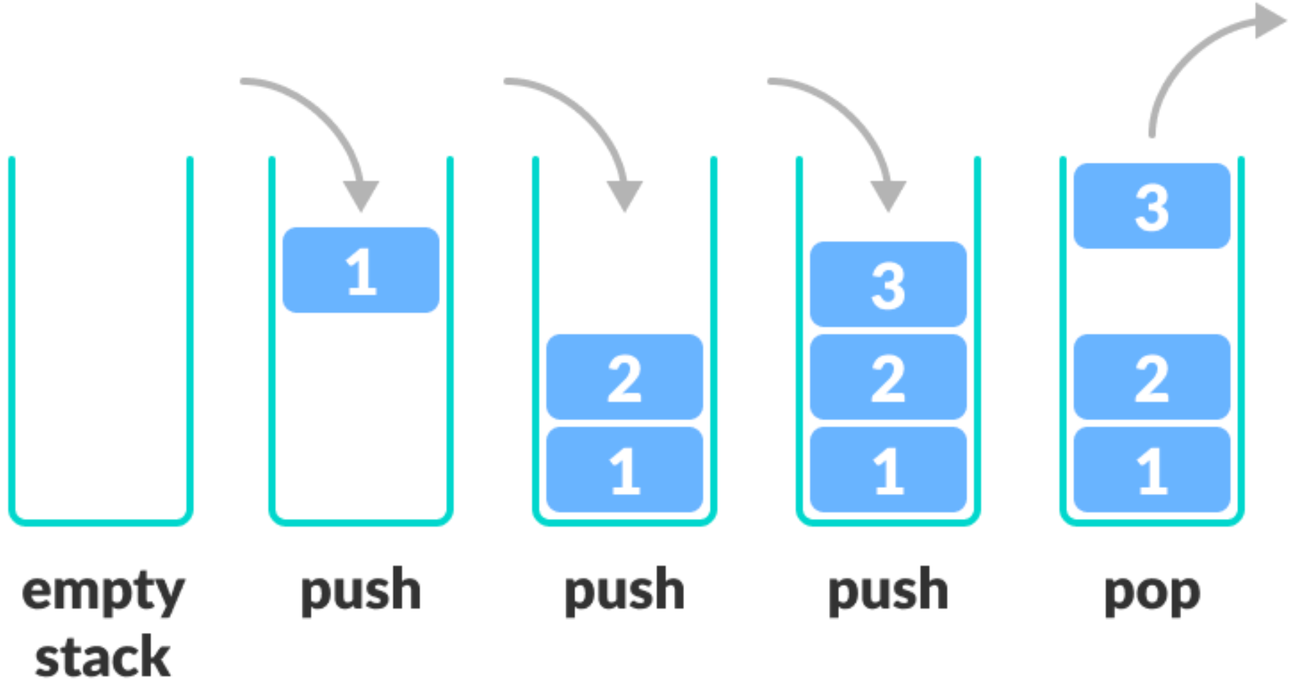
- **Purpose:** Check whether the stack (using a fixed-size array) has no remaining space.
- **Application:** Must be checked before calling Push() to avoid overflow.

size (Stack Size)

- **Purpose:** Return the current number of elements in the stack.



2.1.3 Basic Operations with Stack



2.1.4 Stack Implementation Methods

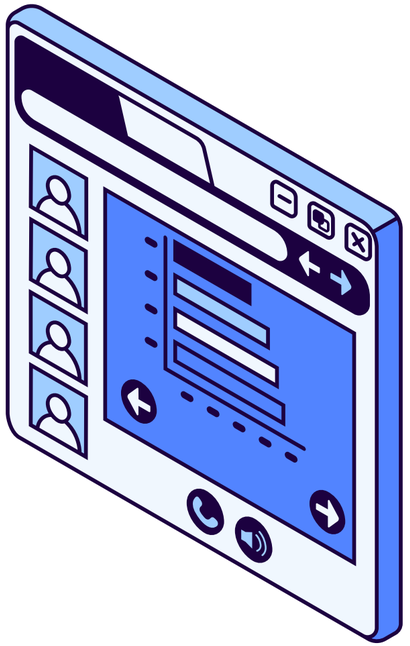
Array-based Implementation

Principle: Use a one-dimensional array + a top variable to manage the top element.

- Push: Assign new value → increment top.
- Pop: Take array[top] → decrement top.
- Peek: Access array[top].

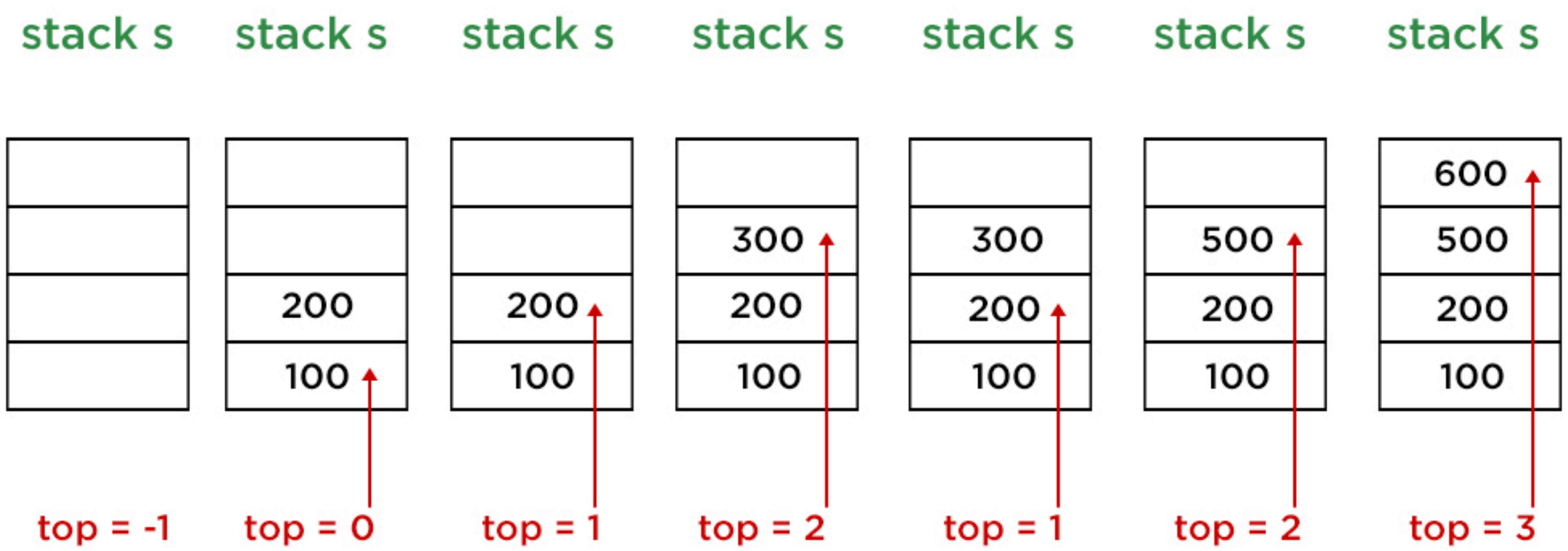
Advantages: Simple, fast.

Disadvantages: Fixed size, prone to overflow.



2.1.4 Stack Implementation Methods

Array-based Implementation

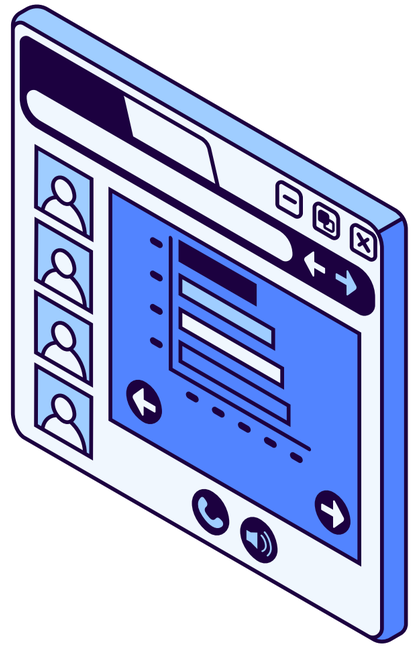


2.1.4 Stack Implementation Methods

Linked List-based Implementation

Principle: Each element is a node consisting of: data + next pointer (top = null initially).

- Push: Create a new node → point to current top → update top.
- Pop: Update top = top → next.
- Peek: Return top.data.

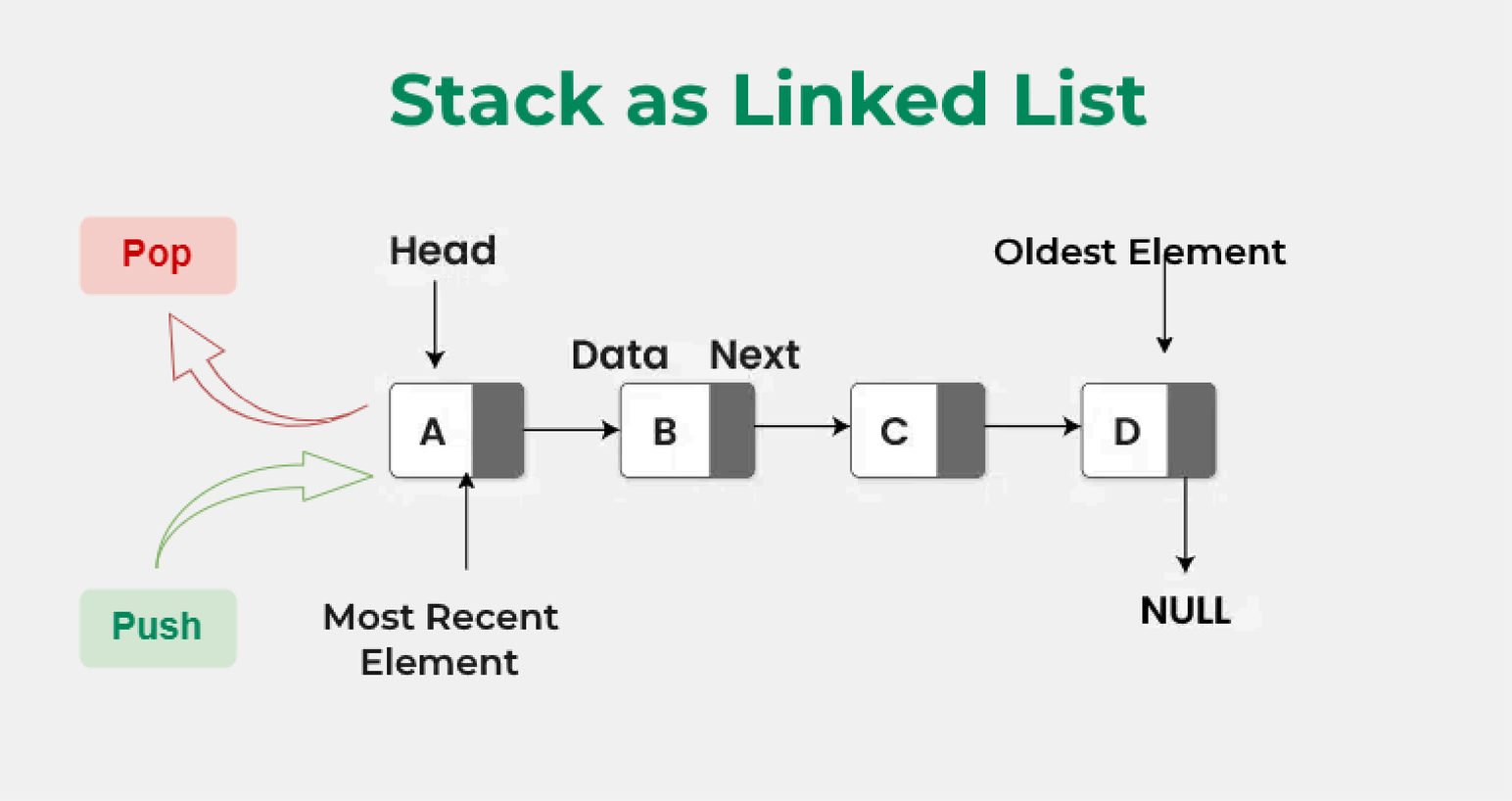


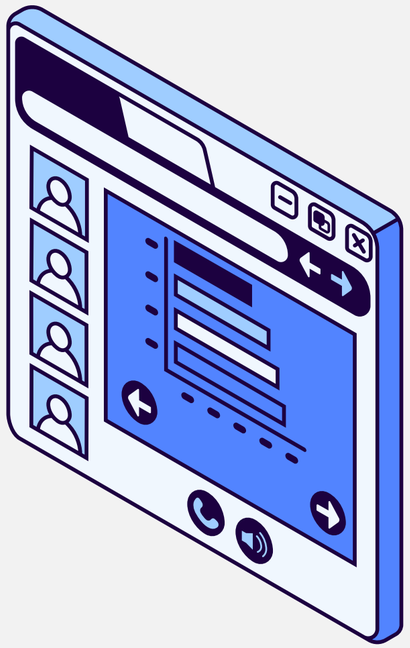
Advantages: Flexible, dynamic size.

Disadvantages: Extra memory for pointers, poor locality.

2.1.4 Stack Implementation Methods

Linked List-based Implementation





2.1.4.3 Comparison of Two Stack Implementation Methods

Criteria	Array-based	Linked List-based
Advantages	Simple implementation. Fast top access (good cache locality).	Dynamic size, flexible. No stack overflow (except when system memory is exhausted).
Disadvantages	Fixed size, risk of overflow or waste. Resizing is costly.	Extra memory needed for pointers. Poor locality, possibly slower access.
Best Use Cases	When the maximum number of elements is known in advance and high access speed is required.	When the number of elements changes frequently and can be large, requiring flexibility while accepting extra memory cost.

2.1.5 Complexity Analysis

2.1.5.1 Time Complexity

- Push, Pop, Peek, isEmpty: O(1) (directly at the top).
- Dynamic Array: Push is usually O(1), but resizing → O(n) (amortized O(1)).
- Linked List: Also O(1), but practically slower due to memory overhead and poor locality.

2.1.5 Complexity Analysis

Time Complexity

Operation	Fixed-size Array	Dynamic Array	Linked List
Push	O(1)	O(1)/O(n)	O(1)
Pop	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)

2.1.5 Complexity Analysis

2.1.5.2 Space Usage

Feature	Fixed-size Array	Dynamic Array	Linked List
Memory Allocation	Pre-allocated array with size = maxSize	Allocated proportional to the number of elements, with extra “buffer”	Each element is a node consisting of: data + next pointer
Storage Complexity	Θ(maxSize) (reserves all memory in advance)	Θ(n), resizing when full	Θ(n) for data + Θ(n) for pointer overhead
Capacity Limit	Fixed by maxSize	Flexible, no need for isFull	Unlimited, only limited by available RAM

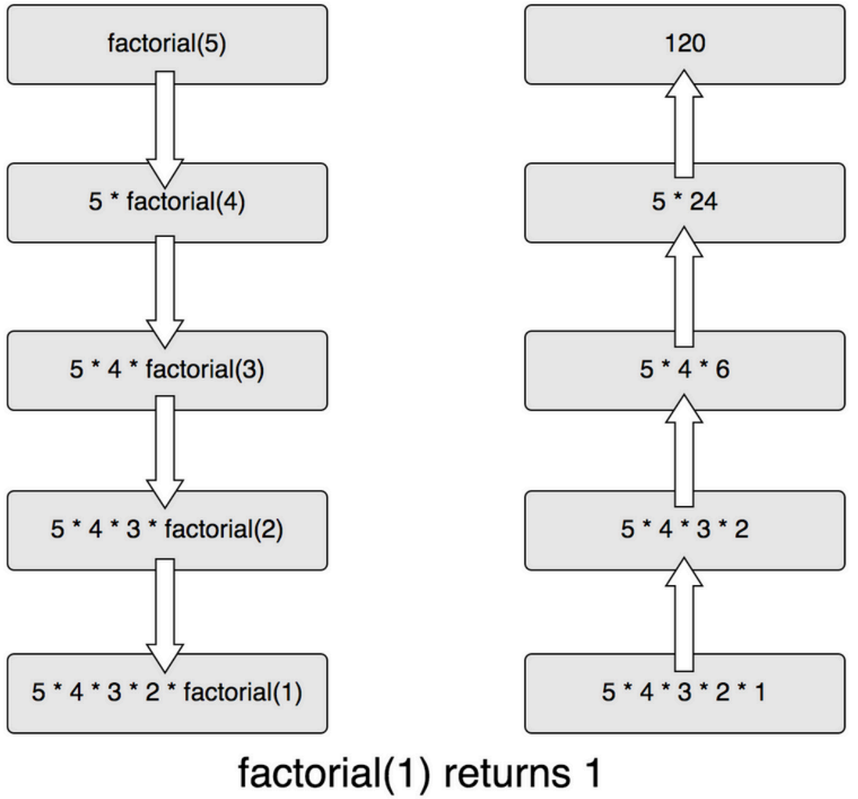
2.1.5 Complexity Analysis

2.1.5.2 Evaluation Methods

Method	Approach	Advantages	Disadvantages
Experimental	Implement and run programs, measure execution time on different datasets	Concrete and practical results. Reflects actual runtime.	Dependent on hardware and programming language. Hard to choose fully representative datasets. Costly in time and resources.
Asymptotic	Theoretical analysis using Big-O notation to express complexity as input size grows large	Independent of hardware and language. Easy to compare algorithms. No implementation needed. Good for large datasets.	Does not provide exact runtime. Ignores constants. Less accurate for small datasets.

2.2.1 Recursive Function Management

- When a function is called → a new frame is pushed onto the Call Stack (parameters, variables, return address).
- When the function ends → the frame is popped → execution returns to the previous position.
- Recursion works by using multiple independent frames.
- Too deep recursion → Stack Overflow.



2.2.2 Expression Conversion and Evaluation

1. Concepts

Infix: Operator is placed between operands.

- Example: A + B * C

Postfix: Operator comes after operands.

- Example: A B C * +

Prefix: Operator comes before operands.

- Example: + A * B C

2.2.2 Expression Conversion and Evaluation

2. Problem

Infix expressions depend on operator precedence and parentheses.

- A + B * C → B * C must be evaluated first.
- (A + B) * C → A + B must be evaluated first.

Computers find infix expressions difficult to process

→ need to convert to Postfix/Prefix for simpler and consistent evaluation.

2.2.2 Expression Conversion and Evaluation

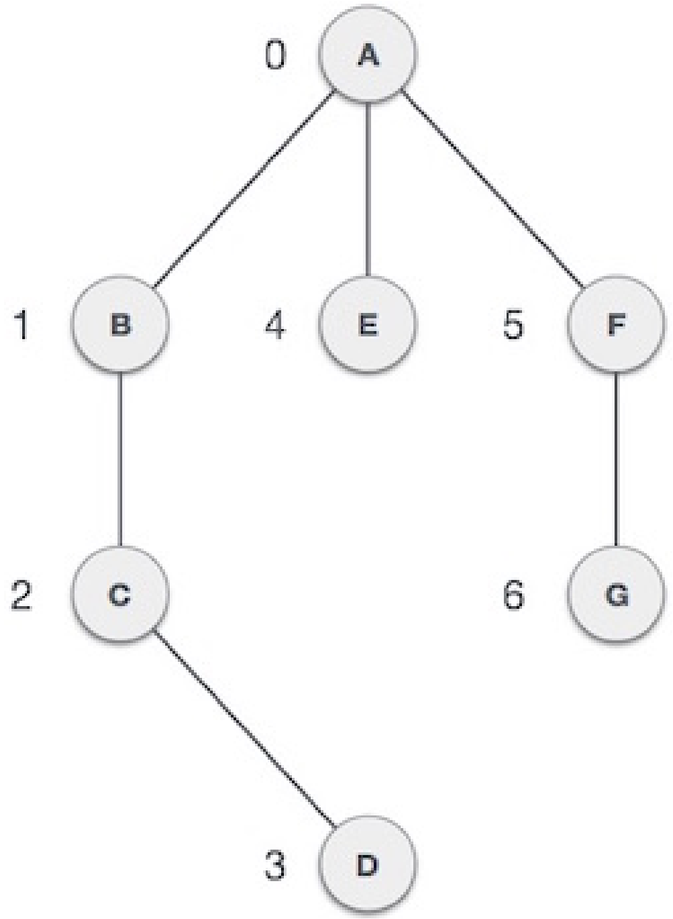
Solution: Using the Shunting-yard Algorithm (Edsger Dijkstra)

- Read the infix expression from left to right.
- If an operand is encountered → append it to the result.
- If an operator is encountered:
  - Compare its precedence with operators in the stack.
  - If higher precedence → push it onto the stack.
- If an opening parenthesis is encountered → push onto the stack.
- If a closing parenthesis is encountered → pop operators to the result until an opening parenthesis is found.
- After reading the entire expression → pop all remaining operators in the stack to the result.

2.2.3 DFS & Backtracking Algorithm

DFS (Depth-First Search):

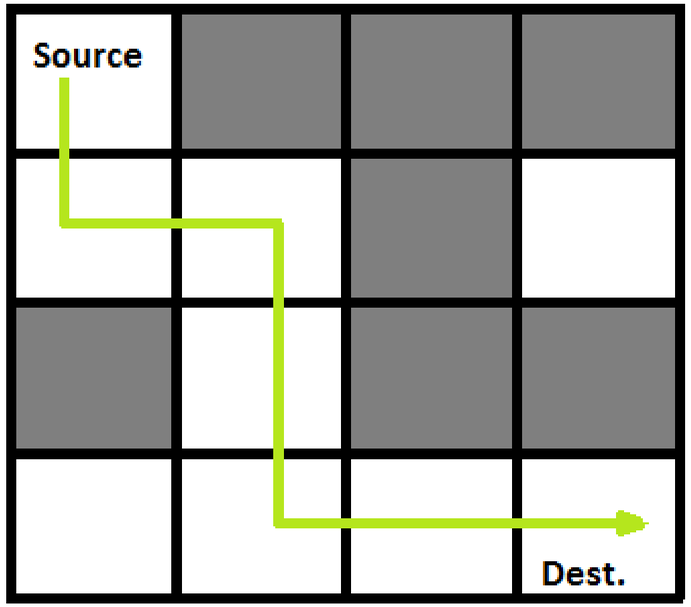
- Traverse deepest first, then backtrack.
- Can be implemented using recursion or a stack.
- Mainly used for graph or tree traversal/search.



2.2.3 DFS & Backtracking Algorithm

Backtracking:

- Based on DFS but includes constraint checking.
- When a wrong path is taken → backtrack and try another branch.
- Commonly used for: Sudoku, mazes, combinations, permutations.

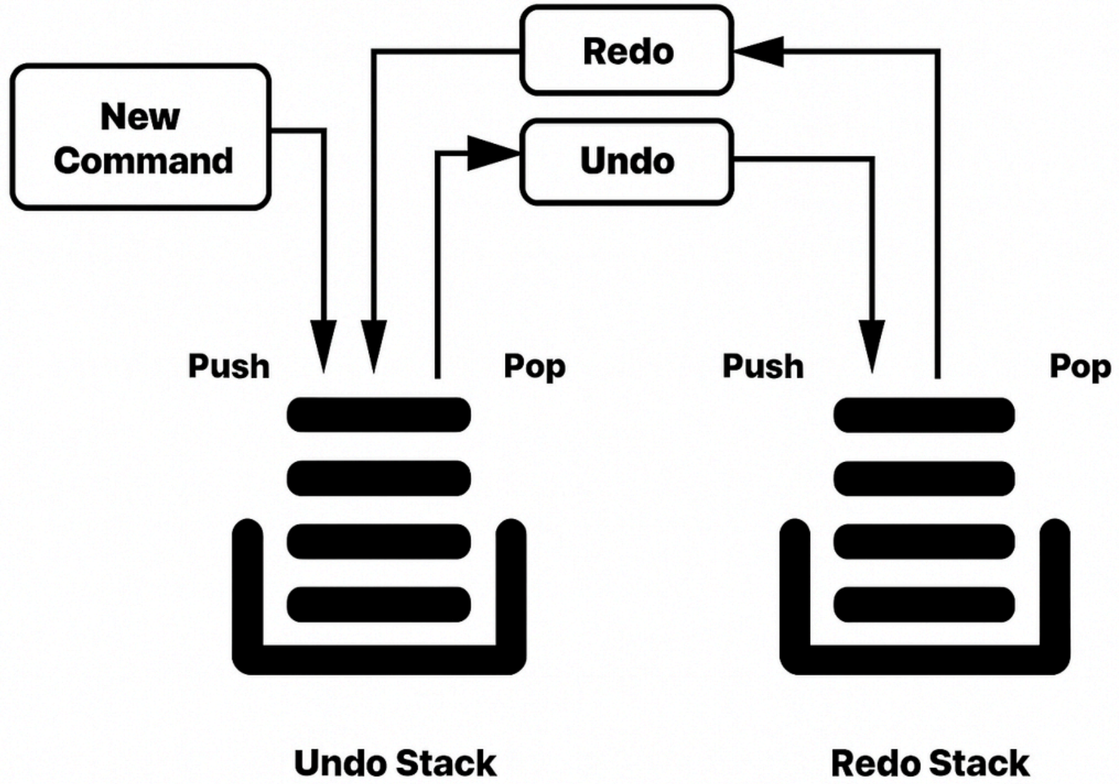


2.2.4 Real-life Applications

Undo/Redo

In text editors, image editors, IDEs… systems often use two stacks:

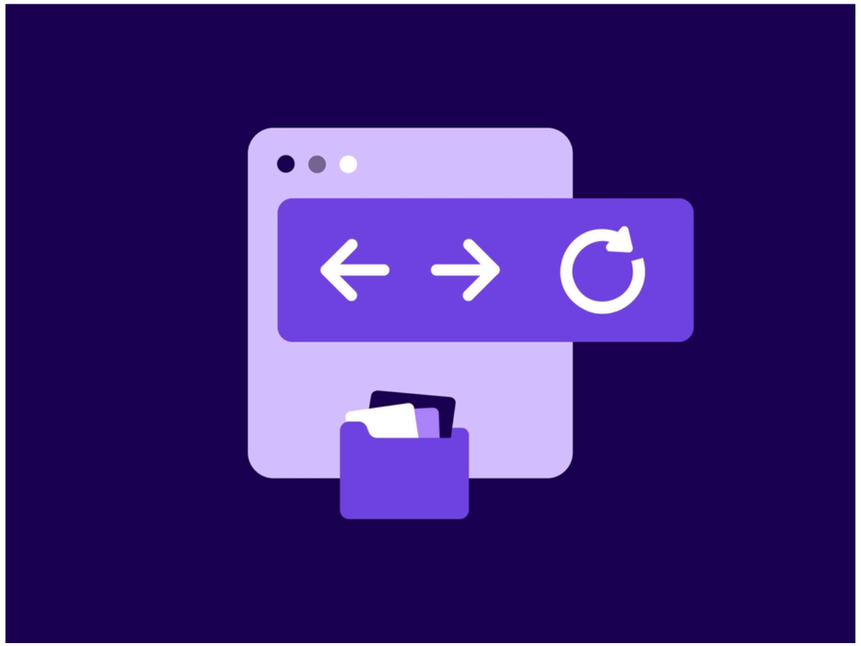
- Undo Stack: stores recent actions (each new action is pushed here).
- Redo Stack: stores actions that have been undone.



2.2.4 Real-life Applications

Web Browsing History

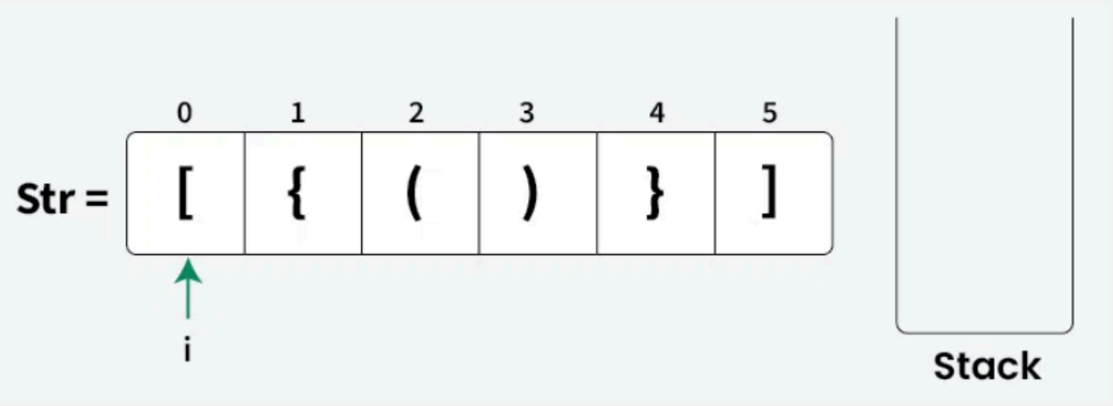
- Browsers use two stacks to manage Back and Forward buttons:
  - Back Stack: stores previously visited pages.
  - Forward Stack: stores pages navigated back from the Back Stack.
- How it works:
  - When opening a new page → current URL is pushed onto the Back Stack.
  - When pressing Back → URL is popped from Back Stack and pushed to Forward Stack.
  - When pressing Forward → URL is popped from Forward Stack and reopened.



2.2.4 Real-life Applications

Bracket Checking in Expressions

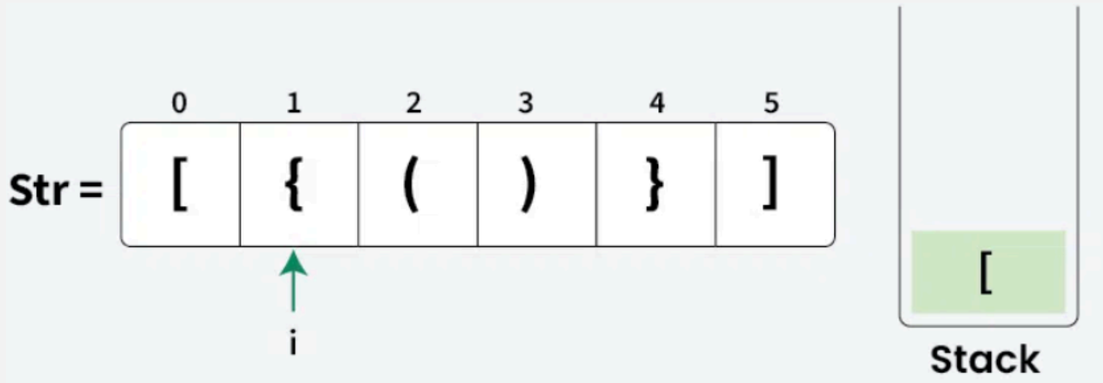
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

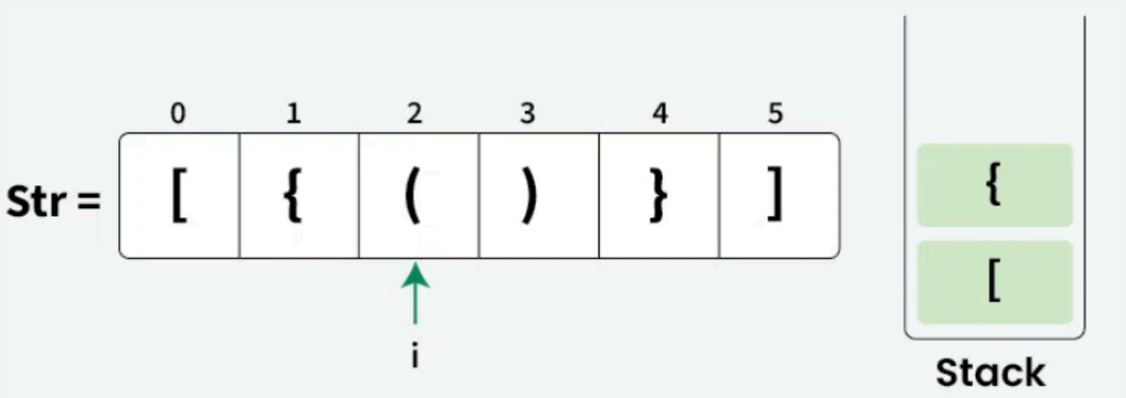
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

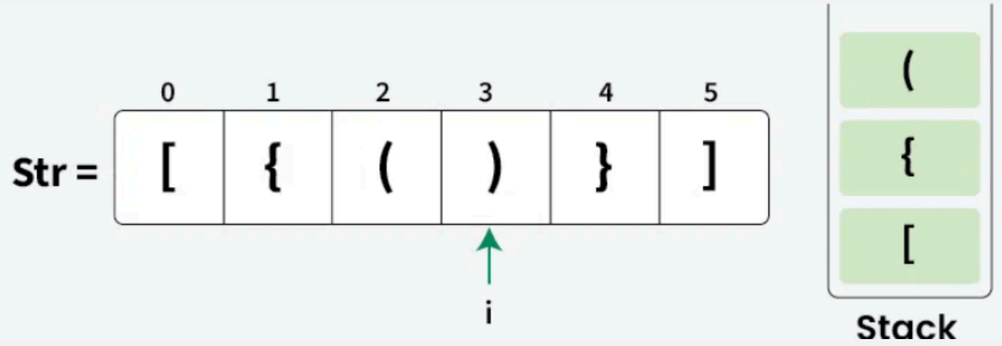
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

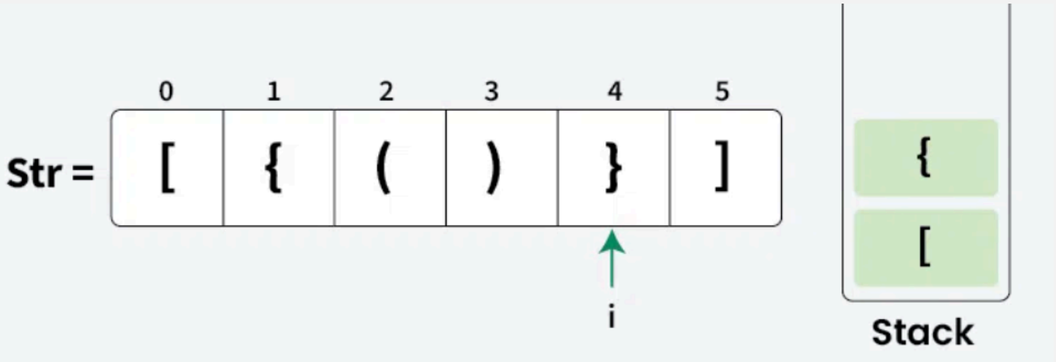
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

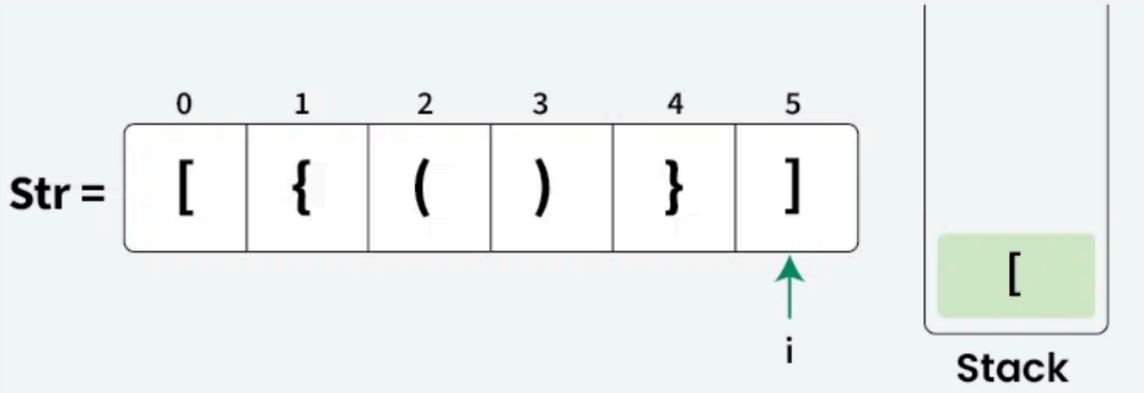
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

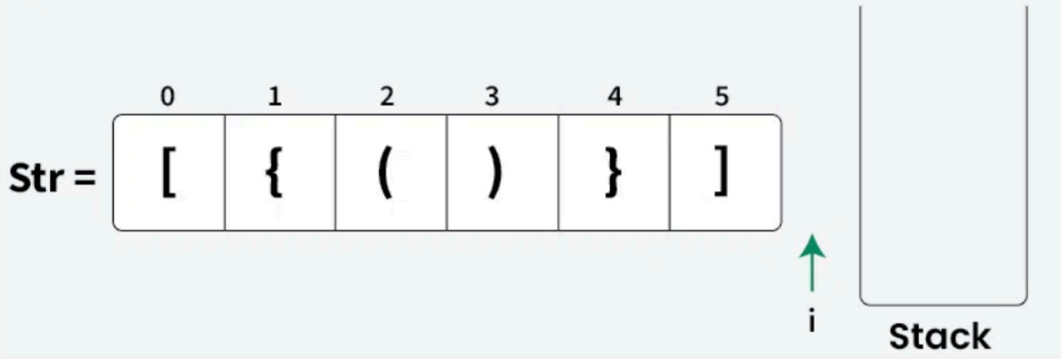
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Bracket Checking in Expressions

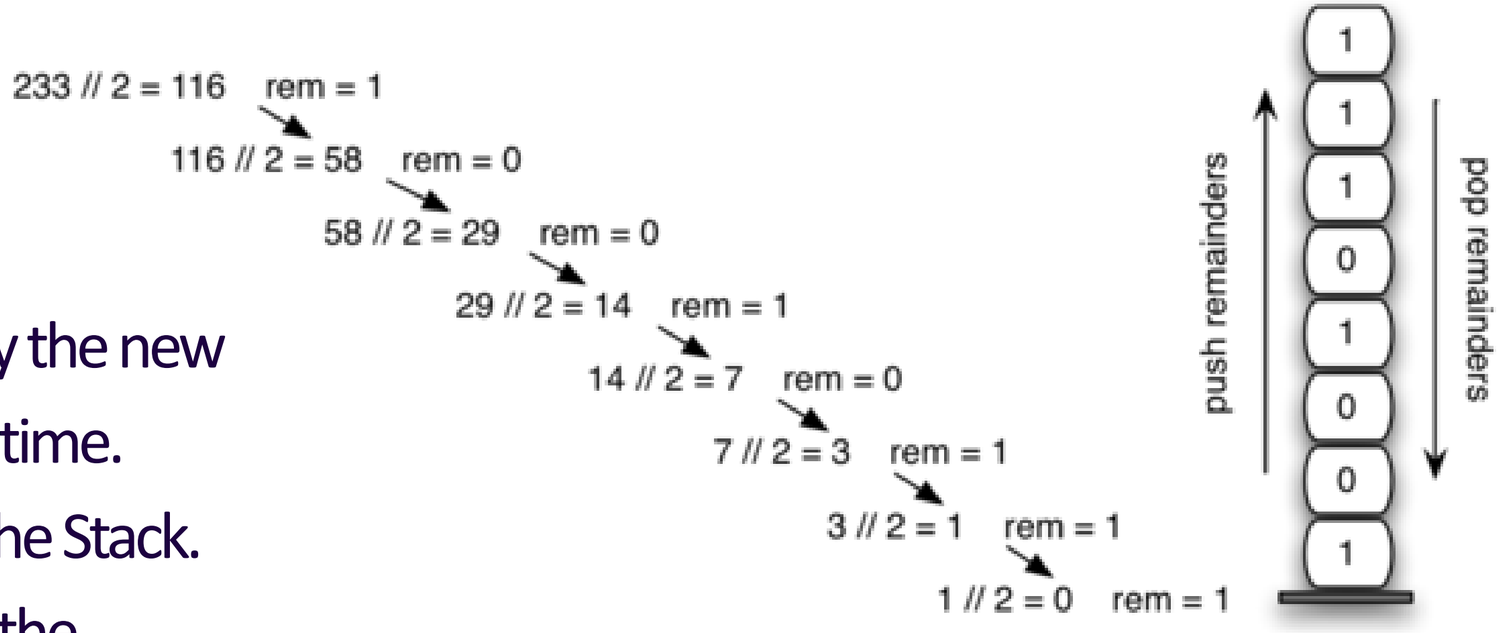
- How it works:
  - When encountering an opening bracket (, {, [→ push into the Stack.
  - When encountering a closing bracket), },] → pop from the Stack and check if it matches the most recent opening bracket.
  - If it matches → continue; if it does not match or the Stack is empty → the expression is invalid.



2.2.4 Real-life Applications

Base conversion:

- How it works:
  - Repeatedly divide the number by the new base, taking the remainder each time.
  - Each remainder is pushed onto the Stack.
  - After the division reaches 0, pop the elements from the Stack in reverse order to form the number in the new base.



2.2.5 Advanced Applications

Applications in Research:

- **Natural Language Processing (NLP):** The Shift-reduce parsing algorithm uses a stack to construct syntax trees.
- **Machine Learning:** Reverse-mode differentiation and backpropagation rely on stacks to store computations and traverse backward to compute gradients.
- **Machine Vision:** Algorithms such as flood-fill and region-growing utilize stacks in image processing tasks.

2.2.6 Implementation Examples in Programming Languages

C++ with array-based implementation and the std::stack library.

```
#include <iostream>
#include <stack>

int main() {
   stack<int> s;            // Initialize an empty stack of integers
   s.push(10);              // Push element 10 onto the top of the stack
   s.push(20);              // Push element 20 onto the top of the stack
   s.push(30);              // Push element 30 onto the top of the stack

   cout << s.top() << " popped\n"; // Print the top element (30)
   s.pop();                        // Pop the top element (30)
   cout << "Top element is: " << s.top() << "\n"; // Peek the current top
element (20) without removing it

   cout << "Elements present in stack: ";
   while (is.empty()) {       // Traverse and print the remaining elements in
the stack
      cout << s.top() << " "; // Print the top element
      s.pop();                // Pop the printed element
   }
   cout << "\n";
   return 0;
}
```

2.2.6 Implementation Examples in Programming Languages

Python with list

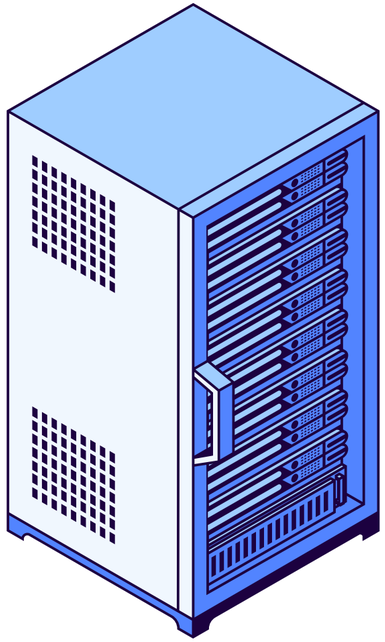
```
def create_stack():
    return []
def is_empty(stack):
    return len(stack) == 0
def push(stack, item):
    stack.append(item)
    print("pushed item:", item)
def pop(stack):
    if is_empty(stack):
        return "stack is empty"
    return stack.pop()

# Test
stack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)

print("popped item:", pop(stack))
print("stack after popping:", stack)
```

2.2.6 Implementation Examples in Programming Languages

Python with collections.deque

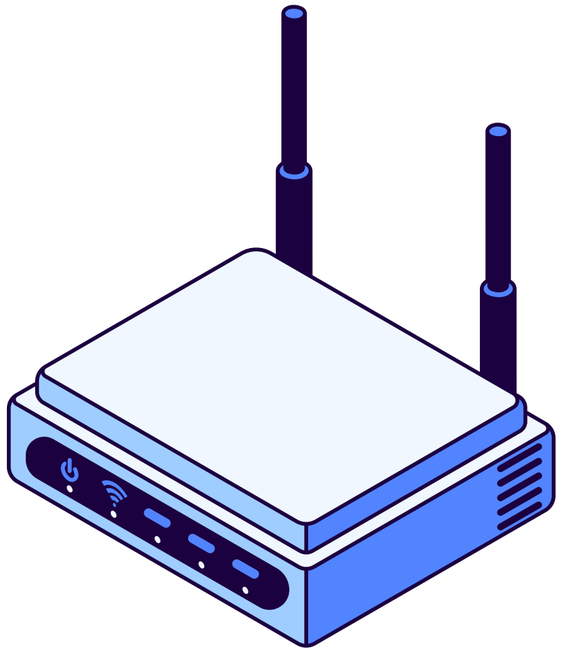
```
from collections import deque

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

# Teststack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)

print("popped item:", pop(stack))
print("stack after popping:", stack)
```

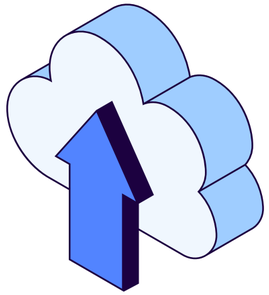
CONCLUSION



Achieved results

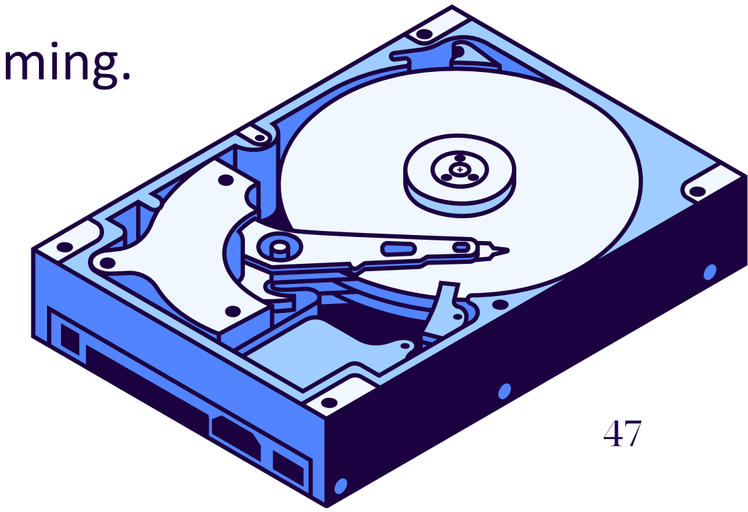
- Understand the theory of stack and the LIFO principle.
- Be proficient in operations such as: push, pop, peek/top, IsEmpty, and IsFull.





Future development directions

- Applications in embedded systems
- Effectively supports artificial intelligence applications, big data processing, and parallel programming.




Thank you!