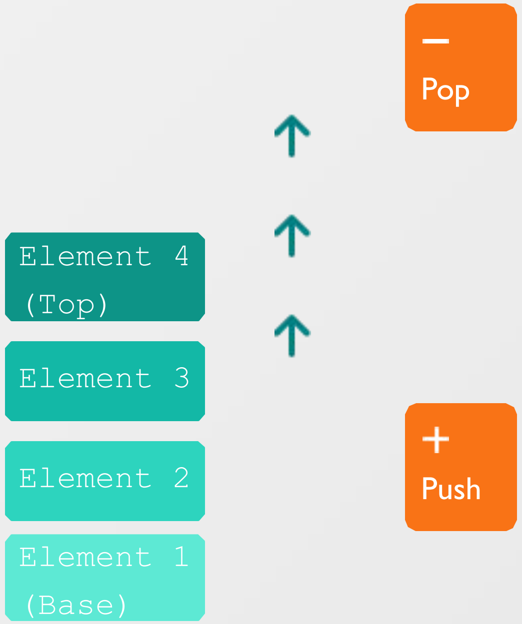
Data Structures: Understanding Stacks

A Comprehensive Exploration of Stack Data Structures



What You'll Learn

- Fundamentals of Last-In-First-Out (LIFO) principle
- Core operations: Push, Pop, Peek, and utility functions
- Array and linked list implementations in Python, Java, and C++
- Real-world applications in function calls, undo mechanisms, and more
- Pros, cons, and advanced problem-solving techniques

September 4, 2025

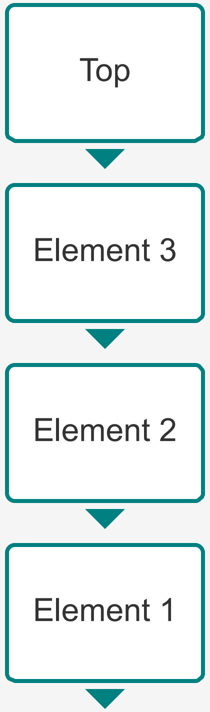
Computer Science & Engineering

What is a Stack?

Definition

A stack is a linear data structure that follows a specific order for operations: the last element inserted is the first one to be removed. This principle is known as Last In, First Out (LIFO).

Imagine a stack of plates: you always add a new plate to the top, and when you want a plate, you take the one from the top first.



Single access point (Top)

Key Characteristics

Linear Structure
Elements are arranged sequentially one after another.

Single Access Point
All operations (insertion and deletion) happen at one end, the "top."

LIFO Principle
The most recently added element is the first to be retrieved.

Real-world example
A shuttlecock box: The last shuttlecock inserted is the first one taken out, as both operations happen from the same end.

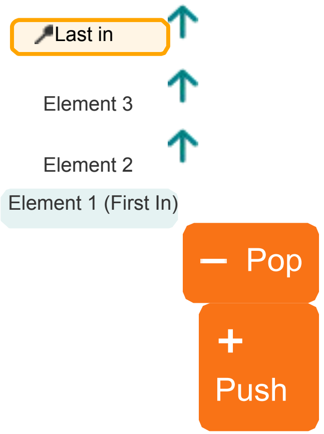
LIFO Principle Explained

What is LIFO?

Last In, First Out (LIFO) is the defining characteristic of a stack. It dictates that the element that was most recently added to the stack will be the first one to be removed.

In computing, new elements are always "pushed" onto the top of the stack, and removal, or "popping," also occurs exclusively from the top.

How LIFO Works



Real-world Examples

Stack of Plates

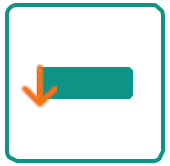
The last plate placed on top is the first one you pick up.

Plate 3
Plate 2
Plate 1

You always take from the top

Shuttlecock Box

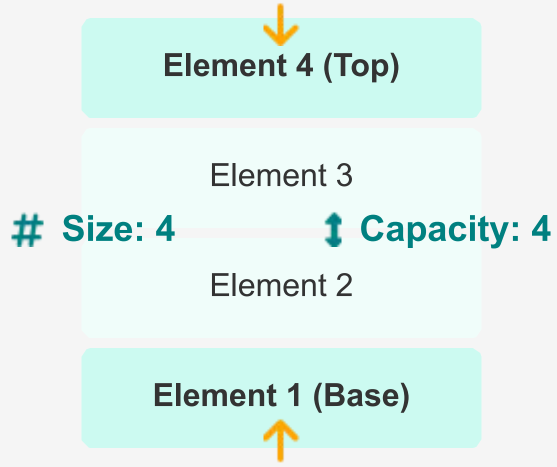
The last shuttlecock inserted is the first one taken out, as both operations happen from the same end.



Insert and remove from same end

Key Insight: LIFO is about restricting access to one end of the data structure, ensuring that the most recent operation is the first to be reversed.

Basic Stack Terminology



For array-based implementations, capacity refers to the maximum number of elements the stack can hold.

Top
The position of the most recently inserted element. Both insertions (push) and deletions (pop) are always performed at the top of the stack.

Base
The position of the very first element inserted, which remains at the bottom of the stack.

Size
The current number of elements present in the stack. This changes as elements are pushed and popped.

Capacity
The maximum number of elements the stack can hold, particularly relevant for fixed-size implementations. Attempting to push onto a full stack results in an OverflowError.

Core Stack Operations: Push

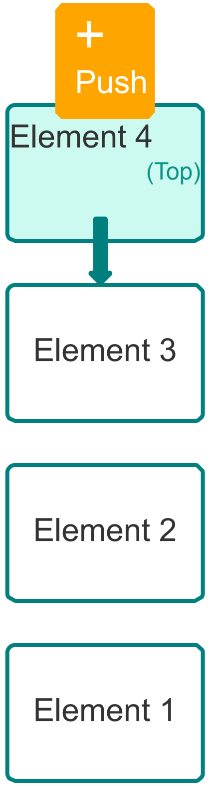
What is Push?

The `Push` operation adds a new element to the top of the stack. This operation makes the new item the topmost element of the stack, adhering to the Last-In, First-Out (LIFO) principle.

Algorithm for Push

1. **Check if the stack is full**
   Attempting to push onto a full stack leads to an OverflowError.
2. **Increment the top index**
   This makes room for the new element at the top of the stack.
3. **Add the new element at the new top position**
   This element is now the most recently added item on the stack.

Visual Representation



Example

The "Undo" feature in a text editor is a perfect example. Each action you take (like typing or deleting) is pushed onto a stack. This ensures the last action you performed is the first one to be undone.

Common Pitfall

Attempting to pop from an empty stack results in an **UnderflowError**. Always check if the stack is empty before performing a pop operation.

Core Stack Operations: Pop

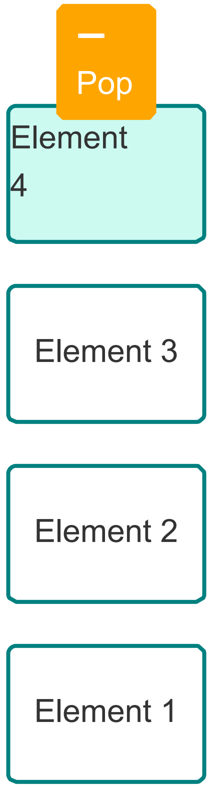
Definition

The **Pop** operation removes and returns the topmost element from the stack. This operation always targets the element currently at the "top" of the stack, adhering to the Last-In, First-Out (LIFO) principle.

Algorithm for Pop

1. **Check if the stack is empty**
   Attempting to pop from an empty stack leads to an UnderflowError
2. **If not empty, retrieve the top element**
   This is the element that was most recently added
3. **Decrement the top index**
   This effectively removes the element from the stack

Visual Representation



Example

When closing browser tabs, the most recently opened tab is "popped" off first. This ensures that you are always interacting with the most recent item.

Common Pitfall

Attempting to pop from an empty stack results in an **UnderflowError**. Always check if the stack is empty before performing a pop operation.

Core Stack Operations: Peek

What is Peek?

The **Peek** operation allows you to **retrieve the topmost element** of the stack without removing it.

This is particularly useful when you need to inspect the next item to be processed or evaluated without altering the stack's state.

How Peek Works

Top Element

Element 3

Element 2

Element 1

Current Top (without removing)

Algorithm for Peek

1. Check if the stack is empty
2. If not empty, retrieve the top element
3. Return the top element without modifying the stack

Practical Example

Calculator Application: The stack holds operators during expression evaluation.

The app **"peeks"** at the top of the stack to determine the next step based on operator precedence, ensuring correct calculations.

Stack Utility Functions

Beyond the core `Push`, `Pop and Peek:`

isEmpty ✅

The **isEmpty** operation checks if the stack contains any elements. It returns *true* if the stack is empty and *false* otherwise. This is the most critical utility, as it's used to prevent UnderflowErrors by ensuring you don't try to *pop* from or *peek* at an empty stack.

isFull 🛑

For stacks with a fixed capacity (like those implemented with a simple array), the i**sFull** operation checks if the stack has reached its maximum size. It returns *true* if no more elements can be added. This is essential for preventing **OverflowErrors** before attempting a *push* operation.

size / count 📏

This utility, often called **size** or **count**, returns the total number of elements currently in the stack. It's useful for monitoring the state of the stack or for algorithms that need to iterate through its contents. Think of it as asking, "How many plates are in the pile?"

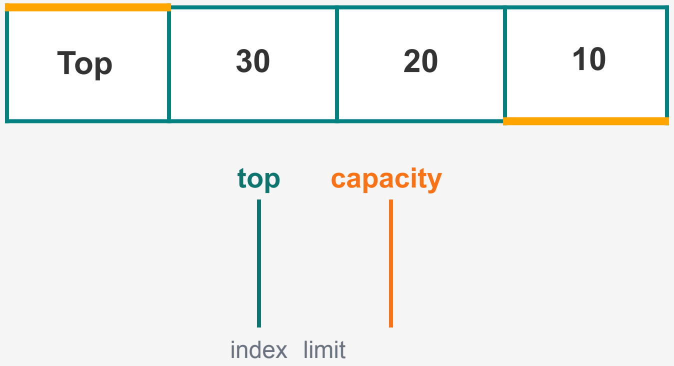
search 🔎

The **search** operation looks for a specific item in the stack and returns its distance from the top. If the item is at the top of the stack, it returns 1; if it's the next one down, it returns 2, and so on. If the item isn't found, it typically returns -1. While useful for debugging, this operation is less common because it inspects elements below the top, which slightly goes against the strict LIFO principle.

Array-Based Stack Implementation

How Arrays Work as Stacks

Stacks can be efficiently implemented using arrays, where elements are stored in contiguous memory locations. This approach is akin to organizing books on a shelf with a fixed number of slots; elements are added and removed from one end, typically referred to as the "top" of the stack.



Array-based stack with fixed capacity

Advantages

Fast Indexing
Elements can be accessed directly using their index, leading to fast push and pop operations with O(1) time complexity.

Predictable Memory Layout
Memory allocation is straightforward and consistent, beneficial in environments with predictable memory usage.

Memory Efficient
Array elements do not require additional memory for pointers, unlike linked lists.

Disadvantages

Fixed Size
Array-based stacks have a predefined capacity. If the number of elements exceeds this capacity, an OverflowError occurs.

Inflexibility
The fixed size limits the stack's ability to grow or shrink dynamically based on demand.

Resize Cost
Resizing a fixed-size array is computationally expensive as it involves creating a new, larger array and copying all existing elements.

Array Stack: Python Implementation

```python
# Array-based stack implementation in Python

class ArrayStack:
    # Initialize the stack with a fixed capacity
    def __init__(self, capacity):
        self.stack = []
        self.capacity = capacity

    # Push an item onto the stack
    def push(self, item):
        if len(self.stack) == self.capacity:
            raise OverflowError("Stack Overflow: Stack is full")
        self.stack.append(item)
        print(f"Pushed: {item}")

    # Pop an item from the stack
    def pop(self):
        if not self.stack:
            raise IndexError("Stack Underflow: Stack is empty")
        popped_item = self.stack.pop()
        print(f"Popped: {popped_item}")
        return popped_item
```

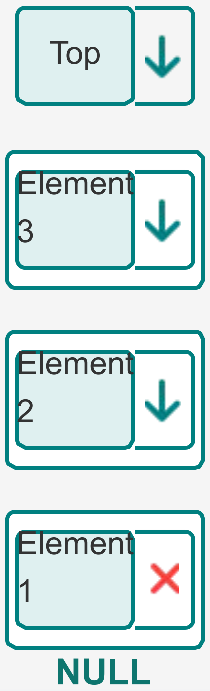
```python
    # Peek at the top item without removing it
    def peek(self):
        if not self.stack:
            raise IndexError("Stack is empty")
        return self.stack[-1]

    # Check if the stack is empty
    def isEmpty(self):
        return len(self.stack) == 0

    # Check if the stack is full
    def isFull(self):
        return len(self.stack) == self.capacity

    # Example usage
if __name__ == "__main__":
        my_stack = ArrayStack(3)
        my_stack.push(10)
        my_stack.push(20)
        my_stack.push(30)
        # my_stack.push(40)
        # This would raise an OverflowError
        my_stack.pop()
        print(f"Top element: {my_stack.peek()}")
```

Linked List-Based Stack Implementation

Implementation Concept

A linked list implementation represents the stack as a series of interconnected nodes, where each node contains data and a pointer to the next node.

The "top" of the stack is typically the head of the linked list, allowing for efficient additions and removals.



Advantages

Dynamic Resizing

Linked list stacks can grow and shrink as needed, without a predetermined size. Memory is allocated only when new elements are added.

Efficient Memory Usage

Memory is allocated on demand, avoiding the potential wastage of pre-allocated but unused space in fixed-size arrays.

Disadvantages

Pointer Overhead

Each node requires additional memory to store a pointer to the next node, making it less space-efficient than array-based implementations.

Slightly Slower Operations

Manipulating pointers for push and pop operations can introduce a small performance cost compared to direct indexing of arrays.

Comparison: Linked list implementation is more flexible than array-based, but may be less efficient for certain operations.

Linked List Stack: Python Implementation

```
# Node class to represent each element in the
stack class Node:
    def init   (self, data):
        self.data = data
        self.next = None

# Linked list stack
implementation class
LinkedListStack:
    def init   (self):
        self.top =
        None self.size
        = 0
    # Push operation: Add element to the
    top def push(self, item):
        new_node = Node(item)
        new_node.next =
        self.top self.top =
        new_node
        self.size += 1
        print(f"Pushed: {item}")

    # Pop operation: Remove element from the
    top def pop(self):
        if self.isEmpty():
            raise IndexError("Stack Underflow: Stack is
        empty") popped_item = self.top.data
        self.top =
        self.top.next self.size
```

```
    # Peek operation: View the top element
    def peek(self):
        if self.isEmpty():
            raise IndexError("Stack is empty")
        return self.top.data

    # Utility function: Check if stack is empty
    def isEmpty(self):
        return self.top == None

    # Utility function: Get stack size
    def size(self):
        return self.size

# Example usage
if   name  == " main ": stack =
    LinkedListStack()
    stack.push(10)
    stack.push(20)
    stack.push(30) stack.pop()
    print(f"Top element: {stack.peek()}")
```

Java Implementation of Stacks

```java
// Array-based stack implementation in Java
class ArrayStack {
    private int[] arr;
    private int top;
    private int capacity;
    // Constructor to initialize the stack with a given capacity
    public ArrayStack(int size) {
        arr = new int[size];
        capacity = size;
        top = -1; // Indicates an empty stack
    }
    // Adds an element to the top of the stack
    public void push(int x) {
        if (isFull()) {
           System.out.println("Stack Overflow: Cannot add element, stack is full")
           return;
         }
         arr[++top] = x;
         System.out.println("Inserting " + x);
    }
    // Removes and returns the top element from the stack
    public int pop() {
        if (isEmpty()) {
           System.out.println("Stack Underflow: Cannot pop from an empty stack."
             return -1;
         }
         int poppedElement = arr[top--];
         System.out.println("Popped " + poppedElement);
         return poppedElement;
     }
```

Key Features

Fixed Capacity

The stack has a predefined maximum size (capacity). If more elements are pushed than the stack can hold, it leads to a stack overflow error.

Error Handling

The implementation includes checks for overflow (push on full stack) and underflow (pop on empty stack) conditions.

Zero-based Indexing

The "top" variable represents the index of the top element. When the stack is empty, top is -1, indicating no elements are present.

Example Usage

```java
ArrayStack stack = new ArrayStack(3);
stack.push(10);
stack.push(20);
stack.push(30);
stack.pop(); // Outputs: Popped 30
stack.peek(); // Outputs: 20
```

```java
// Returns the top element without removing it
public int peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty: No element to peek.");
        return -1;
    }
    return arr[top];
}
// Checks if the stack is empty
public boolean isEmpty() {
    return top == -1;
}
// Checks if the stack is full
public boolean isFull() {
    return top == capacity - 1;
}
// Returns the number of elements in the stack
public int size() {
    return top + 1;
}
// Prints the elements of the stack from bottom to top
public void printStack() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
        return;
    }
    System.out.print("Stack elements: ");
    for (int i = 0; i <= top; i++) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}
```

```java
    // Main method for testing the stack implementation
    public static void main(String[] args) {
        ArrayStack stack = new ArrayStack(3);
        System.out.println("Is stack empty? " + stack.isEmpty());

        stack.push(100);

        stack.push(200);
        stack.push(300);
        stack.printStack();

        stack.push(400); // Stack Overflow

        System.out.println("Top element: " + stack.peek());
        System.out.println("Stack size: " + stack.size());

        stack.pop();
        stack.printStack();
    }
}
```

Real-World Applications: Function Call Management

How Stacks Manage Function Calls

- **Function Call:** When a function is called, its local variables, parameters, and return address are **pushed** onto the call stack.
- **Function Return:** When a function completes, its information is **popped** off the stack, and execution returns to the previous function.
- **Nested Calls:** Each new function call creates a new stack frame that contains all the information needed to resume execution when the function returns.
- **Recursion:** Recursive functions rely on the stack to keep track of multiple simultaneous executions, each with its own set of variables.

Real-World Example: Airbnb

Airbnb's pricing algorithms utilize recursive calls to analyze:

- Historical data
- User preferences
- Property characteristics

Call stacks are essential for managing these complex calculations to ensure accuracy across millions of properties.

Function Call Stack Visualization

main()
Return address: None

calculatePricing()
Return address: main()

analyzeData()
Return address: calculatePricing()

getUserPrefs()
Return address: analyzeData()

Stack Frame Contents:

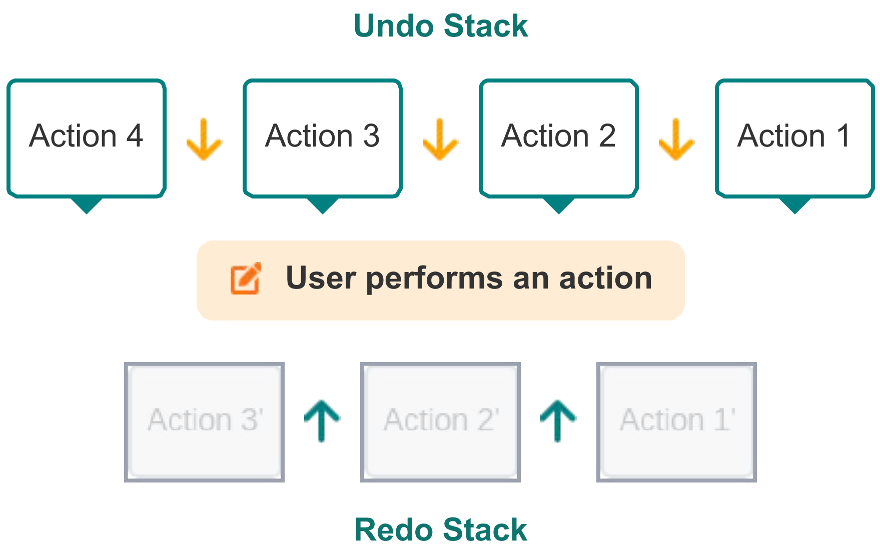
Function name · Parameters · Return address

When a function returns, the stack unwinds one level, resuming execution where it left off.

Real-World Applications: Undo/Redo & Browser History

Undo/Redo Functionality

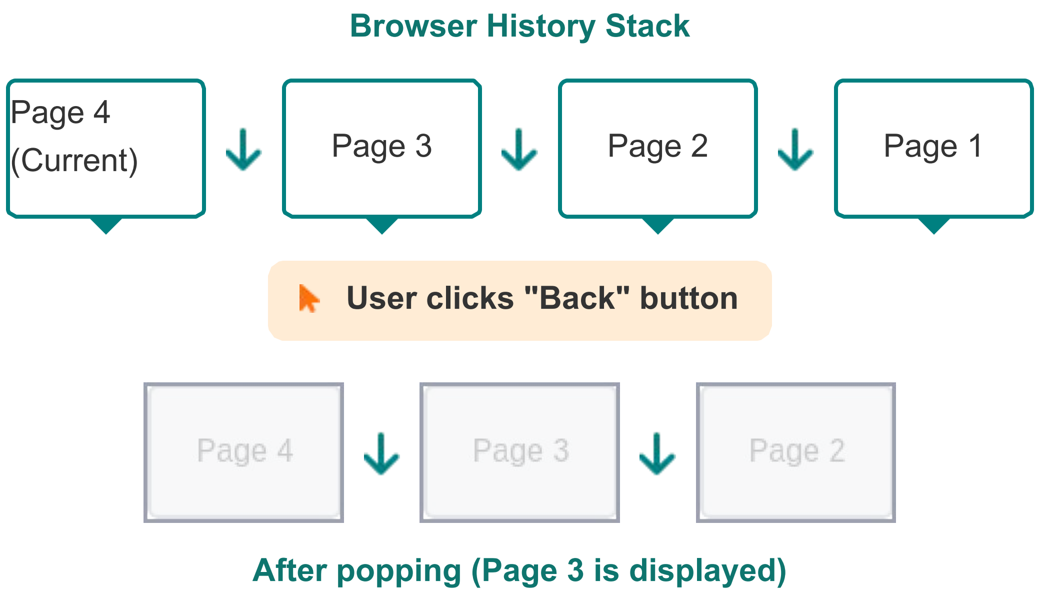
Many applications implement undo/redo using stacks. Each action is pushed onto an "undo" stack.



Example:
Microsoft Word uses a stack-based mechanism to manage user actions, allowing users to revert to previous states.

Browser History Management

Web browsers use a stack to manage visited pages. Each new page is pushed onto the stack.



Key Insight:
The "back" button pops the current page, displaying the previous page (now at the top of the stack).

Real-World Applications: Expression Evaluation

How Stacks Enable Expression Evaluation

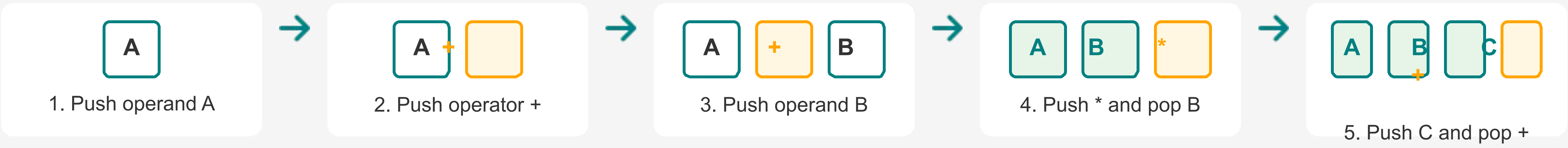
Stacks are vital for parsing and evaluating mathematical expressions, particularly for converting between infix (e.g., A + B * C), prefix, and postfix (e.g., ABC+*) notations. They help manage operator precedence during evaluation.

Infix to Postfix Conversion Example

A + B * C
Infix Expression

→

A B C * +
Postfix Expression



Practical Application

Calculator applications use stacks to hold operators and operands during expression evaluation, peeking at the top of the stack to determine the next step based on operator precedence.

Evaluation Process

During evaluation, the stack helps manage operator precedence and associativity rules, ensuring expressions are evaluated in the correct order.

Real-World Applications: Syntax Validation

How Stacks Validate Syntax

Compilers and code editors use stacks to validate the syntax of programming languages, ensuring that parentheses, brackets, and braces are correctly matched and nested.

- **Opening symbols** are pushed onto the stack
- **Closing symbols** pop the corresponding opening symbol
- **Mismatches** or **unclosed symbols** flag errors

Real-world example

GitHub's editor employs stacks to validate syntax in real-time, helping developers catch errors as they code.

Parentheses Matching Process

(→ [→ { → } →] →)

}	]	)
Pops {	Pops [	Pops (

Valid expression: All symbols are properly matched

Example: Validating parentheses in code

```javascript
function validateSyntax() {
  if (expression.length % 2 !== 0) {
    return false; // Early exit for odd-length expressions
  }
  const stack = [];
  const pairs = {
    ')': '(',
    ']': '[',
    '}': '{'
  };
  for (let char of expression) {
    if (char === '(' || char === '[' || char === '{') {
      stack.push(char);
    } else if (pairs[char]) {
      if (stack.pop() !== pairs[char]) {
        return false; // Mismatch found
      }
    }
  }
  return stack.length === 0; // True if all symbols are matched
}
```

Advantages of Stacks

Simplicity

Stacks are conceptually straightforward and relatively easy to implement using arrays or linked lists. Their operations are simple to understand and code.

Efficient Operations

Primary operations—push, pop, and peek—typically have a time complexity of O(1) (constant time), making them very fast regardless of stack size.

LIFO Principle

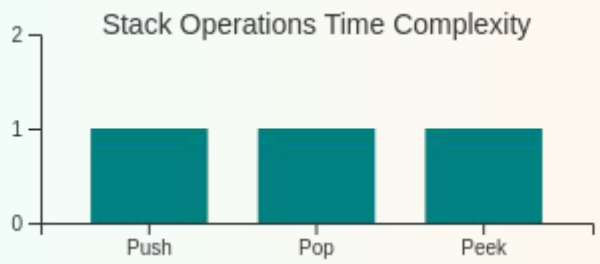
The Last-In, First-Out behavior is inherently useful for tasks requiring elements to be processed in reverse order of arrival, such as undo mechanisms.

Automatic Memory Management

In programming, the call stack automatically handles memory allocation and deallocation for function calls, simplifying memory management for developers.

Memory Efficiency

Array-based stacks can be memory-efficient as they use contiguous memory blocks and do not incur the overhead of pointers associated with linked lists.



Time complexity of stack operations: O(1)

Key insight: Stacks provide an efficient and elegant solution for problems requiring LIFO behavior, with minimal overhead in terms of both time and space complexity.

Limitations and Challenges

Limited Access

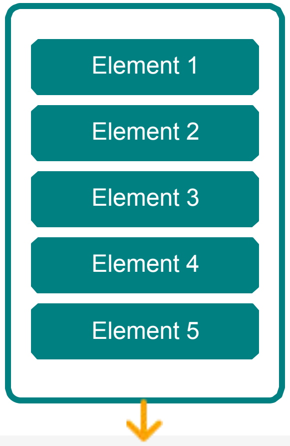
Elements can only be accessed from the top. Retrieving or modifying elements in the middle requires popping all elements above it, which is inefficient.

Fixed Capacity

Array-based implementations have a predefined maximum size. If more elements are pushed than the stack can hold, it leads to a **stack overflow** error.

Stack Overflow

Occurs when too many elements are pushed onto a fixed-size stack, resulting in data loss or program termination.



Stack Underflow

Attempting to perform a `pop`

Advanced Stack Problems & Summary

Advanced Stack Problems

Next Greater Element

Finding the next greater element for each element in an array using a stack-based approach.

Largest Rectangle

Calculating the largest rectangle in a histogram using stack to track increasing heights.

Min Stack

Designing a stack that supports getMin() operation in O(1) time using auxiliary storage.

Sort Stack

Sorting elements of a stack using another stack and limited additional memory.

These problems often require a deeper understanding of stack properties and creative application of its LIFO principle to achieve efficient solutions.

Course Summary

- **Fundamentals:** LIFO principle, basic operations, and terminology
- **Implementations:** Array-based and linked list-based approaches in multiple languages
- **Applications:** Function calls, undo mechanisms, expression evaluation, and syntax validation
- **Pros & Cons:** Understanding time complexity and memory efficiency trade-offs

Stacks are a foundational concept in computer science.

Mastering them is crucial for developing robust and efficient software systems.

Practice implementing stacks to solidify your understanding!