



Ask a Question



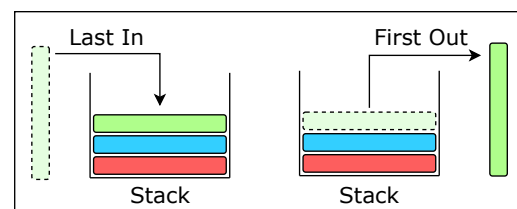
# Introduction to Stacks

Let's go over the Stacks pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following... ▼

## About the pattern

A **stack** is a linear data structure that organizes and manages data in a Last In, First Out (LIFO) manner. This means the last element added to the stack is the first to be removed. Think of it like a stack of plates where you can only add or remove plates from the top.



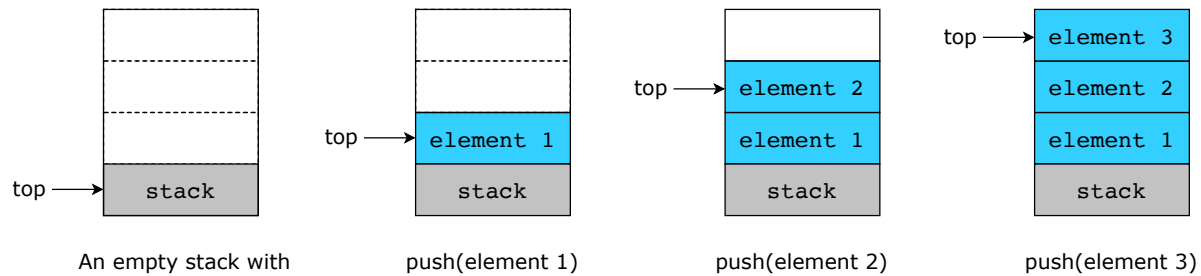
There are two fundamental operations, push and pop, to add and remove elements while maintaining the LIFO order. Let's delve into each operation:

- **push:** This operation involves placing an element at the top of the stack. Whenever we push a new element onto the stack it becomes the new top element and this is why the stack grows in an upward direction.



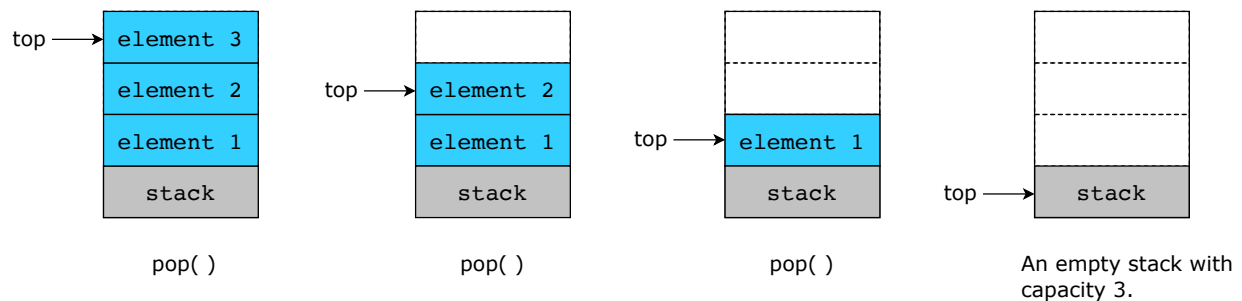
- **pop:** This operation involves removing the top element from the stack. The removed element is usually returned by the pop operation so that we can use or process it. After a pop operation, the element just below the one removed becomes the new top element.

The slides below are the visual demonstration of the push and pop operations:



**Note:** The top here represents the most recently added element. It changes dynamically with each push call.

1 of 2



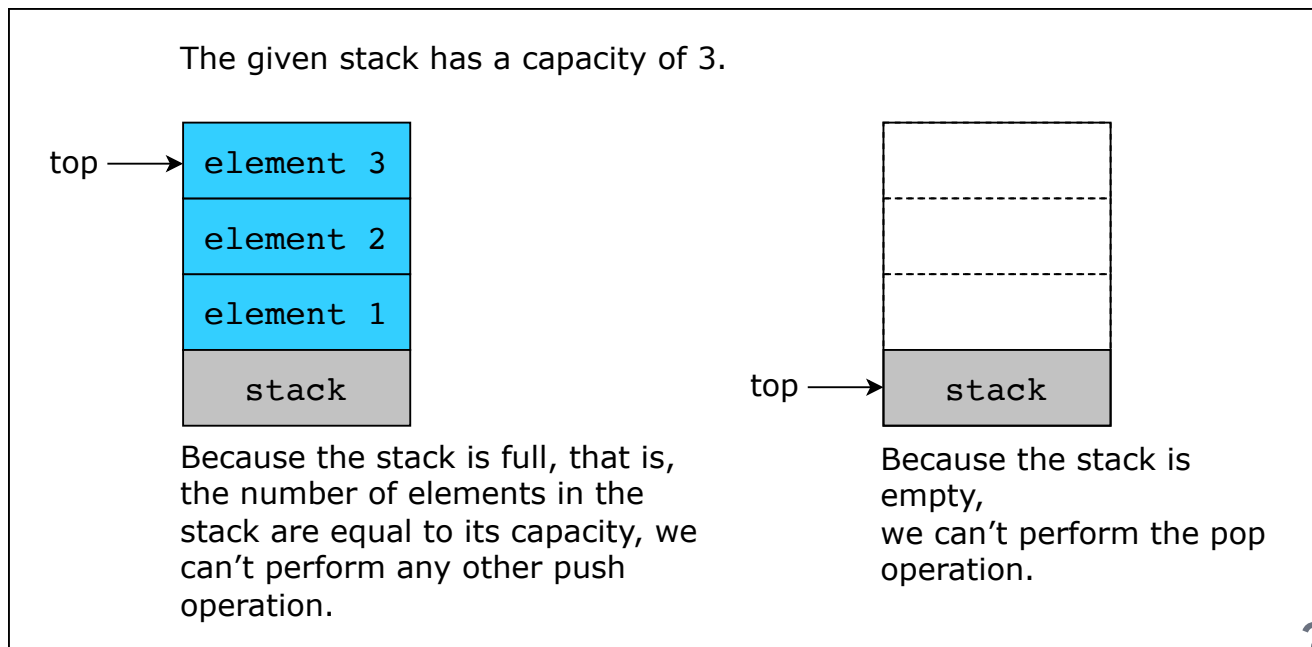
Each pop operation removes and returns the top-most element in the stack.

2 of 2

There is some capacity associated with every stack which is nothing but the size of the stack. It is important to keep an eye on the stack's capacity while performing the push and pop operations otherwise it results in either stack overflow or underflow.

- A **stack overflow** occurs when we try to push an element onto a full stack. Therefore, a push operation can't be called on a full stack. For example, if the stack has a capacity of 3, and three push operations have already been executed, attempting another push operation would exceed the stack's capacity and is not allowed.
- A **stack underflow** occurs when we try to pop an element from an empty stack. Therefore, a pop operation can't be called on an empty stack.

If either a stack overflow or underflow occurs, it can result in a memory-related issue, potentially leading to a crash or triggering an error.



There are a few other stack operations, so let's quickly go over them along with a summary of the previous two:



Operation	Time Complexity	Description
Push	$O(1)$	Adds the element at the top of the stack.
Pop	$O(1)$	Removes and returns the element from the top of the stack.
Peek	$O(1)$	Returns the element at the top of the stack without removing it.
IsEmpty	$O(1)$	Checks whether the stack is empty or not. Returns true if empty, false otherwise.
Size	$O(1)$	Returns the total number of elements in the stack.

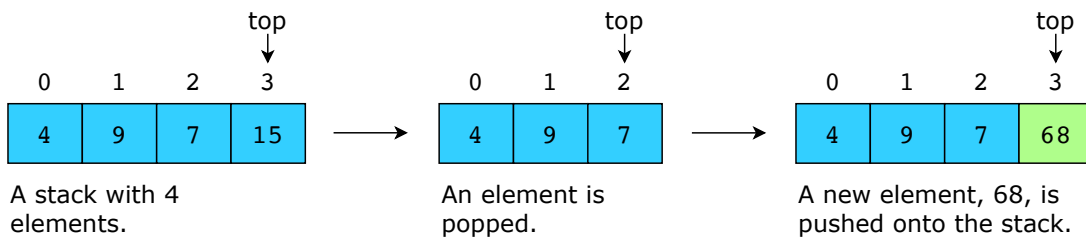
Stacks can be implemented using either arrays or linked lists. Let's explore both implementations:

- In an array-based implementation, a stack is represented as a fixed-size array where elements are added or removed from one end, typically the end of the array. A pointer or index variable is used to keep track of the top element of the stack. When an element is pushed onto the stack, it is added at the top of the stack by incrementing this pointer, and when an element is popped from the stack, it is removed from the top of the stack by decrementing this pointer.
- In a linked list-based implementation, each element of the stack is represented by a node in a linked list. Each node contains the data element and a pointer/reference to the next node. The top of the stack is represented by the head (or first) node of the linked list. When an element is pushed onto the stack, a new node is added at the beginning of the linked list, and when an element is popped from the stack, the head node is removed.

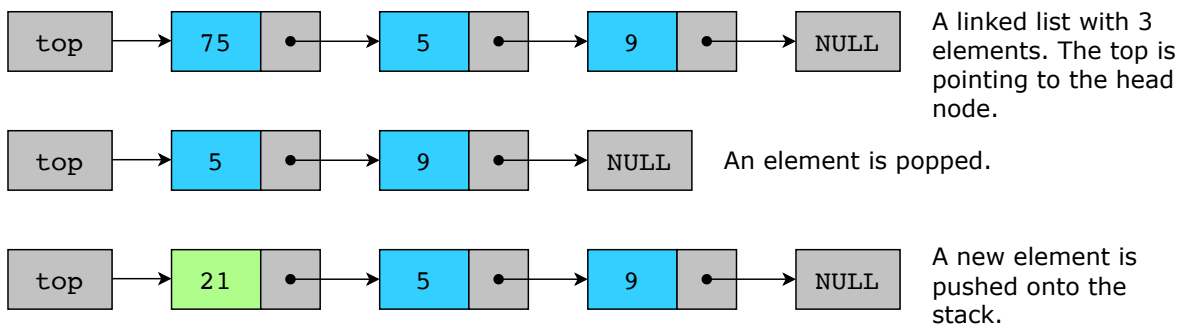
?

T



**Stack represented as an array:**

1 of 2


**Stack represented as a linked list:**

2 of 2



Stacks can be used to:



- 
- Store elements with sequential dependencies, such as in expressions or algorithms. For example, if we have the expression  $2 + 3 \times 7$ , then the stack ensures that the  $\times$  operator (having a higher precedence) must be performed before the  $+$  operator (having a lower precedence).
  - Ensure safe storage without arbitrary modification from middle positions. This property is particularly useful in scenarios where preserving the order and integrity of data is critical. For example, in banking apps to maintain a transaction history for each bank account.
  - Repeatedly modifying a stream of elements based on specified conditions. This behavior is commonly seen in various algorithms and problem-solving scenarios where elements are processed iteratively, and decisions are made based on the current state of the stack. For example, if there is a stream of incoming job requests with their priorities, and the server can handle only one job at a time, then the job with the highest priority will be processed first. Let's say the stream of jobs includes job A (high), job B (low), job C (medium), and job D (high). Here, job A will be processed first. Once it is finished, job D will be executed due to its high priority. Following that, job C will be processed, and finally, job B will be executed.

## Examples

The following examples illustrate some problems that can be solved with this approach:

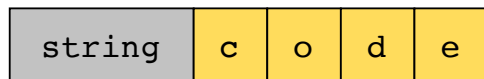
1. **Reverse a string using stack:** Given an input string as an array of characters, reverse all the characters in this string.

?

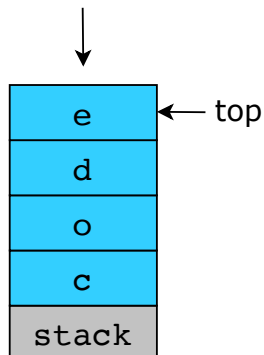
Tt



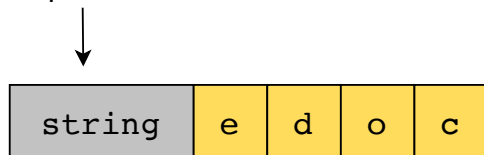
A string given as an array of characters.



Push all the characters onto the stack starting from index 0.



Pop all the characters from the stack.



The string is now reversed from "code" to "edoc".

2. **Evaluate postfix expression:** Given an array of tokens that represents an arithmetic expression in a postfix notation, evaluate the expression and return an integer that represents the value of the expression.

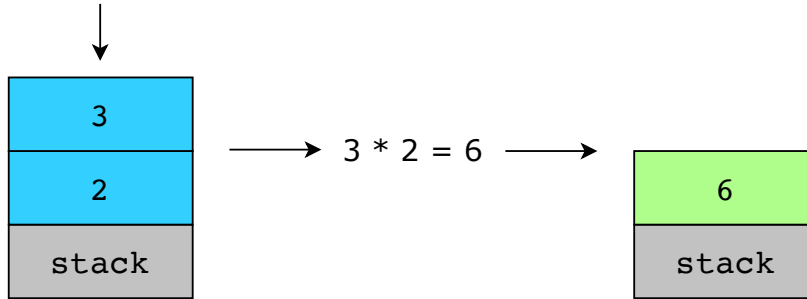
?

Tt



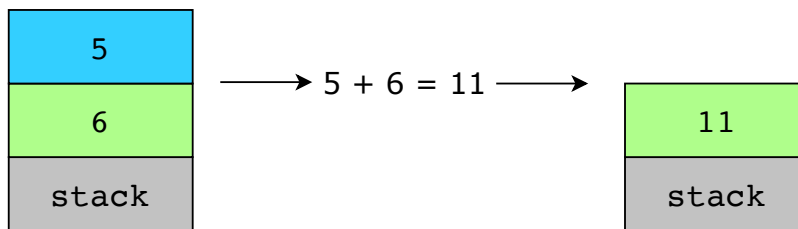


Push the digits onto the stack until an operand comes up.



When an operand appears, pop the elements in stack and perform the relevant arithmetic operation. Push the result back to stack.

Repeat the process until we traverse the entire array.



Since we have traversed the complete array, return the value present in the stack.

## Does your problem match this pattern?

Yes, if either of these conditions is fulfilled:

- **Reverse order processing:** The problem involves processing elements in reverse order or requires the last element added to be processed first.
- **Nested structures handling:** The problem involves nested structures, like parentheses, brackets, or nested function calls.
- **State tracking:** The problem requires keeping track of previous states or undoing operations.
- **Expression evaluation:** The problem involves evaluating expressions.


?

Tt





No, if either of these conditions is fulfilled:

- 
- **Order dependence:** The problem requires either a different order dependence than Last In, First Out (LIFO) or there is no order dependency at all.
  - **Random access:** The problem involved frequent access or modification of elements at arbitrary positions is needed and not just from the end.
  - **Need for searching:** The problem requires efficient searching for elements based on values or properties.

## Real-world problems

Many problems in the real world share the stack pattern. Let's look at some examples.

- **Function call stack:** Stacks are used to manage function calls in programming languages. When a function is called, its context is pushed onto the stack, and when the function completes, it is popped off the stack.
- **Text editor undo/redo feature:** Stacks are commonly used to undo/redo the changes made while editing. Each edit operation is pushed onto the stack, allowing users to revert to the previous state of an action by popping the most recent edit.
- **Browser back and forward buttons:** The back and forward navigation in web browsers is implemented using a stack to keep track of visited pages. Clicking the back button pops the current page, while the forward button pushes pages back onto the stack.
- **Call history in smartphones:** Smartphones maintain a call history stack, allowing users to navigate through the list of recent calls in a Last In, First Out fashion.

# Strategy time!

Match the problems that can be solved using the stacks pattern.



**Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

## Match The Answer

① Select an option from the left-hand side

Check if the parentheses in a mathematical expression are balanced or not.

Stacks

Find the minimum number of perfect squares

Some other pattern

