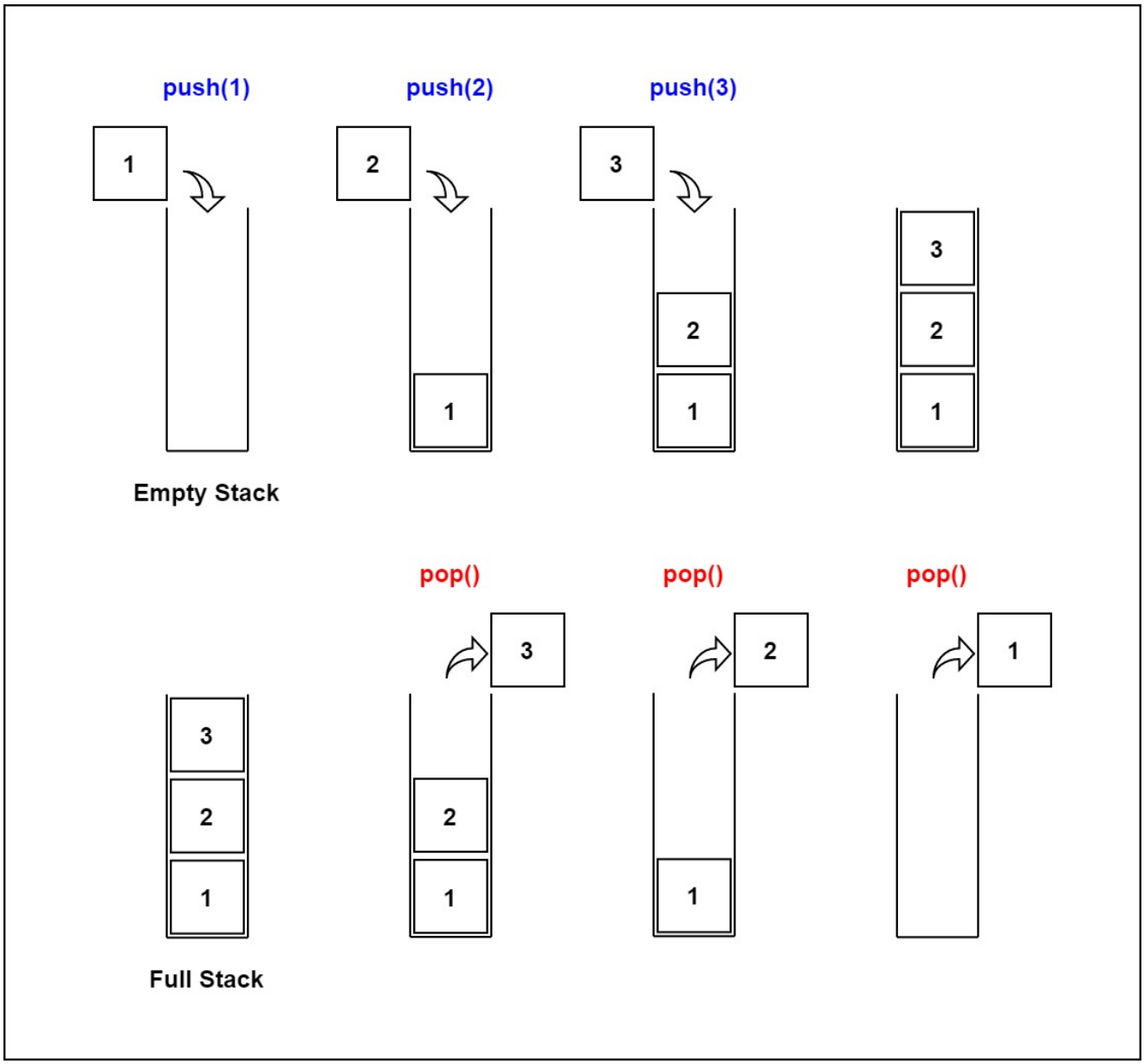
Stack Data Structure

Alright, let's dive into the fascinating world of the Stack data structure! Today, we'll explore a linear data structure that follows the **Last In First Out (LIFO)** principle. In simpler terms, the last element we put into the stack will be the first one to be removed.

To help you understand the concept better, think of a stack as a pile of plates stacked on top of each other. You can add a new plate to the top of the stack, remove the top plate, but if you want the plate at the bottom, you must first remove all the plates on top. This perfectly demonstrates how a stack operates.

LIFO Principle of Stack

In programming, when we put an item on top of the stack, we call it "**push**," and when we remove an item, we call it "**pop**." Take a look at the image we've provided; even though item 3 was placed last, it was removed first. This is the essence of the Last In First Out (LIFO) Principle.

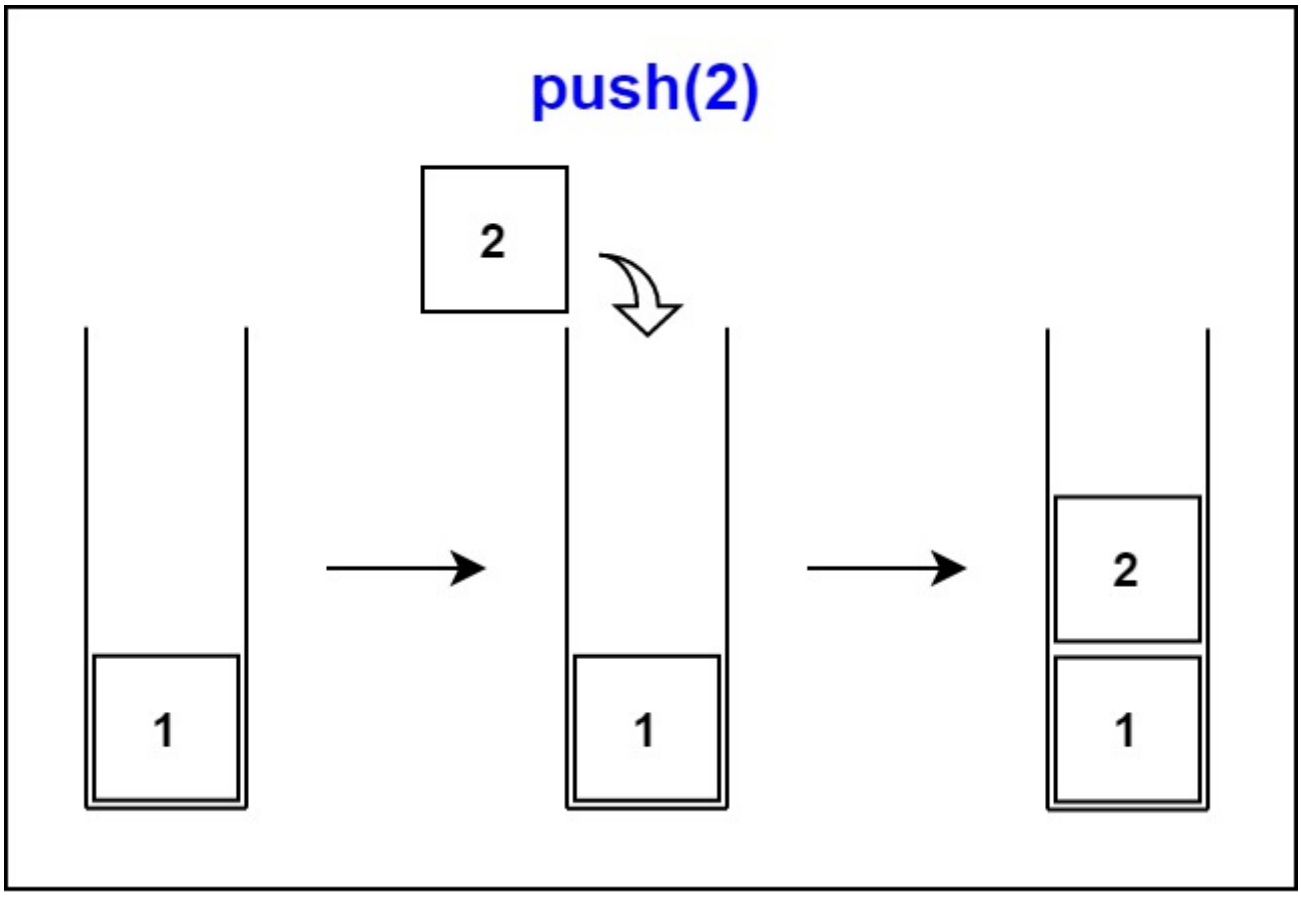


You'll be happy to know that we can implement a stack in various programming languages such as C, C++, Java, Python, or C#. The specification for the stack remains pretty much the same across all languages.

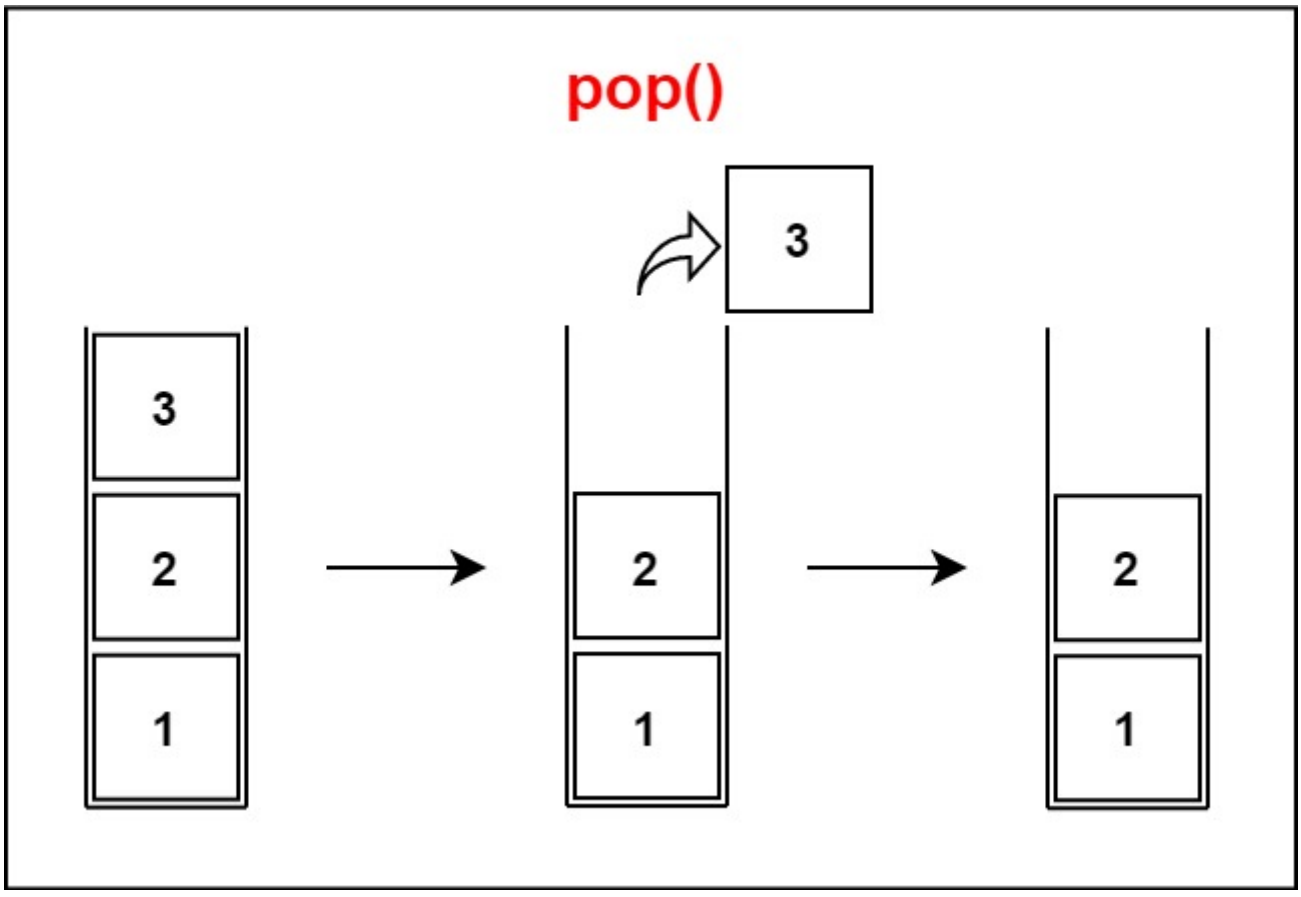
Basic Operations of Stack

Now, let's explore the basic operations that allow us to perform different actions on a stack:

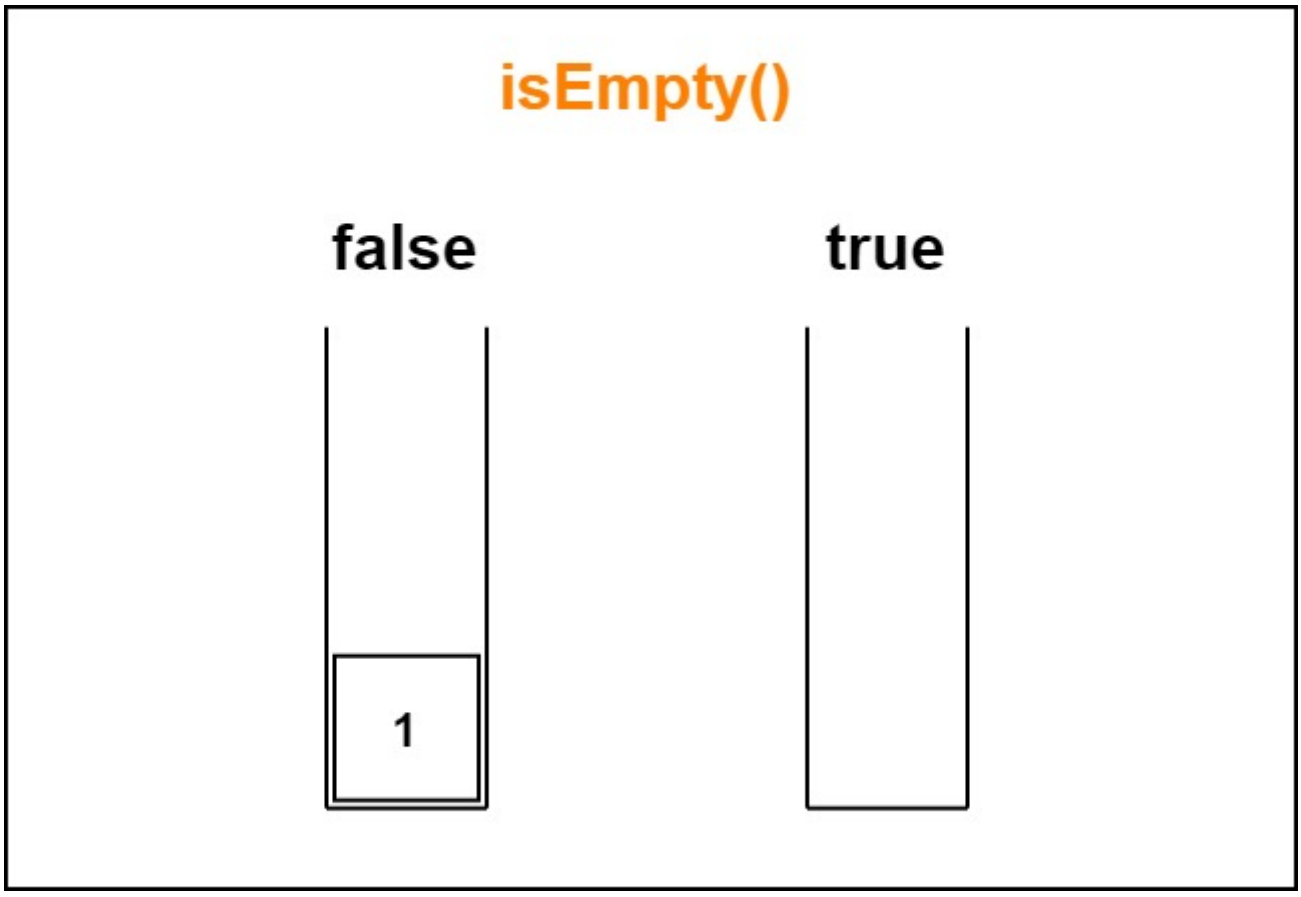
1. **Push**: This operation adds an element to the top of the stack.



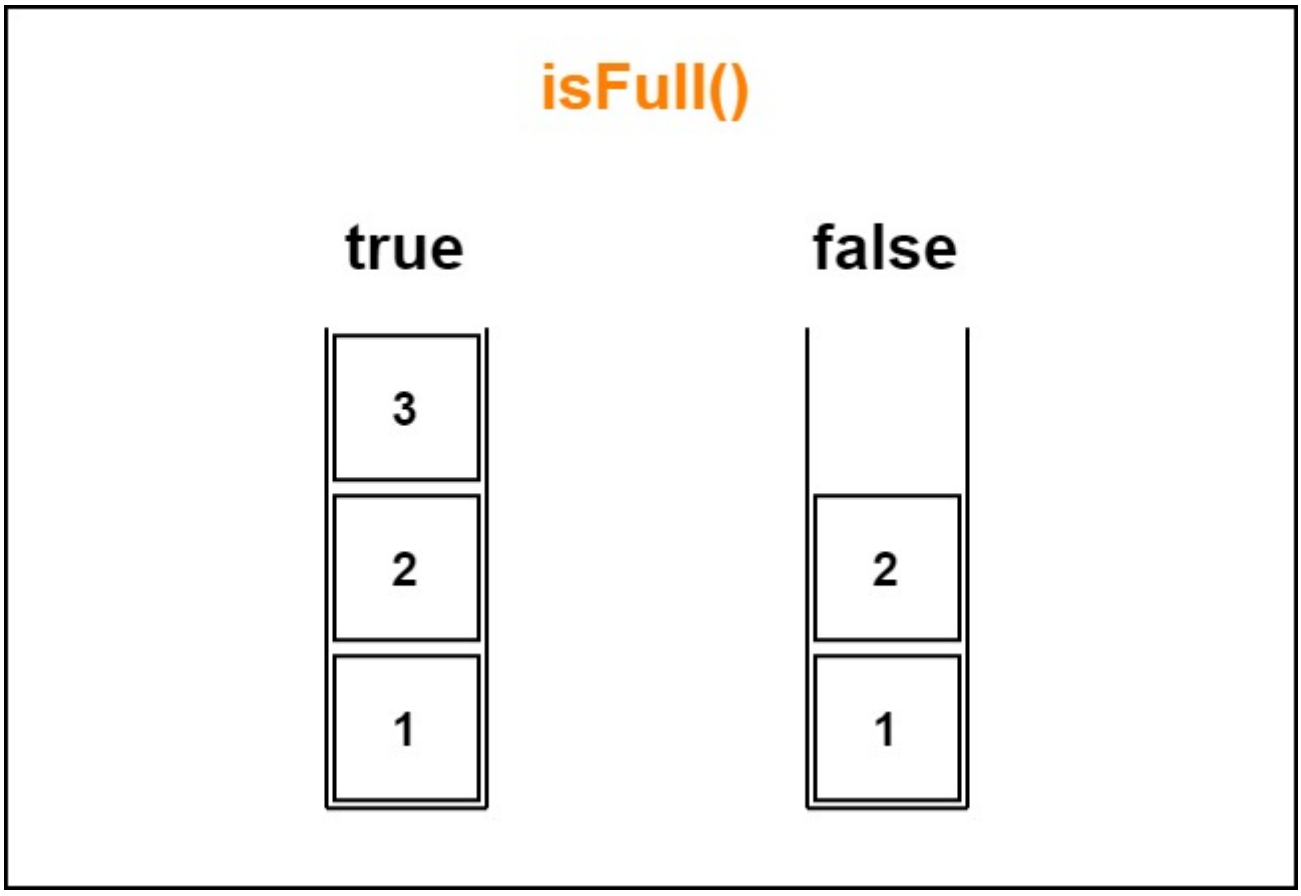
2. **Pop**: This operation removes an element from the top of the stack.



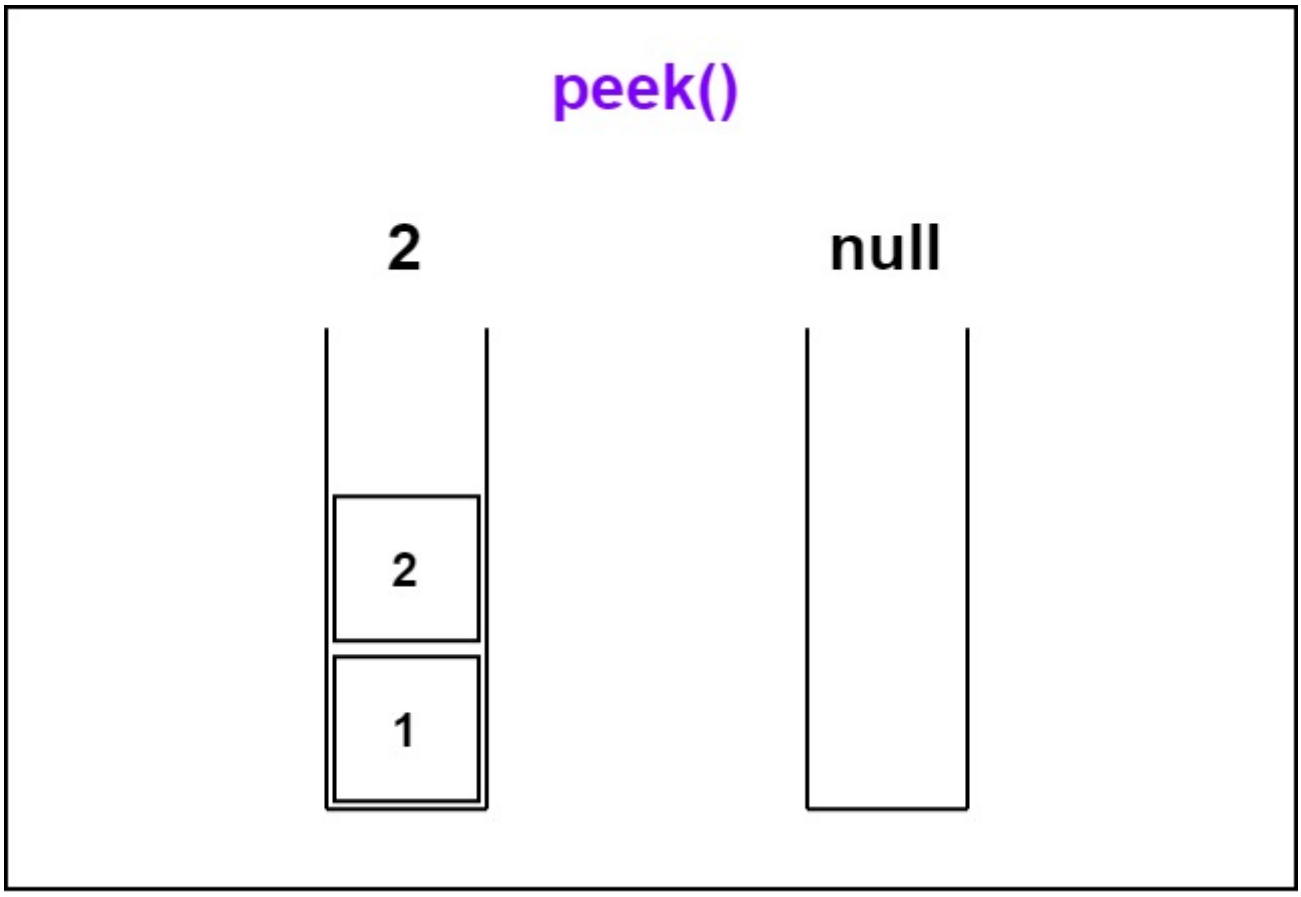
3. **IsEmpty**: We can use this operation to check if the stack is empty.



4. **IsFull**: This operation helps us check if the stack has reached its maximum capacity.



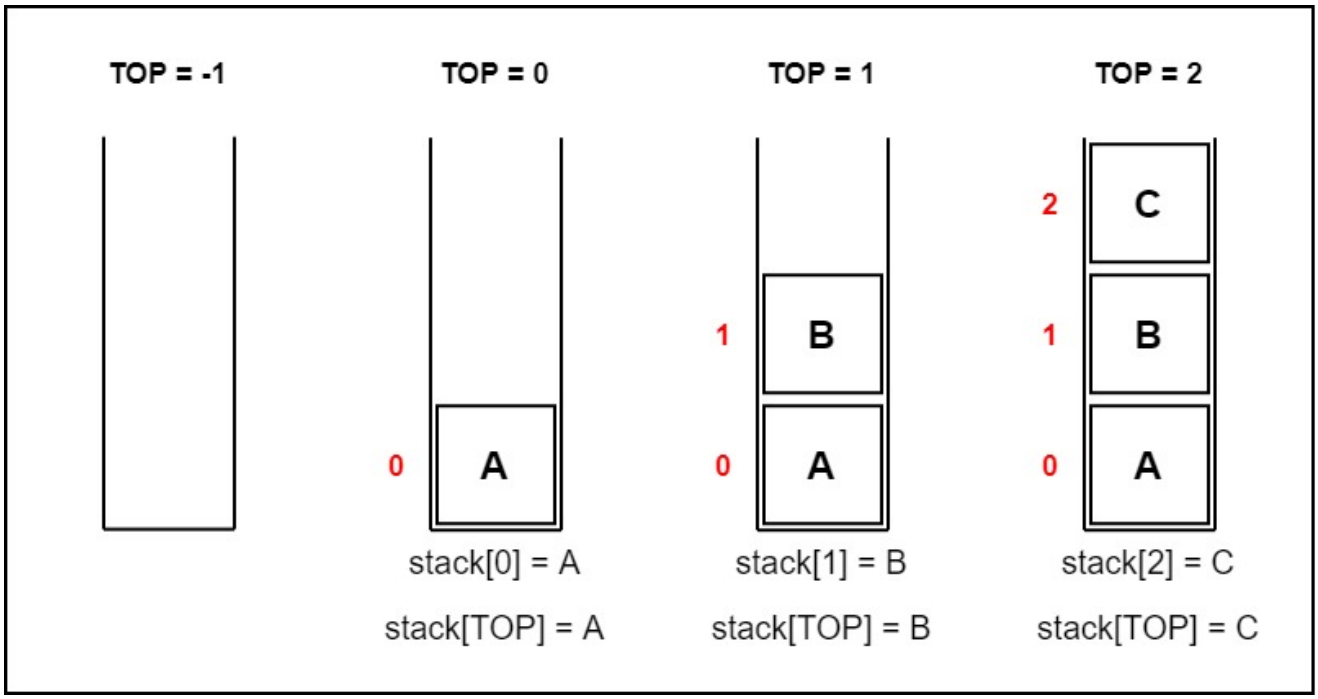
5. **Peek**: The peek operation allows us to get the value of the top element without removing it.

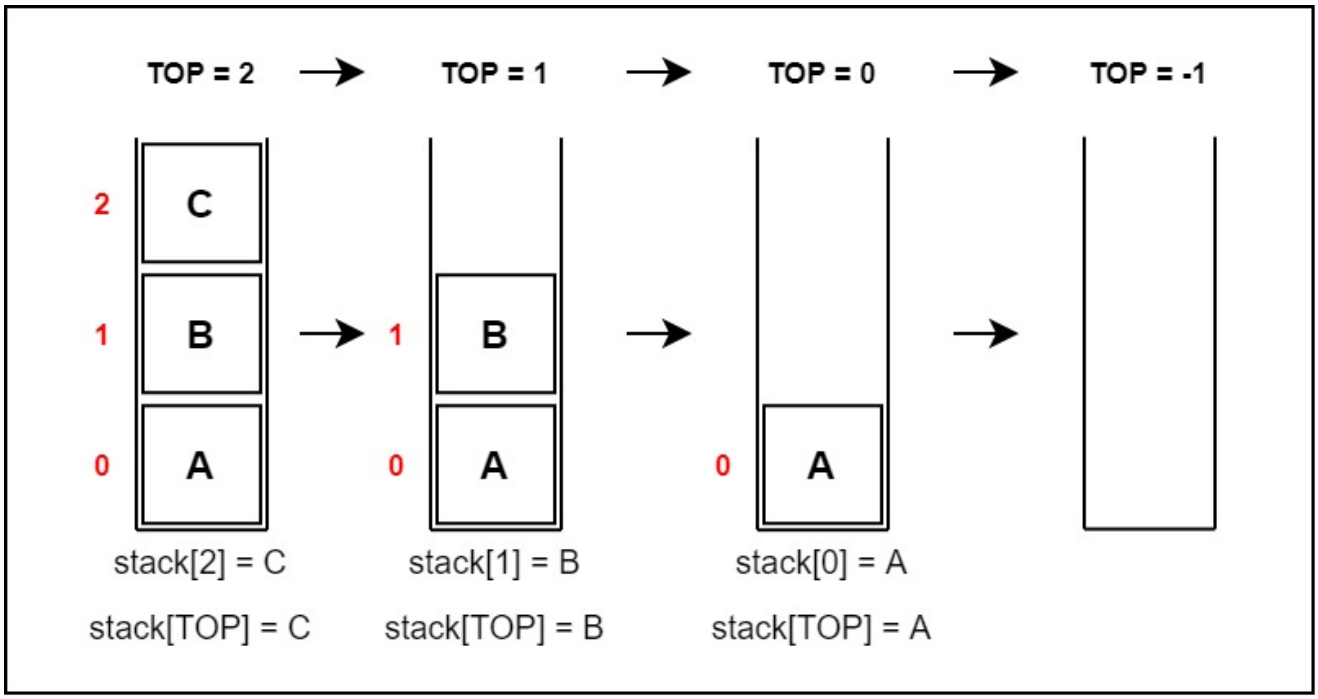


Working of Stack Data Structure

Here's how the stack operations work in detail:

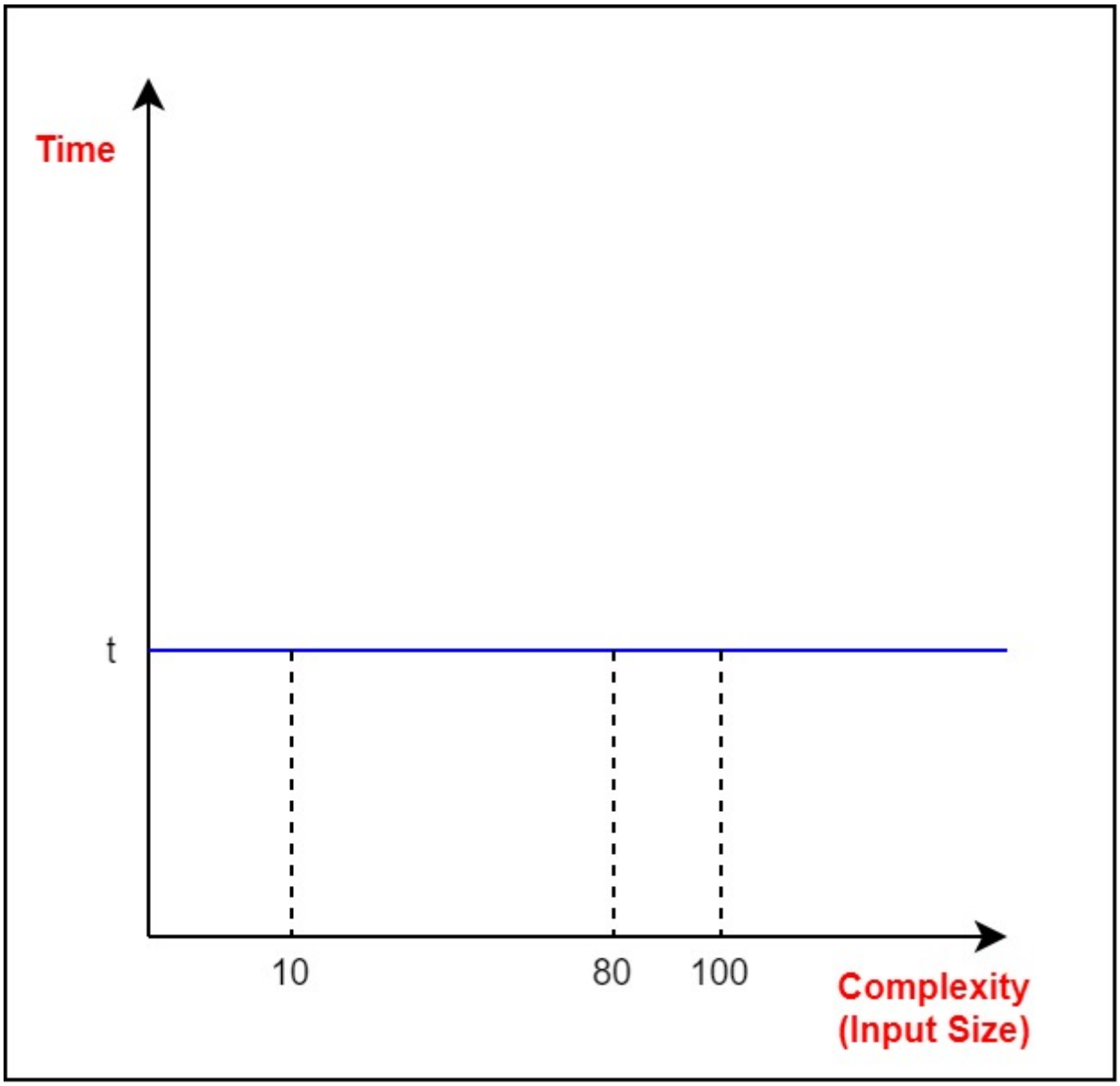
1. A pointer called **TOP** is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1, which allows us to check if the stack is empty by comparing TOP == -1.
3. When pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. When popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full.
6. Before popping, we check if the stack is already empty.





Stack Time Complexity

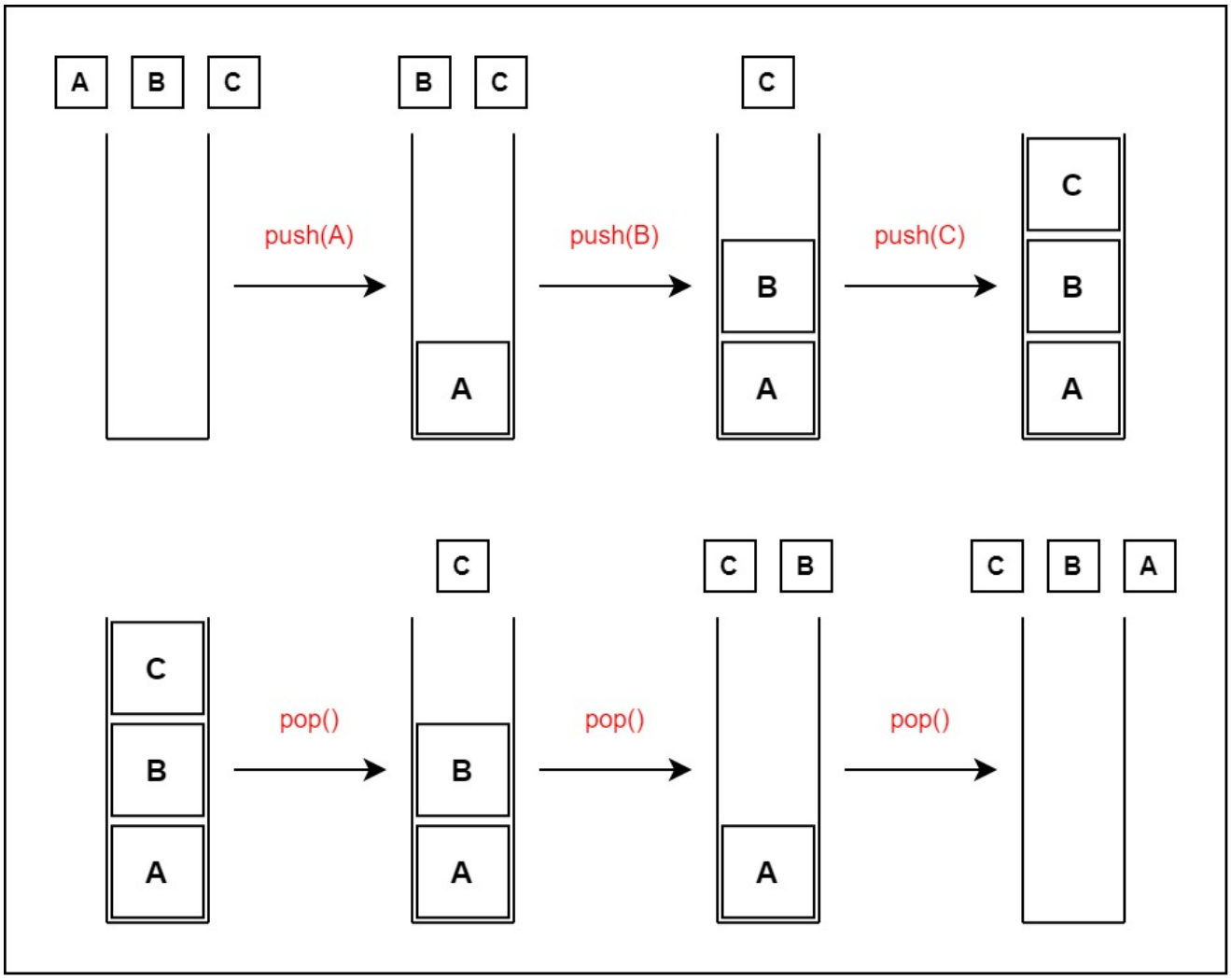
For the array-based implementation of a stack, the push and pop operations take constant time, which is **O(1)**. This means that the time taken to perform these operations doesn't depend on the number of elements in the stack.



Applications of Stack Data Structure

The stack might seem simple, but it is incredibly powerful and finds its applications in various areas:

1. **To reverse a word**: By pushing all the letters of a word into a stack and then popping them out, we can reverse the word efficiently.



2. **In compilers**: Compilers use stacks to evaluate expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.
3. **In browsers**: The back button in a browser uses a stack to store the URLs of previously visited pages. Each time you visit a new page, its URL is added to the top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.