

Data Structures Overview

Data structures are the fundamental building blocks of computer programming, providing efficient ways to organize and manipulate information. From simple arrays to complex hash maps, each data structure has unique characteristics and use cases that make it well-suited for different problem domains.

Arrays

Definition

An array is a collection of elements, all of the same data type, stored in contiguous memory locations.

Key Properties

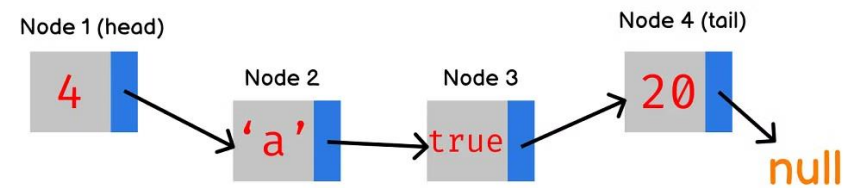
Arrays provide constant-time access to elements by index, but fixed size and insertion/deletion complexity.

Use Cases

Arrays are commonly used for storing and processing large datasets, implementing other data structures, and solving problems that require frequent element access.

Linked Lists

LINKED LIST DATA STRUCTURE



A linked list is formed from many nodes linked through the **next** pointer

The first node is also known as the head node

The last element (tail) will have the **next** property pointing at **null**

1

Nodes

Linked lists are made up of individual nodes, each containing data and a pointer to the next node.

2

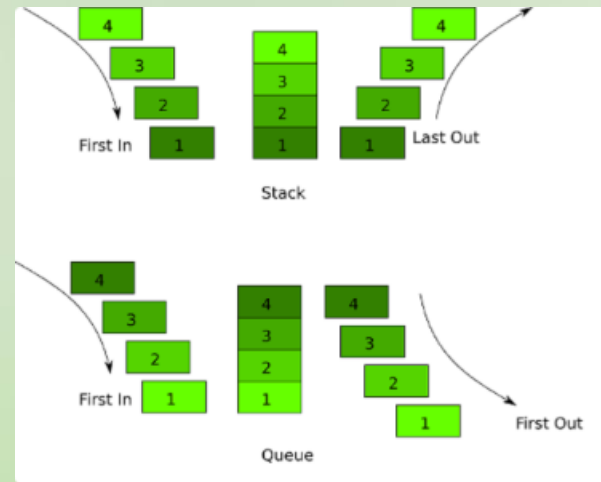
Insertion/Deletion

Linked lists excel at efficient insertion and deletion of elements, with constant-time complexity.

3

Dynamic Size

Unlike arrays, linked lists can grow or shrink in size as needed, making them more flexible.



Stacks and Queues



Stacks

Last-in, first-out (LIFO) data structure. Useful for managing function calls, undo/redo operations, and more.



Queues

First-in, first-out (FIFO) data structure. Ideal for processing tasks in the order they were received.

Hash Maps

Key-Value Pairs

Hash maps store data as unique key-value pairs, allowing for fast lookups and insertions.

Collision Handling

Hash collisions, where multiple keys map to the same index, are resolved using techniques like chaining or probing.

Use Cases

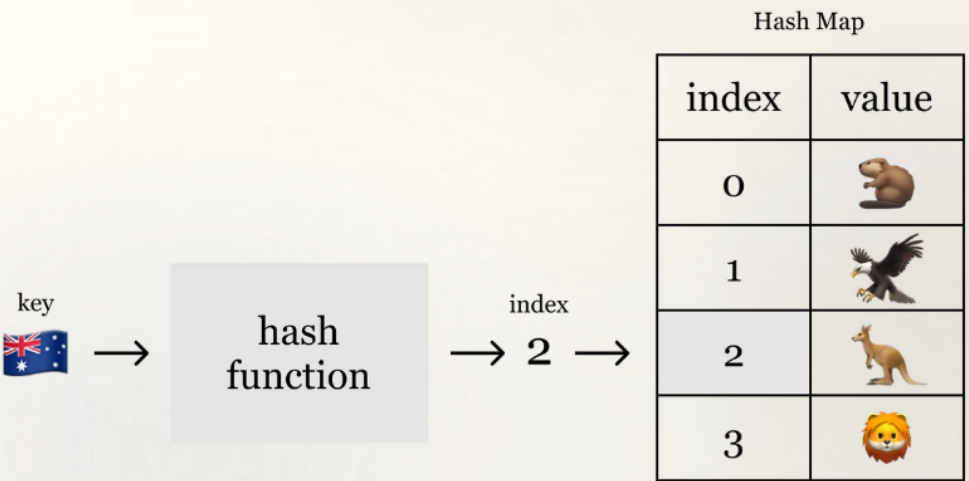
Hash maps are commonly used for caching, frequency counting, and implementing other data structures like sets.

Performance





With a good hash function, hash maps can provide constant-time average case performance for key operations.

Data

key	value
	
	
	
	

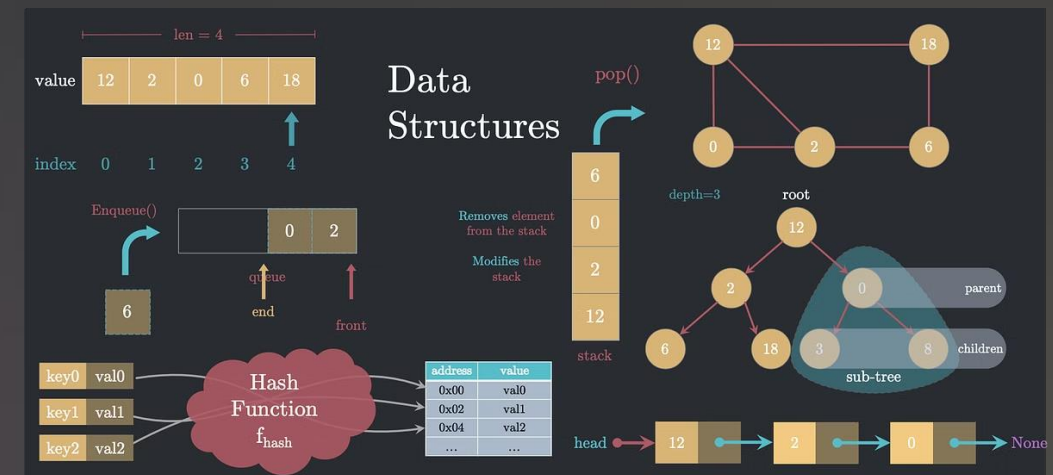


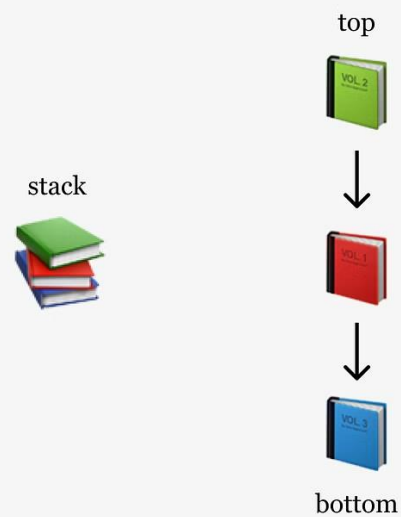
or

animalsHashMap = [⁰ , ¹ , ² , ³]

Data Structures Operations

Data structures are fundamental building blocks in programming. They organize and store data efficiently, enabling various operations for efficient data management.





Stack Operations

Stacks are a fundamental data structure that follow the Last-In-First-Out (LIFO) principle. They offer a range of basic operations that allow you to manage elements pushed onto the stack.

What is a Stack?

Linear Data Structure

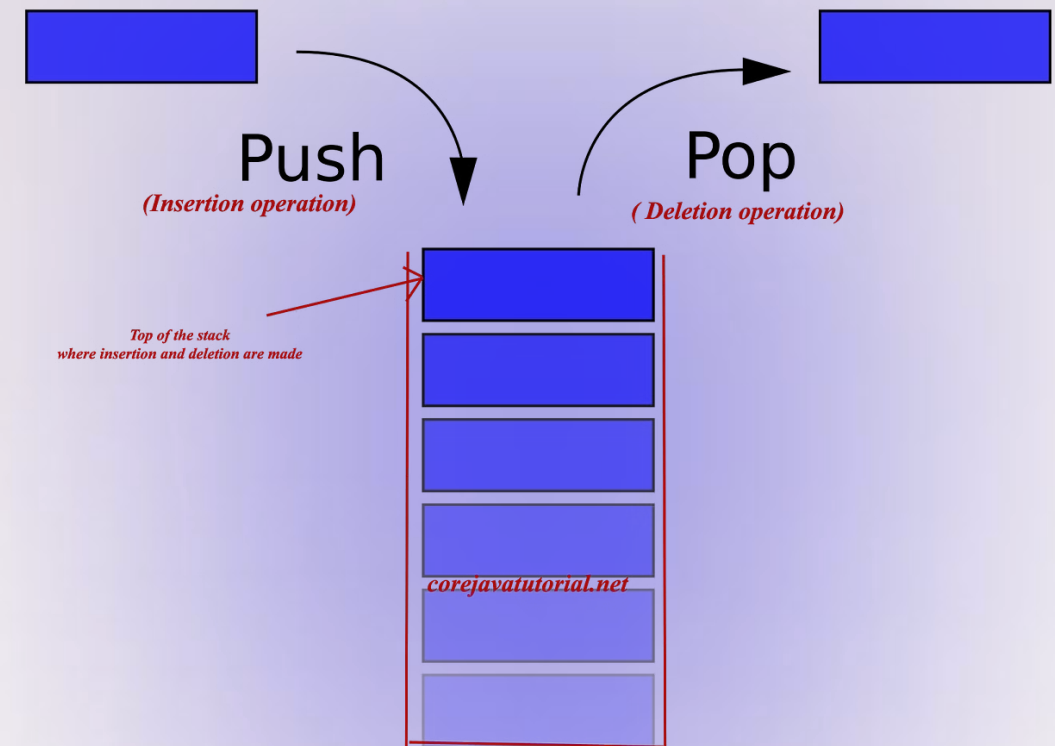
A stack is a linear data structure that stores elements in a sequential manner.

LIFO Principle

The Last-In-First-Out (LIFO) principle determines how elements are added and removed from the stack.

Simple Operations

The basic operations on a stack are push (add), pop (remove), and peek (view top element).



Common Stack Operations

1 Push

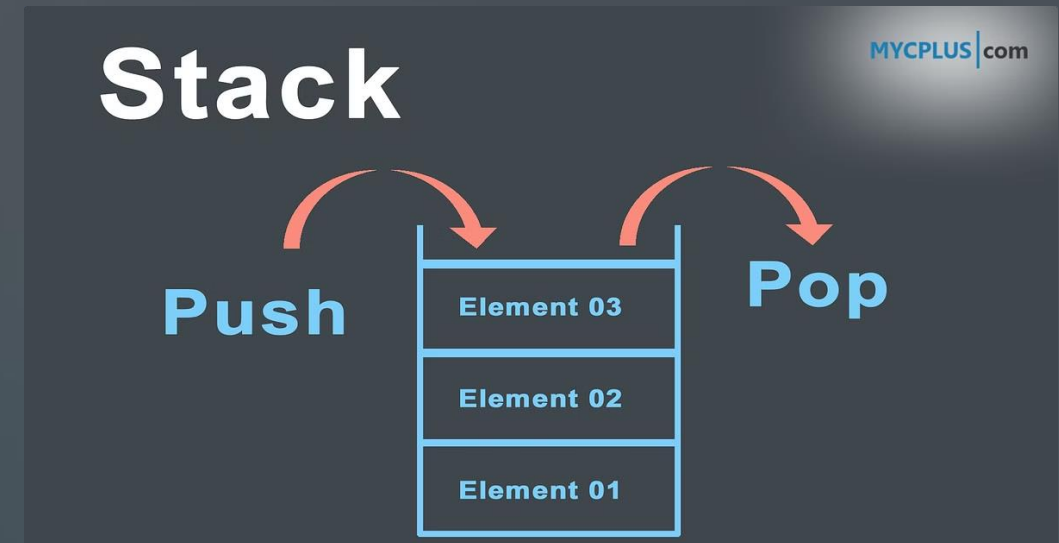
Add an item to the top of the stack.

2 Pop

Remove the top item from the stack.

3 Peek

Retrieve the top item without removing it from the stack.



Push: Add an element at the top

1 Inserting an Element

The push operation adds a new item to the top of the stack, increasing the stack's size by one.

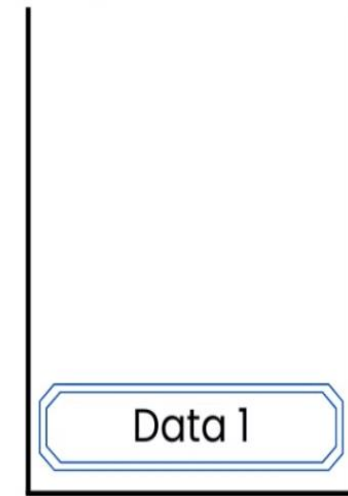
2 Updating the Top

After a push, the newly added element becomes the new top of the stack.

3 Efficient Access

Pushing to the top of the stack is an $O(1)$ operation, making it a fast and efficient way to add elements.

PUSH



← Top

Example student management :

```
public void addStudent(String id, String name, double marks) { 1usage
    students.add(new Student(id, name, marks));
    undoStack.push(item: "ADD " + id); // Save operations to stack
    redoStack.clear(); // Clear redo when new operation occurs
}
```

Pop: Remove the top element

Removing the Top

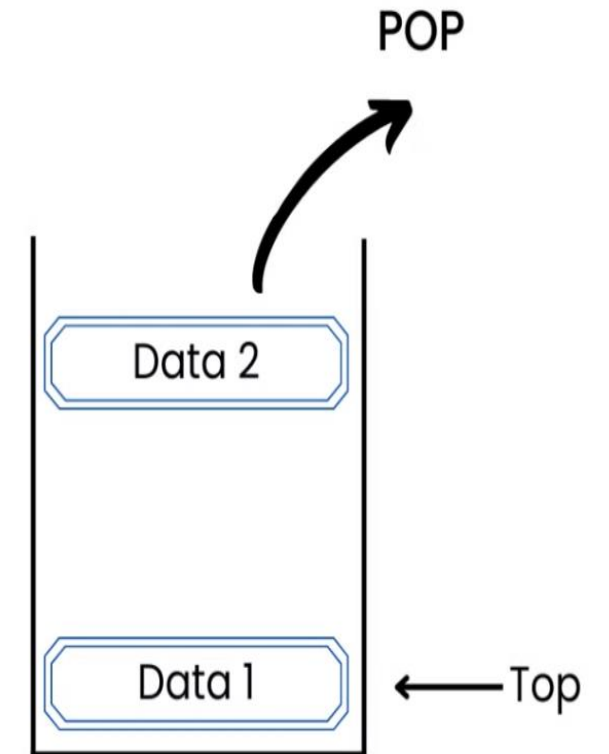
The pop operation removes the top element from the stack, decreasing the stack's size by one.

Updating the Top

After a pop, the element that was previously below the top becomes the new top of the stack.

Efficient Removal

Popping from the top of the stack is an $O(1)$ operation, making it a fast and efficient way to remove elements.



Example student management :

```
public void deleteStudent(String id) { 1 usage
    for (Student s : students) {
        if (s.getId().equals(id)) {
            undoStack.push( item: "DELETE " + id + " " + s.getName() + " " + s.getMarks());
            students.remove(s);
            redoStack.clear();
            return;
        }
    }
}
```

Top/Ppeek: View the top element without removing it



Viewing the Top

The top or peek operation allows you to access the top element of the stack without removing it.



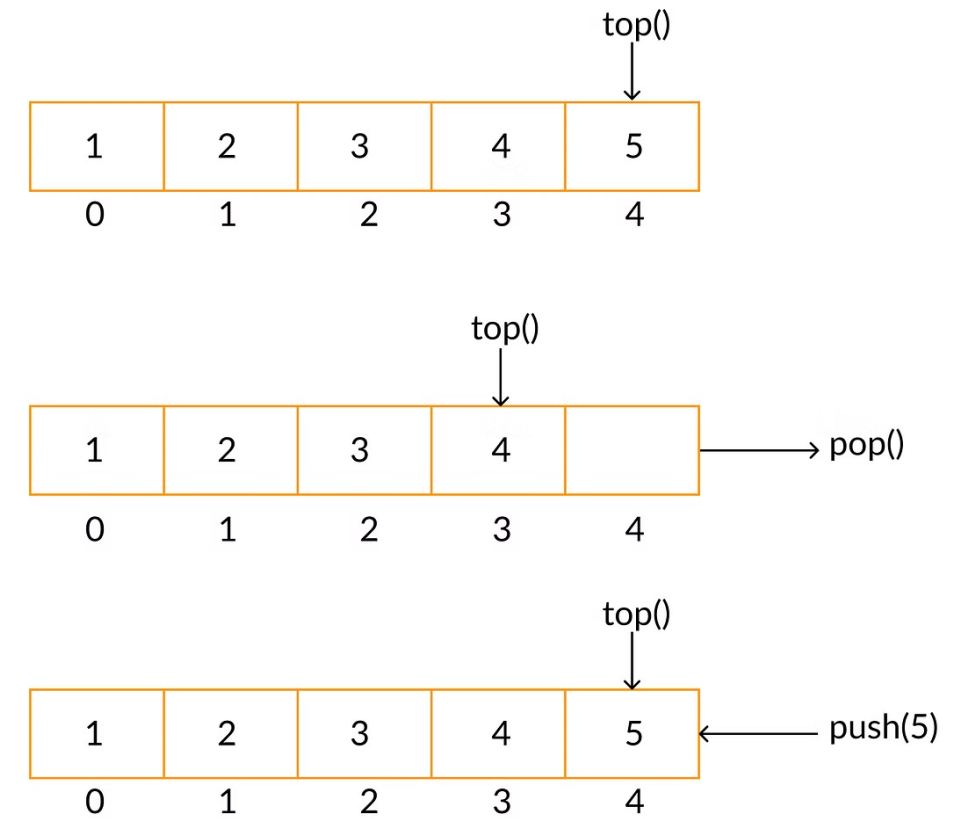
Non-Destructive

Peeking does not modify the stack, preserving the original state of the data structure.



Efficient Access

Peeking at the top element is an $O(1)$ operation, making it a quick way to inspect the stack.



isEmpty: Check if the stack is empty

1

Checking Status

The isEmpty operation determines if the stack is currently empty, containing no elements.

2

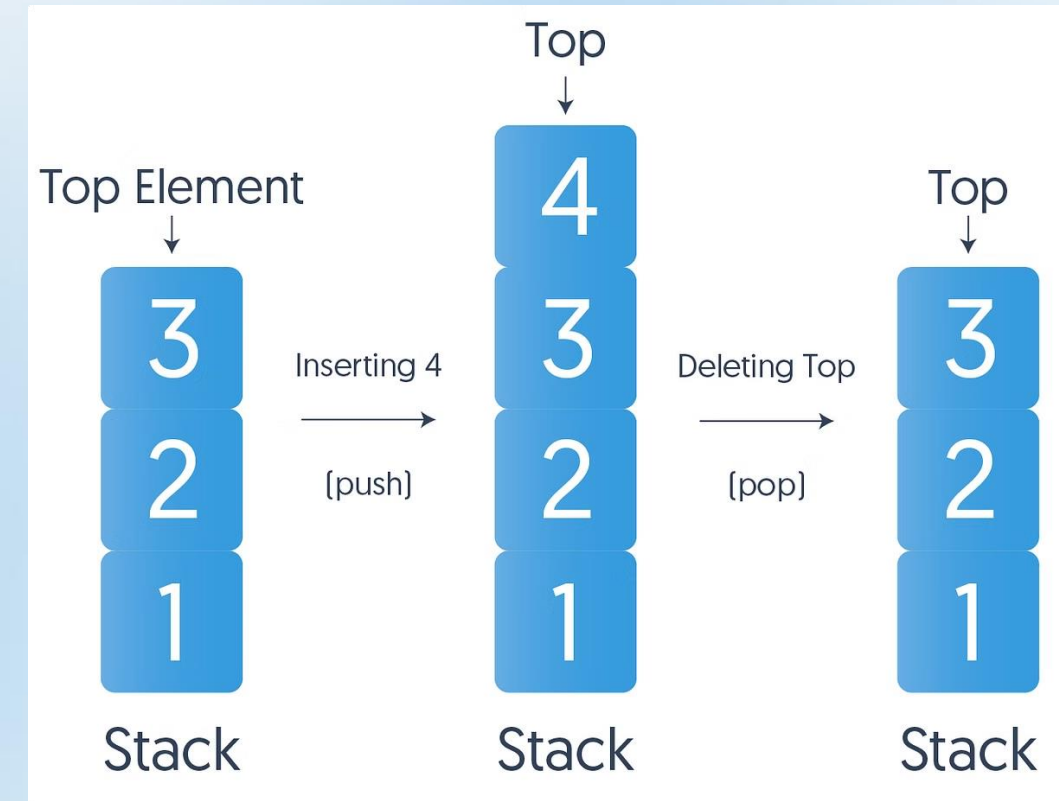
Boolean Result

The operation returns a boolean value, true if the stack is empty, false otherwise.

3

Efficient Check

Checking if a stack is empty is an $O(1)$ operation, making it a quick way to inspect the stack's state.



isFull: Check if the stack is full

Stack Capacity

Stacks have a finite size or capacity, which limits the number of elements they can hold.

Checking Fullness

The isFull operation determines if the stack has reached its maximum capacity and can no longer accept new elements.

Boolean Result

The operation returns a boolean value, true if the stack is full, false otherwise.

	Time Complexity	Space Complexity
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$
Top	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
isFull	$O(1)$	$O(1)$

Application Examples

1

Undo/Redo

Stacks are commonly used to implement undo and redo functionality in software applications.

2

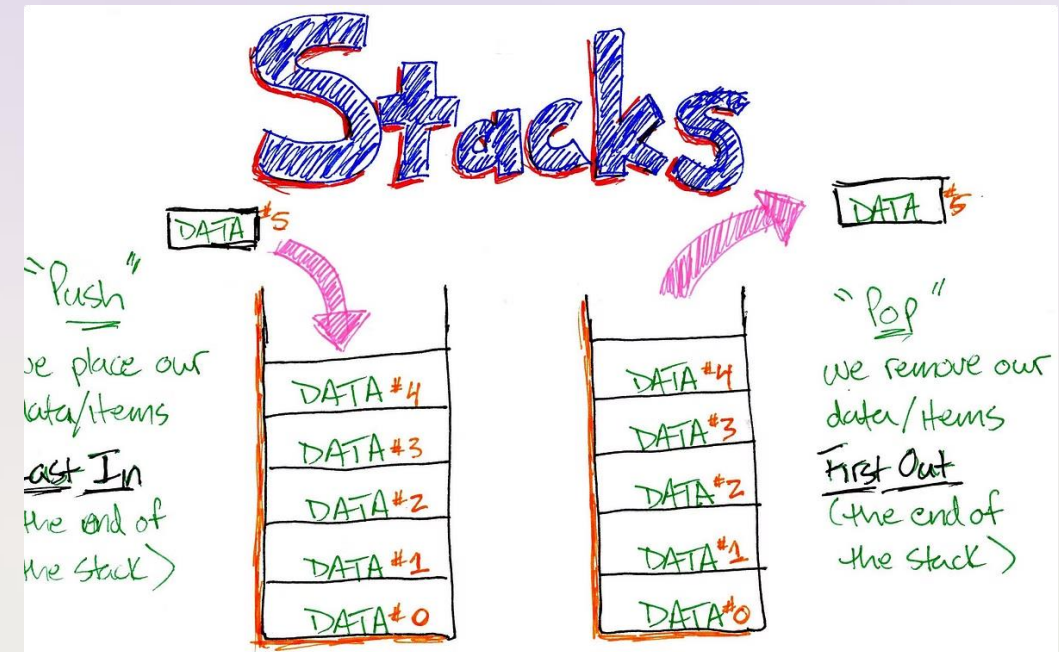
Backtracking

Stacks are useful for solving problems that require backtracking, such as maze solving and expression evaluation.

3

Function Calls

Stacks are used by the CPU to manage function calls and return addresses during program execution.



Queue Data Structure

A queue is an ordered list where items are inserted at the rear and deleted from the front, following the First-In-First-Out (FIFO) principle. This data structure is commonly used to manage processes, tasks, or resources in a sequential manner.



Definition

1 Ordered List

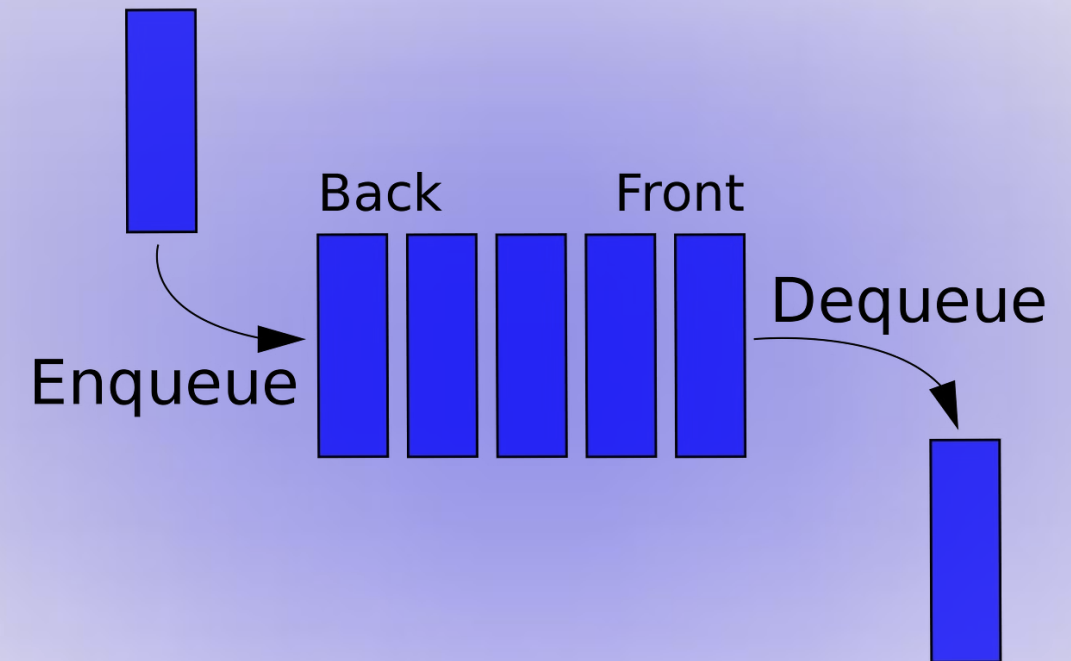
A queue is an ordered collection of elements, where the order of insertion and deletion is strictly maintained.

2 Insertion at Rear

New elements are added to the rear (or back) of the queue, similar to people joining a line.

3 Deletion at Front

Elements are removed from the front (or head) of the queue, similar to people leaving a line.



FIFO: First In, First Out

Sequential Processing

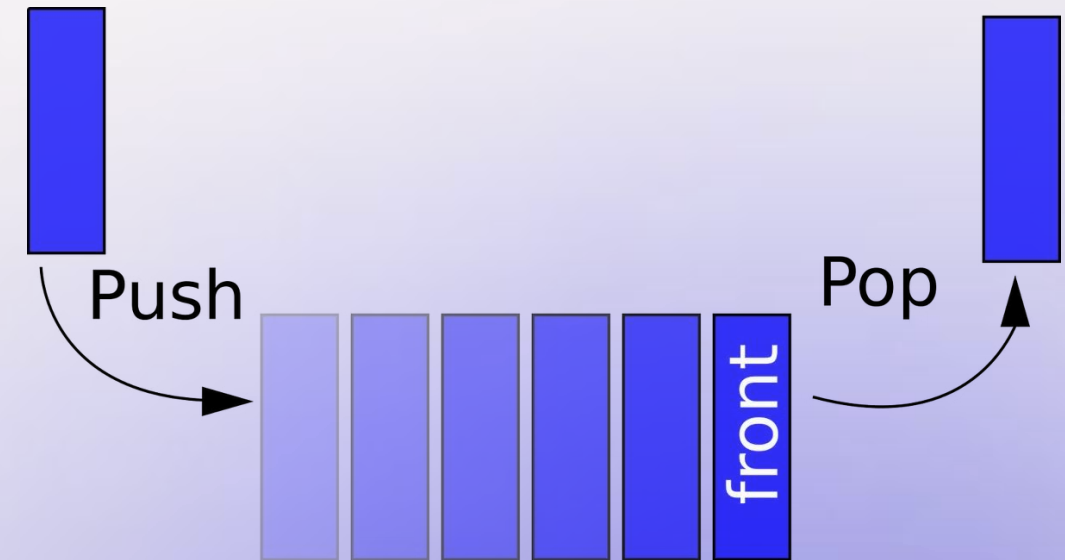
The queue ensures that elements are processed in the order they were added, similar to a line of people.

Fairness

The FIFO principle ensures fairness by serving the oldest (first) request before the newer ones.

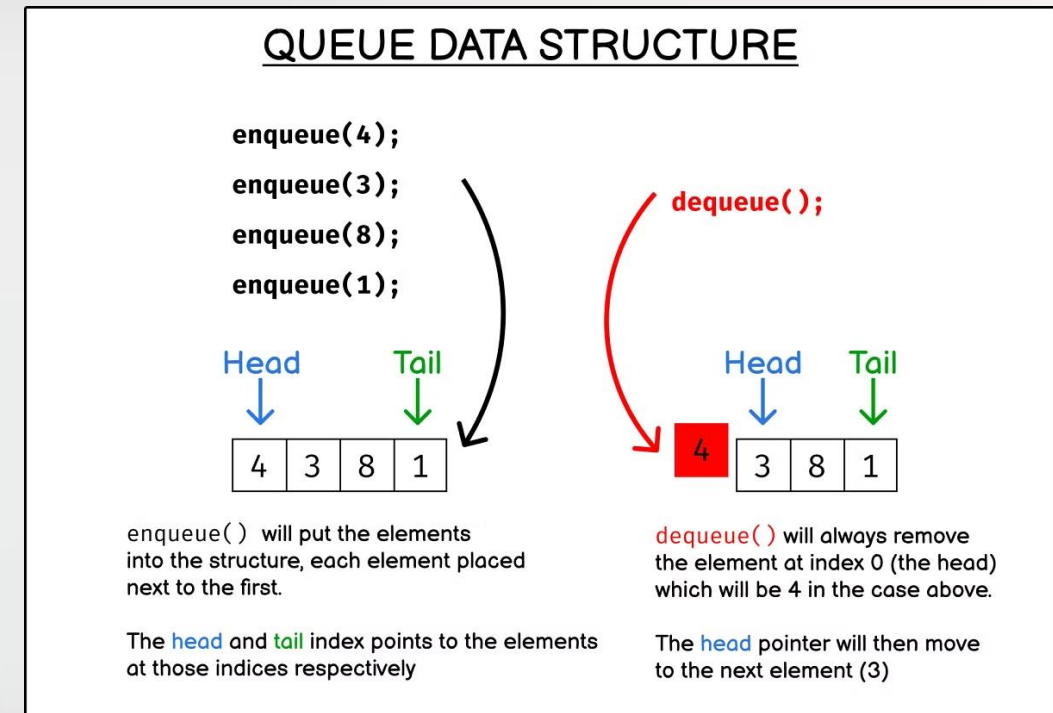
Predictable Behavior

The FIFO order guarantees a predictable and deterministic behavior for the queue data structure.



Queue Operations

Queues are an important data structure that follow the First-In-First-Out (FIFO) principle. They provide efficient ways to manage the flow of elements through a variety of applications.



Queue Operations

Enqueue

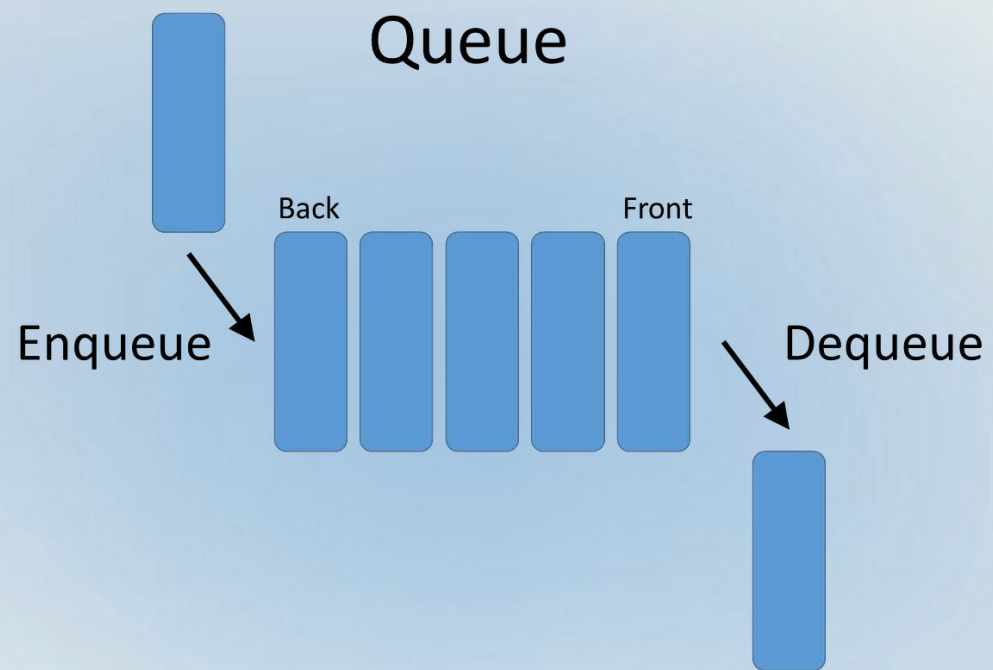
Inserts an element at the rear (back) of the queue.

Dequeue

Removes and returns the element at the front (head) of the queue.

Peek

Returns the element at the front (head) of the queue without removing it.



Queue Operations

1

Enqueue

Adding an element to the rear of the queue.

2

Dequeue

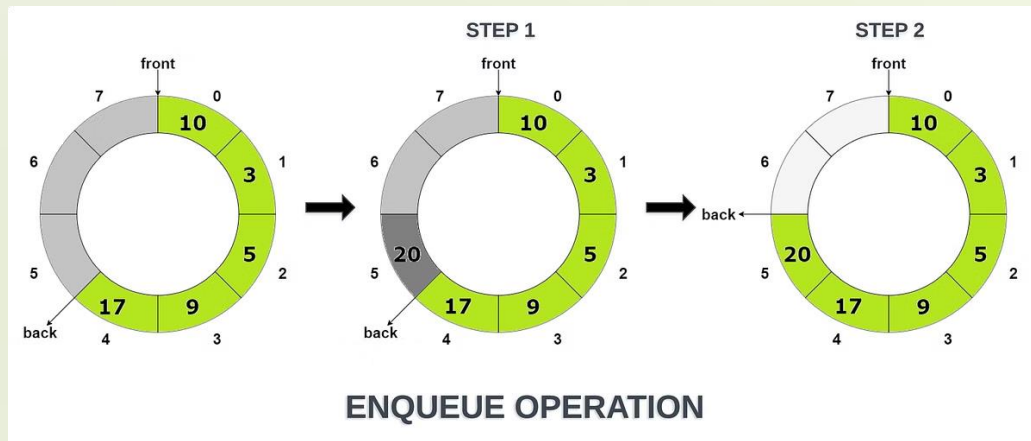
Removing the element from the front of the queue.

3

Peek

Accessing the element at the front of the queue without removing it.

Enqueue: Insert Element at Rear



Adding to the Queue

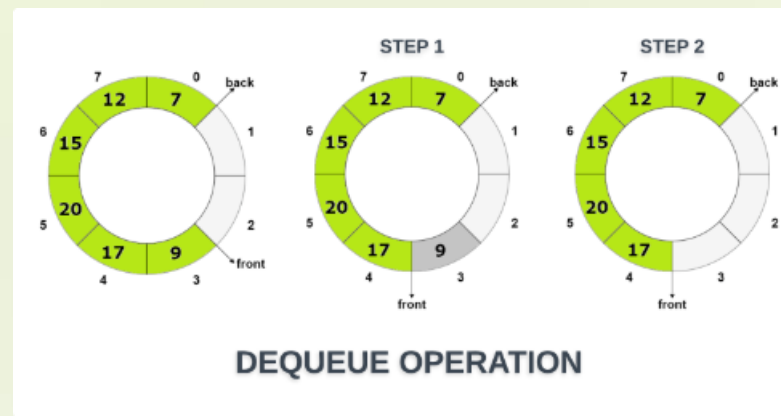
The enqueue operation inserts a new element at the rear (back) of the queue.

Maintaining Order

This ensures the FIFO order is preserved, with the first element added being the first one removed.

Efficient Insertion

Enqueue operations have constant time complexity, $O(1)$, making them a fast way to add items to the queue.



Dequeue: Remove Element from Front

Removing the Front

The dequeue operation removes and returns the element at the front of the queue.

Maintaining Order

This ensures the FIFO order is preserved, with elements leaving the queue in the same order they entered.

Shift Elements

After the front element is removed, the remaining elements in the queue are shifted forward.

Peek: View Front Element

1 Inspect Front

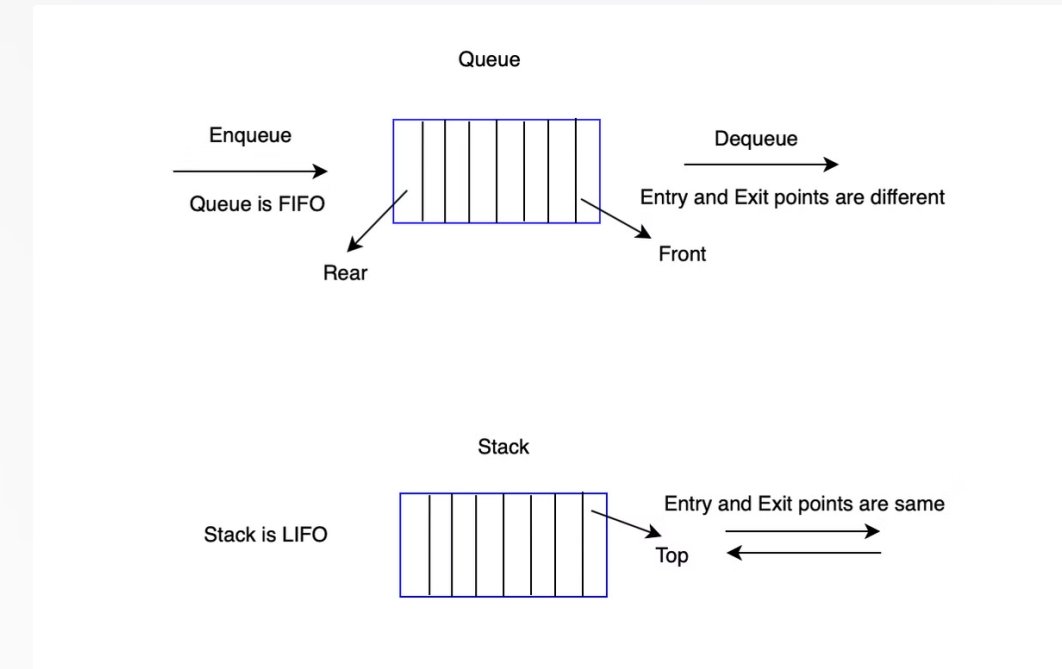
The peek operation allows you to view the element at the front of the queue without removing it.

2 Retain Data

This is useful when you need to check the next element in the queue without modifying the queue itself.

3 Constant Time

Peeking at the front element can be done in constant time, $O(1)$, making it an efficient queue operation.



Rear: View Rear Element

Access Rear

The rear operation allows you to view the element at the rear (back) of the queue.

Useful for Tracking

Knowing the rear element can be helpful for managing the queue, such as when enqueueing new items.

Constant Time

Like the peek operation, accessing the rear element can be done in constant time, $O(1)$.



isEmpty/isFull: Check Queue Status



`isEmpty()`

Checks if the queue is empty and has no elements.



`isFull()`

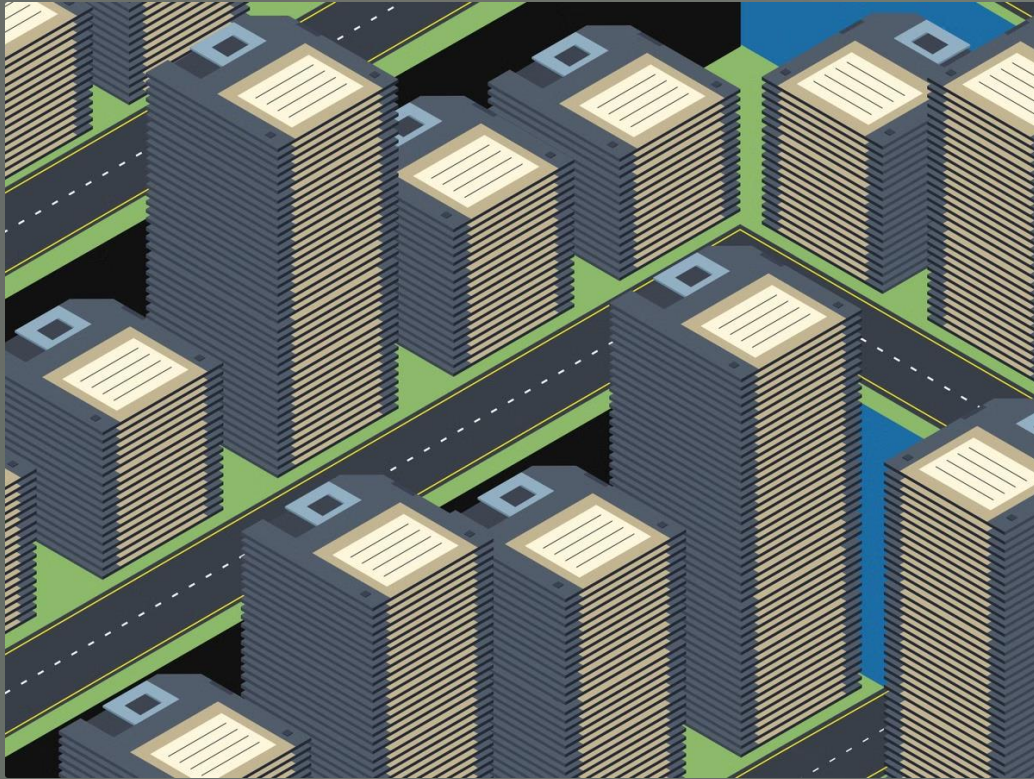
Checks if the queue has reached its maximum capacity and cannot accept new elements.



Constant Time

Both operations can be performed in constant time, $O(1)$, to quickly determine the queue's status.

	Time Complexity	Space Complexity
enqueue	$O(1)$	$O(n)$
dequeue	$O(1)$	$O(n)$
peek	$O(1)$	$O(n)$
isEmpty	$O(1)$	$O(n)$



Memory Stack in Function Calls

The memory stack is a dynamic data structure managed in RAM that follows the Last-In-First-Out (LIFO) principle. It plays a crucial role in handling function calls, especially in recursive and nested scenarios.

What is the Memory Stack?

The Building Blocks Of Stack Memory



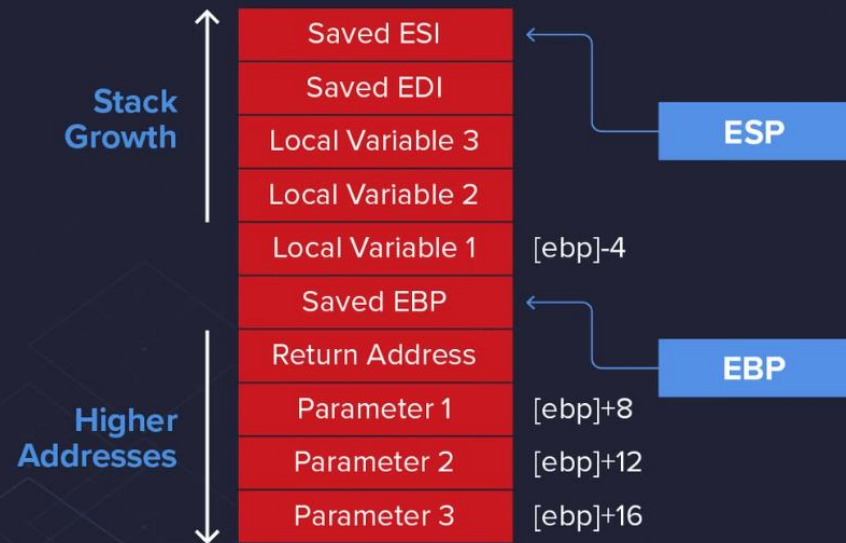
Stack frames are the foundation of stack memory.



ESB, aka "frame pointers," are an unchanging reference point for data on the stack.



ESP, aka "stack pointers," point to the next item on the stack.



VARONIS

1

Managed in RAM

The memory stack is a region in the computer's random access memory (RAM) used to store information about the active subroutines of a computer program.

2

LIFO Structure

The stack follows the Last-In-First-Out (LIFO) principle, meaning the last item pushed onto the stack is the first one to be removed or "popped" off.

3

Handles Function Calls

The memory stack is essential for managing function calls, especially in recursive and nested scenarios, by storing return addresses and function parameters.

Memory Stack Operations

Push

The Push operation stores the function's return address and parameters on the top of the stack.

Pop

The Pop operation retrieves the return address from the top of the stack after the function completes, allowing the program to continue execution.

Peek

The Peek operation allows the program to view the top of the stack without removing the item, useful for inspecting the state of the stack.



The Stack Pointer (SP)



Tracks the Top

The Stack Pointer (SP) keeps track of the top of the memory stack, allowing efficient push and pop operations.



Increments on Push

When a new item is pushed onto the stack, the SP is incremented to point to the new top of the stack.



Decrements on Pop

When an item is popped from the stack, the SP is decremented to point to the new top of the stack.



Holds Memory Address

The SP holds the memory address of the top of the stack, enabling efficient access to the stored data.

Function Call Handling

When a function is called, the CPU needs to carefully manage the function's state to ensure proper execution and return to the main program. This involves a series of steps to push and pop data on the program stack.

```
<div style="margin:9>  
<a name="www"></a>  
<table width="500% border=10" _ align=center" _9></a>  
  <tr>  
    <td height="68" width="256" colspan="8" padding:  
    <td> <form name=login method=post action=</a>  
    <input type=hidden name=action value=login</a>  
    ... align="left" cellpa
```

Recursive Function Calls

1

Function Call

When a function is called, its return address and parameters are pushed onto the stack.

2

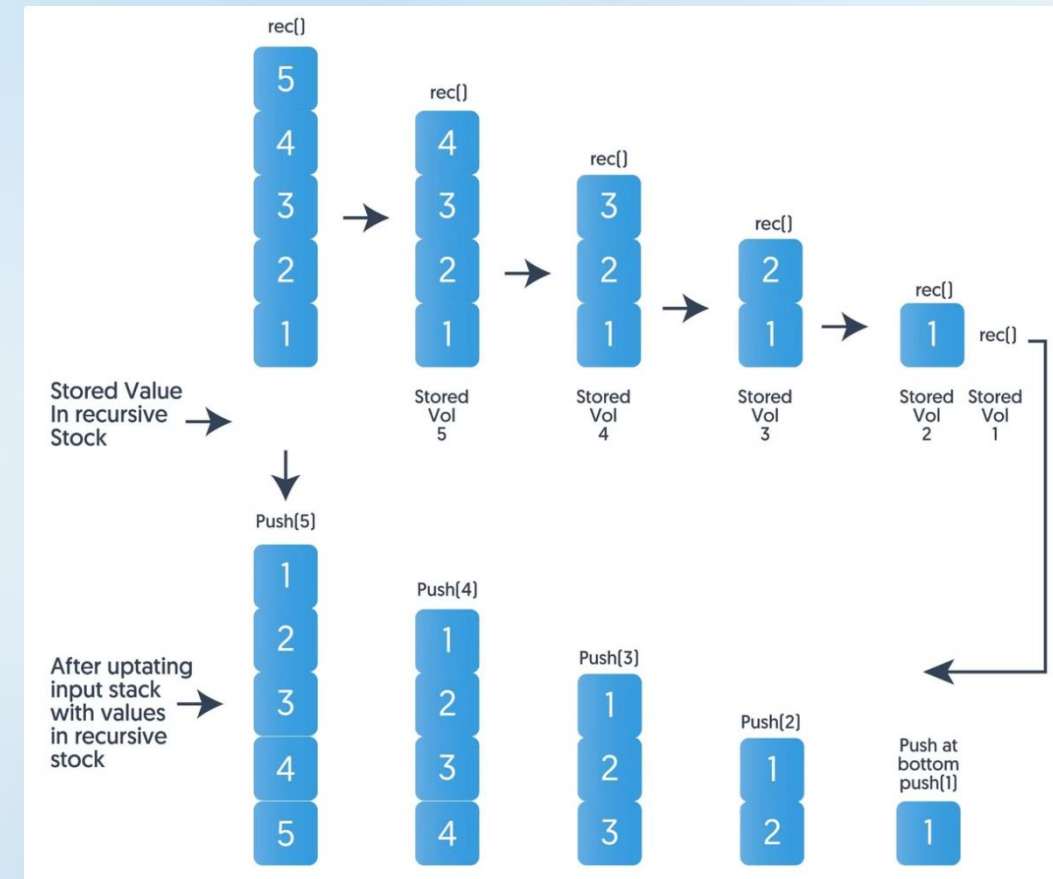
Nested Calls

If the function calls itself (recursively) or another function, the new call information is added to the top of the stack.

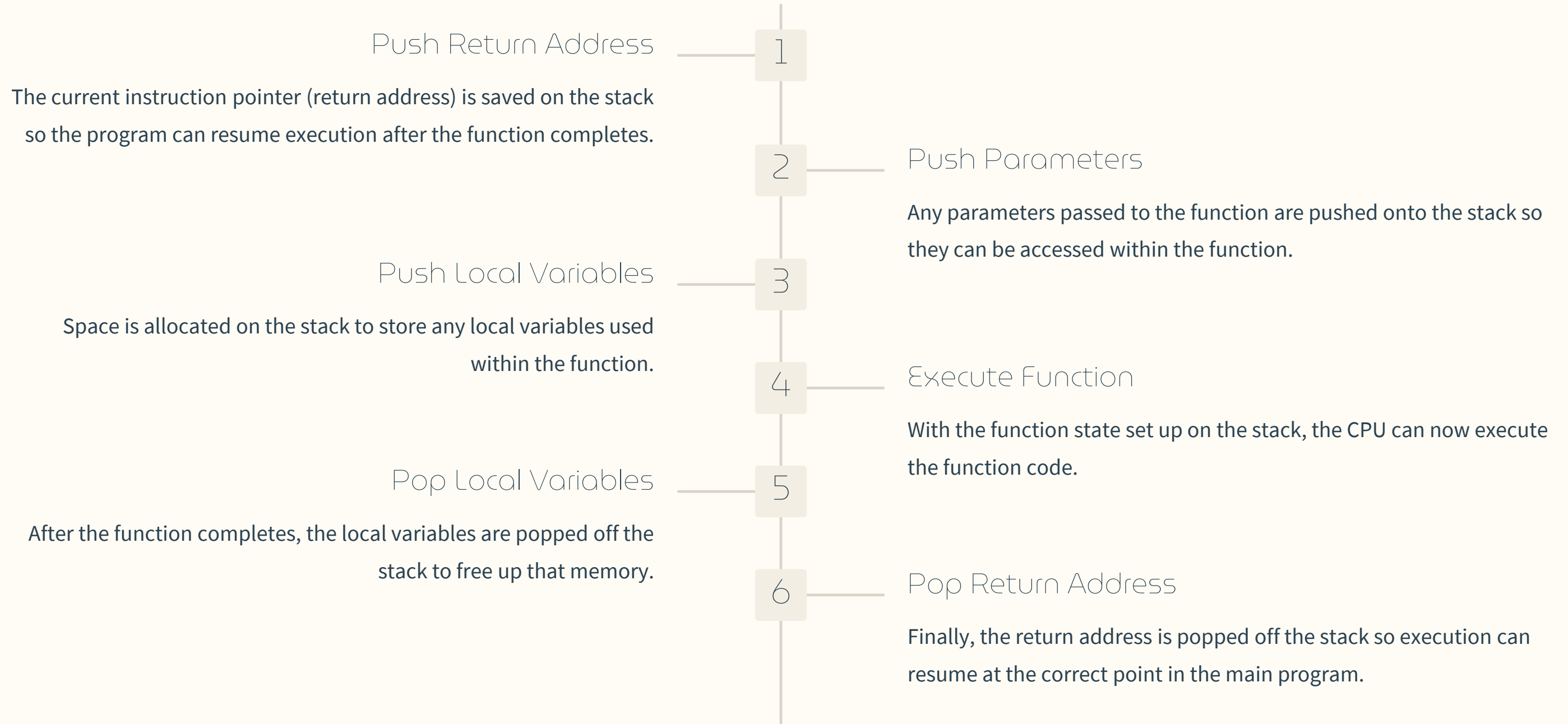
3

Function Return

When the function completes, the top item on the stack is popped, retrieving the return address to continue execution.



Pushing and Popping Function State





Conclusion

This concrete example highlights the importance of understanding data structures and algorithms in software development. By analyzing the performance differences between sorting algorithms, we can make informed choices that optimize our applications.

I encourage you to continue exploring these concepts and to ask any questions you may have. Mastering data structures and algorithms is a critical skill that will serve you well throughout your career.