Data Structures and Algorithms

Introduction

In the rapidly evolving world of software development, the efficient management of data and the use of structured approaches play a crucial role in the success of any project. Abstract Data Types (ADTs) are essential building blocks in modern software engineering, as they provide a way to define data structures independently from their implementation. By separating the "what" from the "how," ADTs enhance modularity, reduce complexity, and promote maintainability throughout the software development lifecycle.

content

I Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

II Determine the operations of a memory stack and how it is used to implement function calls in a computer.

I)CREATE A DESIGN SPECIFICATION FOR DATA STRUCTURES, EXPLAINING THE VALID OPERATIONS THAT CAN BE CARRIED OUT ON THE STRUCTURES.

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Identify

- A collection of elements, each identified by an index or key.
- Fixed size and elements are stored in contiguous memory locations.

Define the Operations

-Access: Retrieve an element at a specific index.
-Update: Modify an element at a specific index.
-Insert: Add a new element (usually at the end, or at a specified index if dynamic arrays are used).
-Delete: Remove an element from a specific index.

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Specify Input Parameters

Access:

- **Input: int index**
- **Description: The index of the element to access.**

Update:

- **Input: int index, T newValue**
- **Description: The index of the element to be updated and the new value.**

Insert:

- **Input: int index, T newValue**
- **Description: The index where the new element should be inserted and the value to insert.**

Delete:

- **Input: int index**
- **Description: The index of the element to be deleted.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Pre- and post-conditions

Access

- **Pre-condition: The index must be within the bounds of the array (0 <= index < array.length).**
- **Post-condition: Returns the element at the specified index.**

Update

- **Pre-condition: The index must be within bounds, and the array must be non-empty.**
- **Post-condition: The element at the specified index is replaced by the new value.**

Insert

- **Pre-condition: For static arrays, the array must not be full. For dynamic arrays, sufficient memory must be available.**
- **Post-condition: The new element is inserted, and subsequent elements (if any) are shifted to the right.**

Delete

- **Pre-condition: The index must be within bounds, and the array must contain elements.**
- **Post-condition: The element at the specified index is removed, and subsequent elements are shifted to the left.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Time and Space Complexity

- **Access:**
  - **Time Complexity: O(1) (constant time, as arrays allow direct access by index).**
  - **Space Complexity: O(1) (no additional memory is needed).**
- **Update:**
  - **Time Complexity: O(1) (directly updating the element by index).**
  - **Space Complexity: O(1).**
- **Insert (end or specific index):**
  - **Time Complexity:**
    - **Inserting at the end (dynamic array): O(1) amortized.**
    - **Inserting at a specific index: O(n), as elements need to be shifted.**
  - **Space Complexity: O(1) (unless resizing the array, which may take O(n) for copying elements to a larger array).**
- **Delete:**
  - **Time Complexity: O(n), as elements need to be shifted to fill the gap.**
  - **Space Complexity: O(1).**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Identify

- **A sequence of elements, where each element (node) contains data and a reference (or pointer) to the next element in the sequence.**
- **Can be singly linked (each node points to the next) or doubly linked (each node points to both the next and previous).**

Define the Operations

-Access: Retrieve an element at a specific index.
-Update: Modify an element at a specific index.
-Insert: Add a new element (usually at the end, or at a specified index if dynamic arrays are used).
-Delete: Remove an element from a specific index.

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Specify Input Parameters

Insert at Head:

- **Input: T newValue**
- **Description: The new value to be inserted at the head of the list.**

Insert at Tail:

- **Input: T newValue**
- **Description: The new value to be inserted at the tail of the list.**

Delete Node:

- **Input: T value**
- **Description: The value of the node to be deleted.**

Search:

- **Input: T value**
- **Description: The value to search for in the linked list.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Pre- and post-conditions

Insert at Head

- **Pre-condition: None.**
- **Post-condition: A new node is created, and the head now points to this new node.**

Insert at Tail

- **Pre-condition: None.**
- **Post-condition: A new node is created and linked at the end of the list. The tail now points to this new node.**

Delete Node

- **Pre-condition: The linked list must contain the node (by value or reference) to be deleted.**
- **Post-condition: The node is removed, and the previous node (if any) points to the next node.**

Search

- **Pre-condition: None.**
- **Post-condition: Returns the node if found; otherwise, returns null or an equivalent result.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Time and Space Complexity

- **Insert at Head:**

- **Time Complexity: O(1) (constant time to add a new head node).**
- **Space Complexity: O(1).**
- **Insert at Tail:**

- **Time Complexity: O(1) (if a tail pointer is maintained), otherwise O(n) for traversal.**
- **Space Complexity: O(1).**
- **Delete Node:**

- **Time Complexity: O(n) (traversal to find the node).**
- **Space Complexity: O(1).**
- **Search:**

- **Time Complexity: O(n) (linear search through the list).**
- **Space Complexity: O(1).**
-

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Identify

- **A collection that follows the Last In, First Out (LIFO) principle.**
- **Elements are added and removed from one end called the "top."**

Define the Operations

-Push: Add a new element to the top of the stack.
-Pop: Remove and return the element from the top of the stack.
-Peek (Top): Return the top element without removing it.
-isEmpty: Check if the stack is empty.

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Specify Input Parameters

Push:

- **Input: T newValue**
- **Description: The value to push onto the stack.**

Pop:

- **Input: No input parameters.**
- **Description: Removes the top value from the stack and returns it.**

Peek (Top):

- **Input: No input parameters.**
- **Description: Returns the top value without removing it.**

isEmpty:

- **Input: No input parameters.**
- **Description: Checks if the stack is empty.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Pre- and post-conditions

Push

- **Pre-condition: The stack must not be full (if using a static implementation like an array-based stack).**
- **Post-condition: The new element is placed on top of the stack.**

Pop

- **Pre-condition: The stack must not be empty.**
- **Post-condition: The top element is removed and returned, and the second-to-top element (if any) becomes the new top.**

Peek (Top)

- **Pre-condition: The stack must not be empty.**
- **Post-condition: The top element is returned, but the stack remains unchanged.**

isEmpty

- **Pre-condition: None.**
- **Post-condition: Returns true if the stack is empty, false otherwise.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Time and Space Complexity

Push:

- **Time Complexity: O(1) (adding to the top of the stack).**
- **Space Complexity: O(1).**

Pop:

- **Time Complexity: O(1) (removing from the top of the stack).**
- **Space Complexity: O(1).**

Peek (Top):

- **Time Complexity: O(1) (constant access to the top).**
- **Space Complexity: O(1).**

isEmpty:

- **Time Complexity: O(1).**
- **Space Complexity: O(1).**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Identify

- **A collection that follows the First In, First Out (FIFO) principle.**
- **Elements are added at the back (tail) and removed from the front (head).**

Define the Operations

-Enqueue: Add a new element to the rear (tail) of the queue.
-Dequeue: Remove and return the element from the front (head) of the queue.
-Peek (Front): Return the front element without removing it.
-isEmpty: Check if the queue is empty.

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Specify Input Parameters

Enqueue:

- **Input: T newValue**
- **Description: The value to add to the queue.**

Dequeue:

- **Input: No input parameters.**
- **Description: Removes and returns the front value from the queue.**

Peek (Front):

- **Input: No input parameters.**
- **Description: Returns the front value without removing it.**

isEmpty:

- **Input: No input parameters.**
- **Description: Checks if the queue is empty.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Pre- and post-conditions

Enqueue

- **Pre-condition: The queue must not be full (for bounded implementations).**
- **Post-condition: The new element is added to the rear of the queue.**

Dequeue

- **Pre-condition: The queue must not be empty.**
- **Post-condition: The front element is removed and returned, and the next element becomes the new front.**

Peek (Front)

- **Pre-condition: The queue must not be empty.**
- **Post-condition: The front element is returned, but the queue remains unchanged.**

isEmpty

- **Pre-condition: None.**
- **Post-condition: Returns true if the queue is empty, false otherwise.**

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Time and Space Complexity

Enqueue:

- **Time Complexity: O(1) (adding to the end).**
- **Space Complexity: O(1).**

Dequeue:

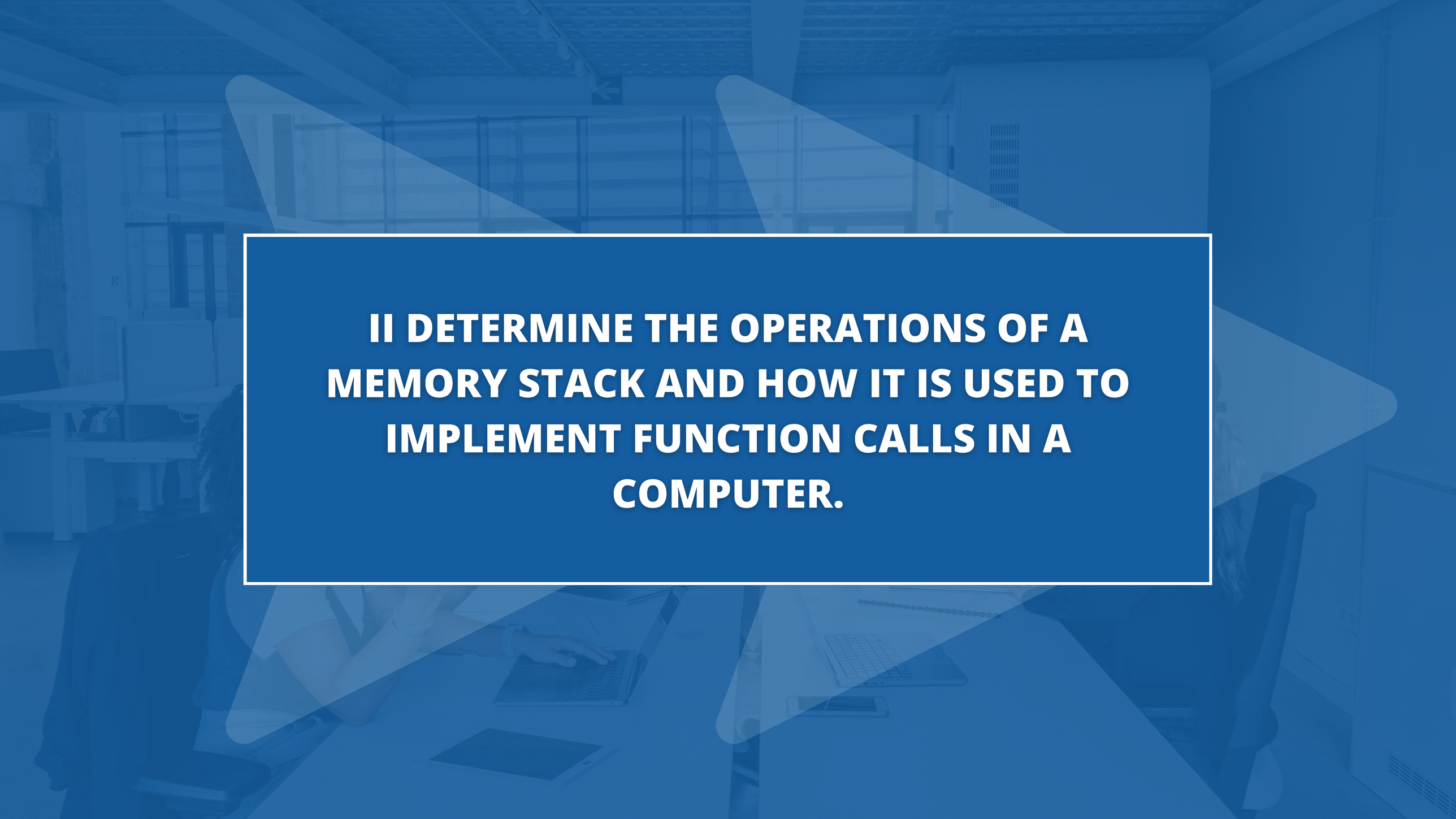
- **Time Complexity: O(1) (removing from the front).**
- **Space Complexity: O(1).**

Peek (Front):

- **Time Complexity: O(1).**
- **Space Complexity: O(1).**

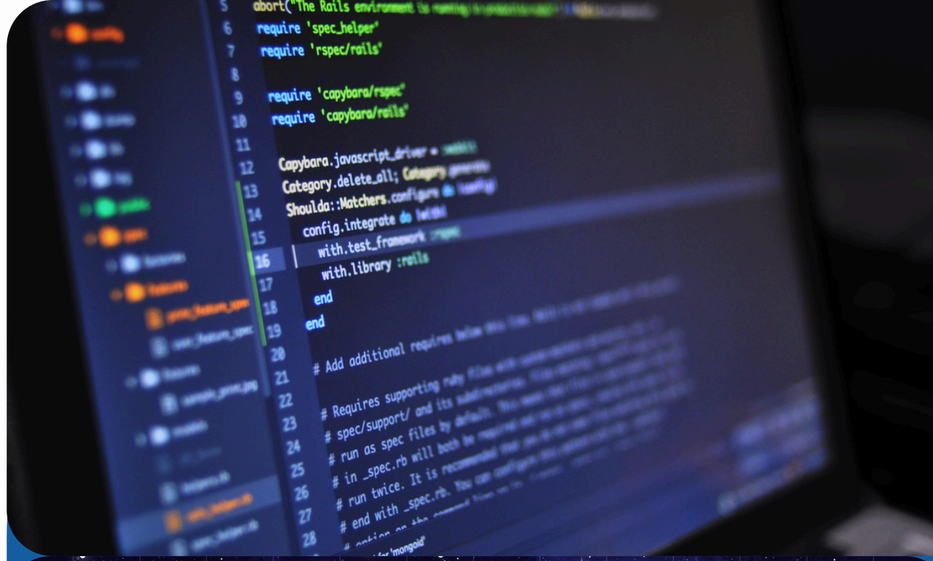
isEmpty:

- **Time Complexity: O(1).**
- **Space Complexity: O(1).**



II DETERMINE THE OPERATIONS OF A MEMORY STACK AND HOW IT IS USED TO IMPLEMENT FUNCTION CALLS IN A COMPUTER.

II Determine the operations of a memory stack and how it is used to implement function calls in a computer.



Define a Memory Stack

A memory stack is a section of memory that follows the Last In, First Out (LIFO) principle. It is primarily used for holding temporary data, such as function parameters, return addresses, and local variables. Whenever a function is invoked, a new stack frame is created at the top of the stack, and this frame is removed when the function completes.

Key Operations

- Push: Add an item (data) to the top of the stack.
- Pop: Remove and return the item from the top of the stack.
- Peek/Top: Retrieve the item at the top of the stack without removing it.
- IsEmpty: Determine if the stack is empty.
- Size: Return the total number of items currently in the stack.

Function Call Process

01

- Push the Return Address: The address of the instruction immediately following the function call is pushed onto the stack, enabling the program to return to the correct point once the function has finished executing.

02

- Allocate Space for Local Variables: Space is reserved on the stack for the local variables and parameters of the function.

Function Call Process

03

- Push Parameters: Any arguments passed to the function are added to the stack.

05

- Pop Stack Frame: After the function execution is complete, the stack frame is removed from the stack, reverting to the previous state, including the return address.

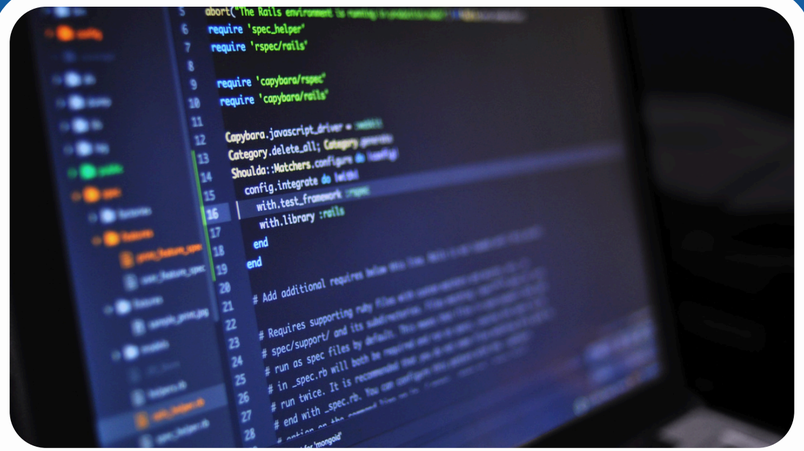
04

- Execute Function Code: The function runs using the local variables and parameters stored in the stack frame.

Illustration of Stack Frames

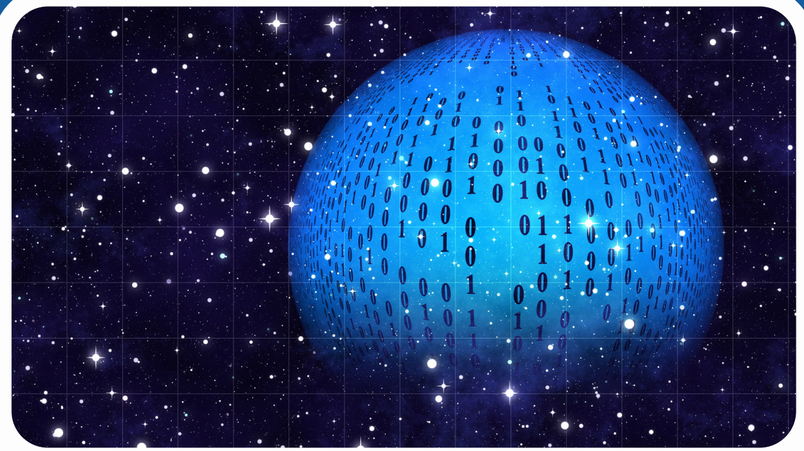
Return Address

The location to return to after the function execution.



Local Variables

Variables defined within the function.



Parameters

The arguments passed to the function.

EXAMPLE

Software

```
void functionA(int x) {
int y = x + 1;
functionB(y);
}

void functionB(int z) {

}
```

Stack Frame	Content
Frame for functionB	Return Address (to functionA), Parameter z
Frame for functionA	Return Address (to main), Parameter x, Local Variable y

Significance of the Memory Stack

Management of Function Calls

It oversees function calls and returns, ensuring that the execution flow remains accurate.

Efficient Memory Usage

The stack provides a limited, structured area for temporary data, which is automatically managed through allocation and deallocation as functions are invoked and completed.

Significance of the Memory Stack

Support for Recursion

The stack effectively handles recursive function calls by creating distinct stack frames for each invocation, which keeps local variables and return addresses separate.

Data Integrity

By utilizing separate stack frames, each function call can function independently, avoiding any interference with the data of other function calls.

THANK YOU!