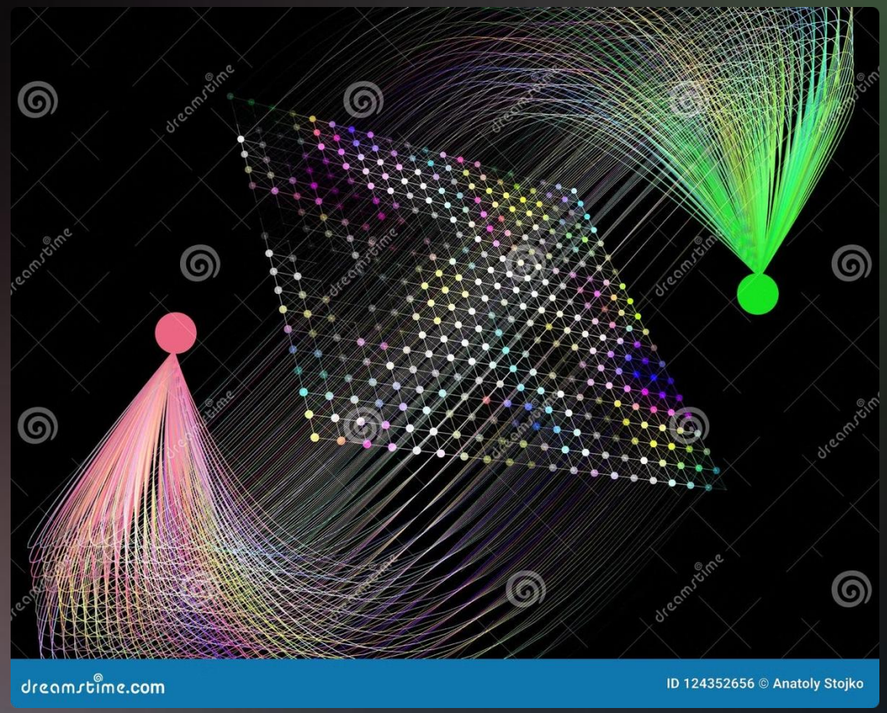
Introduction to Data Structures

Data structures are fundamental concepts in computer science that organize and store data efficiently. Creating a design specification for data structures is crucial for effective implementation and usage. This process involves identifying appropriate structures, defining operations, specifying parameters, establishing conditions, and analyzing complexities.

Understanding these elements allows developers to choose the right data structure for specific tasks, optimize performance, and create robust software solutions. This presentation will delve into each aspect of data structure design, providing insights and examples to guide you through the process.

N by Nguyen Hoang Tuan Anh (BTEC HN)



Create a design specification for data structures, explaining the valid operations that can be carried out on the structures. (P1)

1: Identifying Data Structures

Selecting the appropriate data structure is crucial for efficient problem-solving. Common data structures include arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each structure has unique characteristics suited for specific scenarios.

For instance, stacks are ideal for tasks requiring Last-In-First-Out (LIFO) operations, such as undo functionality in text editors. Queues, on the other hand, excel in First-In-First-Out (FIFO) scenarios, like managing print jobs. Understanding these distinctions helps in choosing the most effective structure for a given task.

Arrays

Contiguous memory allocation for fast access

Linked Lists

Dynamic size with efficient insertions and deletions

Trees

Hierarchical structure for efficient searching and sorting

Hash Tables

Fast lookups using key-value pairs

2: Defining Operations

Each data structure supports specific operations that define its behavior and functionality. For arrays, common operations include access, insertion, and deletion. Stacks utilize push, pop, and peek operations, while queues employ enqueue, dequeue, and peek. Trees support operations like insertion, deletion, and various traversal methods (in-order, pre-order, post-order).

Understanding these operations is crucial for effective implementation and usage of data structures. For example, a stack's push operation adds an element to the top, while pop removes the topmost element.

1. **Array Operations**
   Access, insertion, deletion
2. **Stack Operations**
   Push, pop, peek
3. **Queue Operations**
   Enqueue, dequeue, peek
4. **Tree Operations**
   Insert, delete, traversal

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Input Parameters for Operations

Each operation on a data structure requires specific input parameters. For instance, a binary search in an array typically needs the array itself, the target value, and the range (start and end indices). Queue operations like enqueue require the item to be added, while dequeue might not need any parameter as it removes the front element.

Understanding these parameters is essential for correctly implementing and using data structure operations. Proper parameter specification ensures that operations receive the necessary information to function correctly and efficiently.

Each operation needs specific **input parameters** to execute. For example:

Array:

- **Access**: The input parameter is the **index** of the element in the array.
- **Insert**: Requires the **index** and **value** of the element to insert.
- **Delete**: Requires the **index** of the element to delete.

Stack:

- **Push**: Input is the **value** of the element to add.
- **Pop**: No input needed (as you always remove the top element).

Defining Pre- and Post-conditions

Pre-conditions and post-conditions are crucial for ensuring the correct execution of data structure operations. Pre-conditions specify what must be true before an operation is performed. For example, before performing a pop operation on a stack, the stack must not be empty. Post-conditions define what will be true after the operation completes successfully. For instance, after a successful insertion into a list, the item must exist within the list.

These conditions help in maintaining data integrity and preventing errors during operation execution. They also serve as a form of documentation and aid in debugging and testing processes.

Stack Pop Pre-condition

Stack must not be empty before popping an element

List Insertion Post-condition

Item must exist in the list after successful insertion

Queue Dequeue Pre-condition

Queue must contain at least one element before dequeuing

Binary Search Pre-condition

Array must be sorted before performing binary search

Time and Space Complexity

Understanding time and space complexity is crucial for evaluating the efficiency of data structures and their operations. Time complexity refers to the amount of time an operation takes to complete, while space complexity refers to the amount of memory it requires. These are typically expressed using Big O notation.

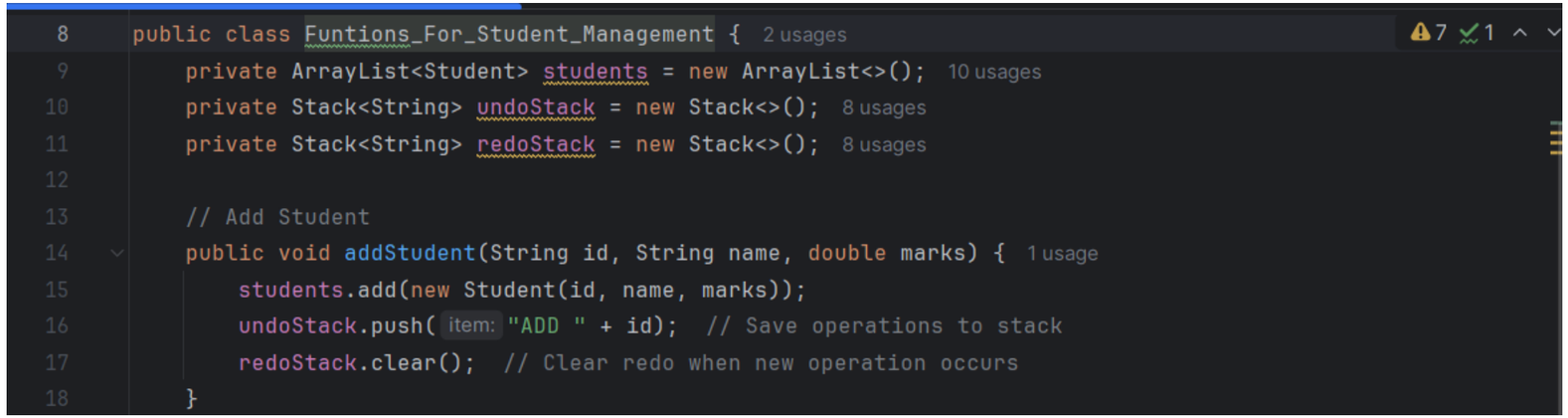
For example, array search operations have different complexities: linear search is O(n), while binary search is O(log n). Queue operations like enqueue and dequeue are typically O(1). Analyzing these complexities helps in choosing the most efficient data structure for specific use cases.

Data Structure	Operation	Time Complexity
Array	Linear Search	O(n)
Array	Binary Search	O(log n)
Stack	Push/Pop	O(1)
Queue	Enqueue/Dequeue	O(1)
Binary Search Tree	Search/Insert/Delete	O(log n)

Examples and Code Snippets

Practical examples and code snippets help illustrate the implementation of data structures and their operations. For instance, a stack implementation in Java might include push and pop methods. Similarly, inserting a node into a binary search tree involves traversing the tree and placing the new node in the correct position.

These examples demonstrate real-world usage of data structures and their operations. They highlight key areas related to input parameters, operation execution, and complexity considerations. Understanding these implementations aids in applying data structures effectively in various programming scenarios.


8 public class Funtions_For_Student_Management { 2 usages
7 1
9 private ArrayList<Student> students = new ArrayList<>(); 10 usages
10 private Stack<String> undoStack = new Stack<>(); 8 usages
11 private Stack<String> redoStack = new Stack<>(); 8 usages
12
13 // Add Student
14 public void addStudent(String id, String name, double marks) { 1 usage
15 students.add(new Student(id, name, marks));
16 undoStack.push(item: "ADD " + id); // Save operations to stack
17 redoStack.clear(); // Clear redo when new operation occurs
18 }

```java
//Edit Student
public void editStudent(String id, String newName, double newMarks) {  3 usages
    for (Student s : students) {
        if (s.getId().equals(id)) {
            undoStack.push( item: "EDIT " + id + " " + s.getName() + " " + s.getMarks());
            s.setName(newName);
            s.setMarks(newMarks);
            redoStack.clear();
            return;
        }
    }
}
```

```java
    // Delete student
    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;
            }
```

```java
// Find student by ID
private Student findStudentById(String id) {  4 usages
    for (Student s : students) {
        if (s.getId().equals(id)) {
            return s;
        }
    }
    return null;
}
```

```java
// Show student list
public void displayStudents() {  1 usage
    for (Student s : students) {
        System.out.println(s);
    }
}
```

```java
// Undo
public void undo() {  1 usage
    if (!undoStack.isEmpty()) {
        String action = undoStack.pop();
        String[] parts = action.split( regex: " ");

        switch (parts[0]) {
            case "ADD":
                redoStack.push( item: "ADD " + parts[1]);
                students.removeIf(s -> s.getId().equals(parts[1]));
                break;
            case "EDIT":
                redoStack.push( item: "EDIT " + parts[1] + " " + findStudentById(parts[1]).getName() + " "
                editStudent(parts[1], parts[2], Double.parseDouble(parts[3]));
                break;
            case "DELETE":
                redoStack.push( item: "DELETE " + parts[1]);
                students.add(new Student(parts[1], parts[2], Double.parseDouble(parts[3])));
                break;
        }
    } else {
        System.out.println("There is no action to undo.");
    }
}
```

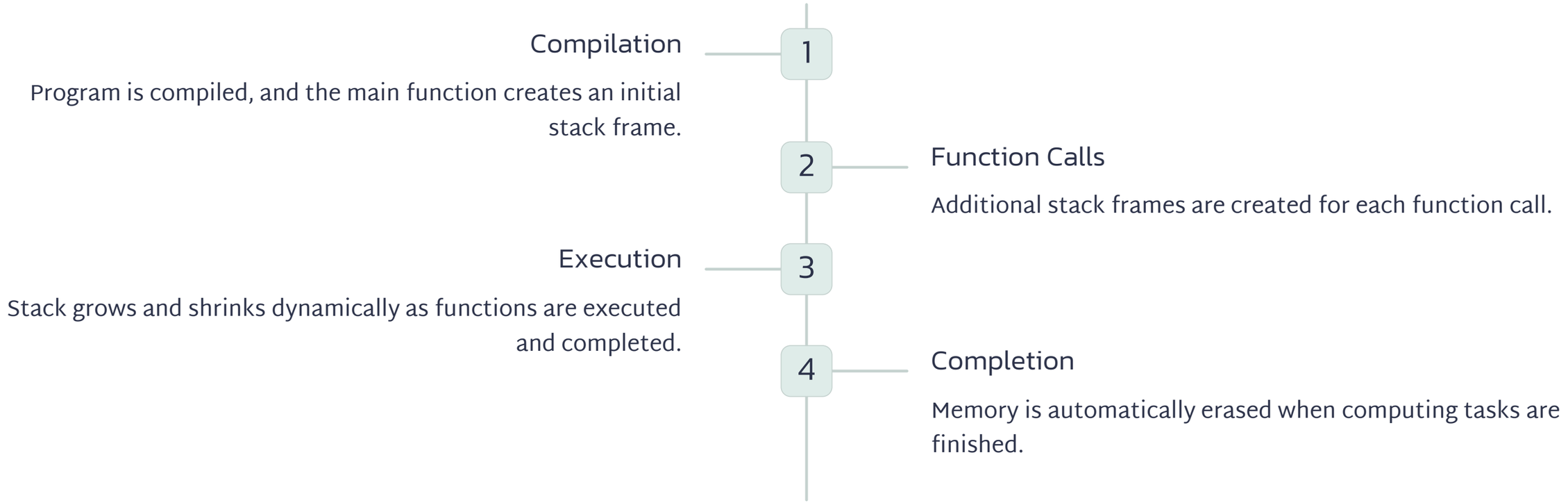
```java
    // Redo
    public void redo() {  1 usage
        if (!redoStack.isEmpty()) {
            String action = redoStack.pop();
            String[] parts = action.split( regex: " ");

            switch (parts[0]) {
                case "ADD":
                    undoStack.push( item: "ADD " + parts[1]);
                    students.add(new Student(parts[1],  name: "",  marks: 0));
                    break;
                case "EDIT":
                    undoStack.push( item: "EDIT " + parts[1] + " " + findStudentById(parts[1]).getName() + " "
                    editStudent(parts[1], parts[2], Double.parseDouble(parts[3]));
                    break;
                case "DELETE":
                    undoStack.push( item: "DELETE " + parts[1]);
                    students.removeIf(s -> s.getId().equals(parts[1]));
                    break;
            }
        } else {
            System.out.println("There is no action to redo.");
        }
    }
}
```

P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1: Defining a Memory Stack

A memory stack is a temporary storage structure created when a program is compiled. It begins with the main function and grows as additional functions are called. Each function call generates a stack frame, also known as an activation record, which contains all the data associated with that particular subprogram call.

The stack section primarily stores methods, local variables, and reference variables. It is dynamic, growing and shrinking as functions are called and returned, and it manages memory automatically. In multithreading environments, each thread typically has its own memory stack.



2: Identify Operations

2.1: Push Operation in Memory Stack

The push operation adds new items to the top of the stack. This occurs when a function is called, storing essential information such as the return address, function parameters, and local variables. As functions call other functions, the stack grows by adding new frames for each subsequent function.

1

Function Call

A new function is called, initiating the push operation.

2

Store Information

Return address, parameters, and local variables are pushed onto the stack.

3

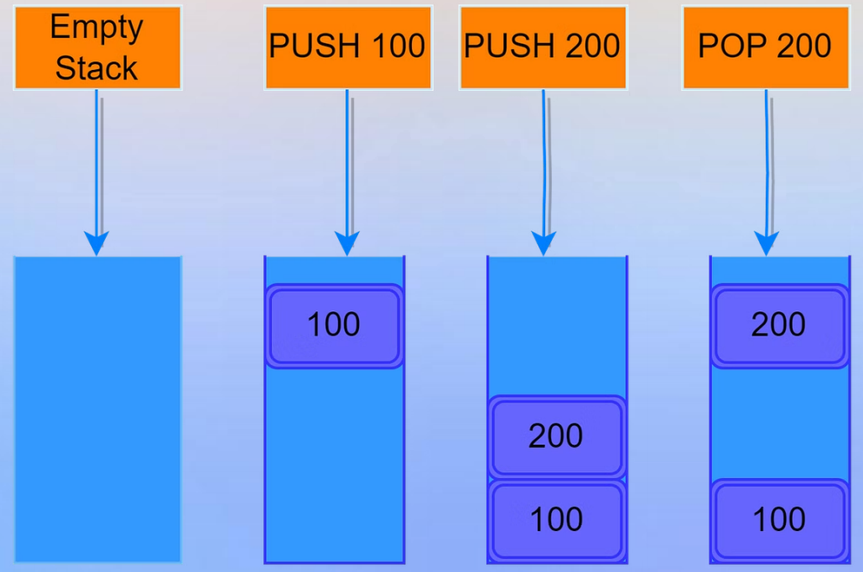
Stack Growth

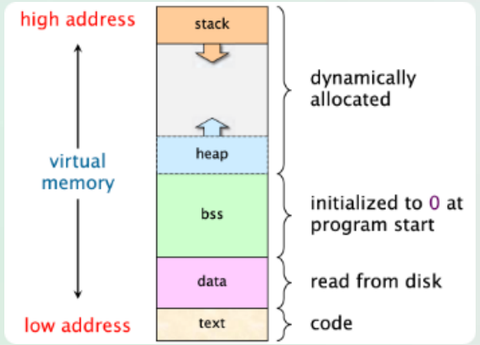
The stack expands to accommodate the new function's frame.

4

Nested Calls

If the function calls another function, the process repeats, further growing the stack.





2.2: Pop Operation in Memory Stack

The pop operation removes the top item from the stack when a function completes its execution. This process removes all information related to that function, including local variables, parameters, and the return address. The pop operation restores the program's previous state, allowing execution to continue from where it left off before the function call.

1 **Function Completion**

When a function finishes executing, the pop operation is triggered.

2 **Frame Removal**

The current stack frame is removed, freeing up memory.

3 **State Restoration**

The program's state is restored to its condition before the function call.

4 **Execution Continuation**

The program resumes execution from the point where it left off.

Peek Operation and Stack Overflow

The peek (or top) operation allows examination of the top item on the stack without removing it. This is useful for checking the current state of the stack without altering it, such as during debugging or managing execution flow. Stack overflow occurs when the stack grows beyond its allocated memory limit, typically due to excessive nested function calls or large amounts of memory used for local variables or recursion.

Peek Operation

- Examines top item without removal - Useful for debugging - Monitors current function execution

Stack Overflow

- Occurs when stack exceeds memory limit - Caused by excessive nested calls - Can result from large local variables or recursion

3: Function Call Implementation

Creating a Stack Frame

When a function is called, a new stack frame is created. This frame contains the return address, function parameters, local variables, and saved registers. The return address stores where the program should return after the function completes. Function parameters are stored for access within the function. Space is allocated for local variables, and CPU registers may be temporarily stored to restore the state after function completion.

Return Address

Stores the address where the program should return after function completion.

Function Parameters

Values passed to the function are stored in the stack frame for access.

Local Variables

Space is allocated for variables declared inside the function.

Saved Registers

CPU registers may be temporarily stored to restore state after function completion.

Executing and Completing Functions

During function execution, the processor uses values in the stack frame to perform operations and control program flow. Local variables are accessed and used as needed. When the function completes, its stack frame is popped off the stack, freeing up memory. The return address is used to continue executing the program from where the function was called, and any saved registers are restored to their pre-function call state.

1. **Execution**

   Processor uses stack frame values for operations and flow control.

2. **Completion**

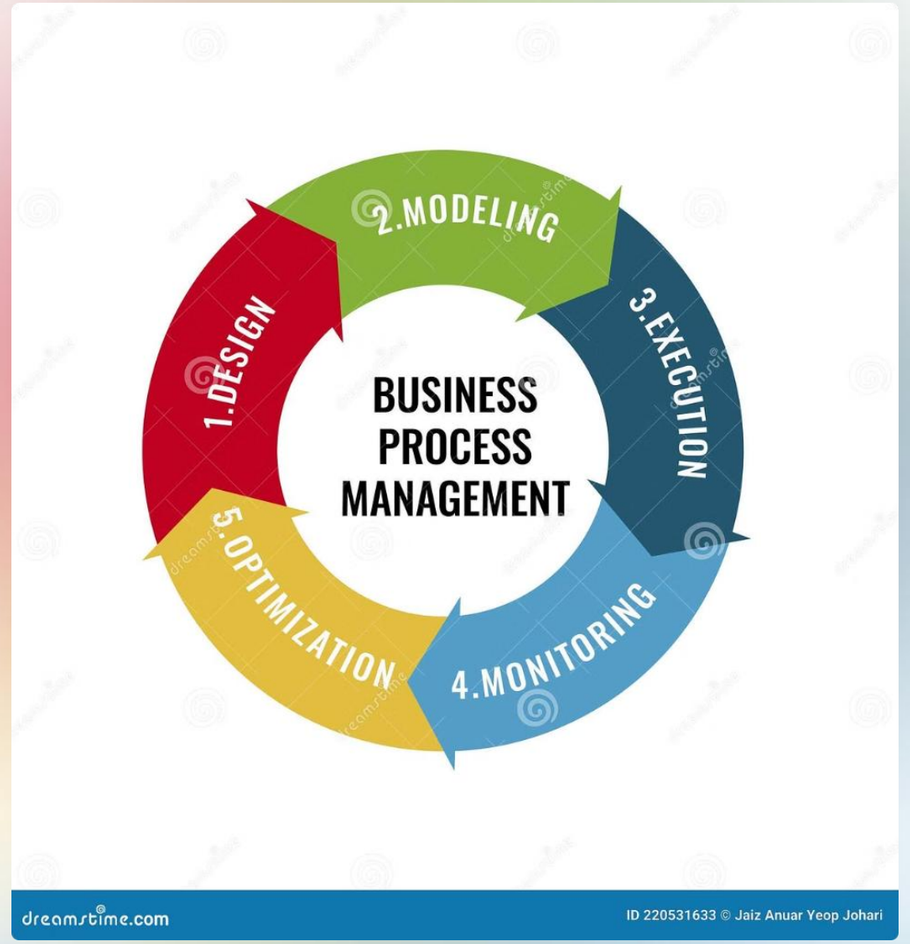
   Function finishes its tasks.

3. **Frame Release**

   Stack frame is popped, freeing memory.

4. **State Restoration**

   Return address and registers are restored.



4: Demonstrate Stack Frames

Importance of Stack Frames

Stack frames are crucial for automatic memory management, allocating and freeing memory for function calls. They support recursion by creating separate states for each recursive call. Stack frames also preserve the context of each function, ensuring integrity when handling complex function calls. This system allows for efficient memory usage and helps maintain the proper flow of program execution.

Automatic Memory Management

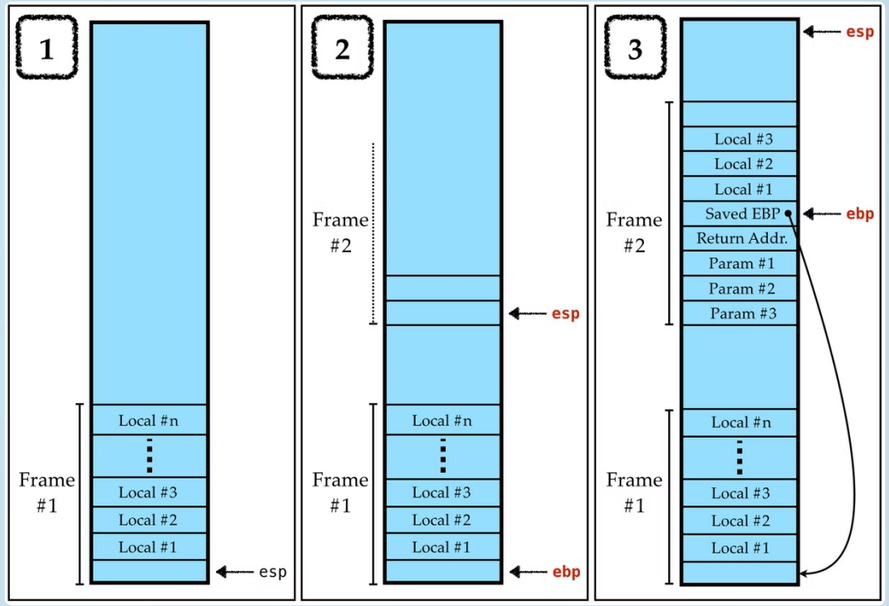
Efficiently allocates and frees memory for function calls.

Recursion Support

Maintains separate states for each recursive call.

Context Preservation

Ensures integrity during complex function calls.



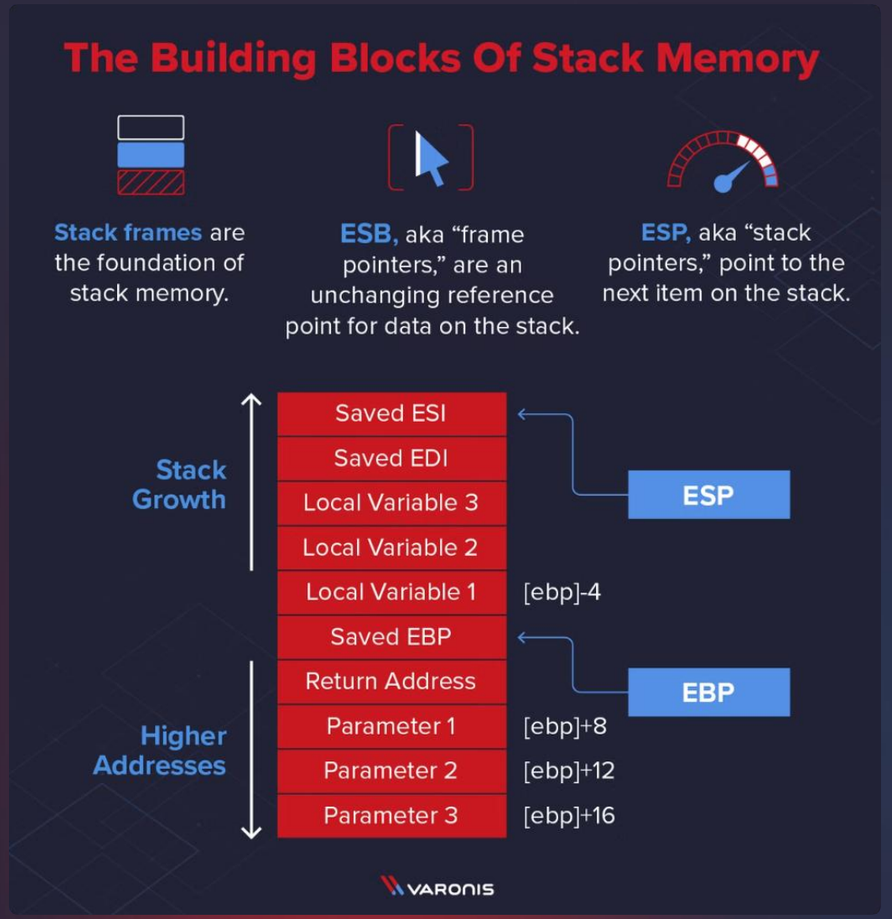
Structure and Features of Stack Frames

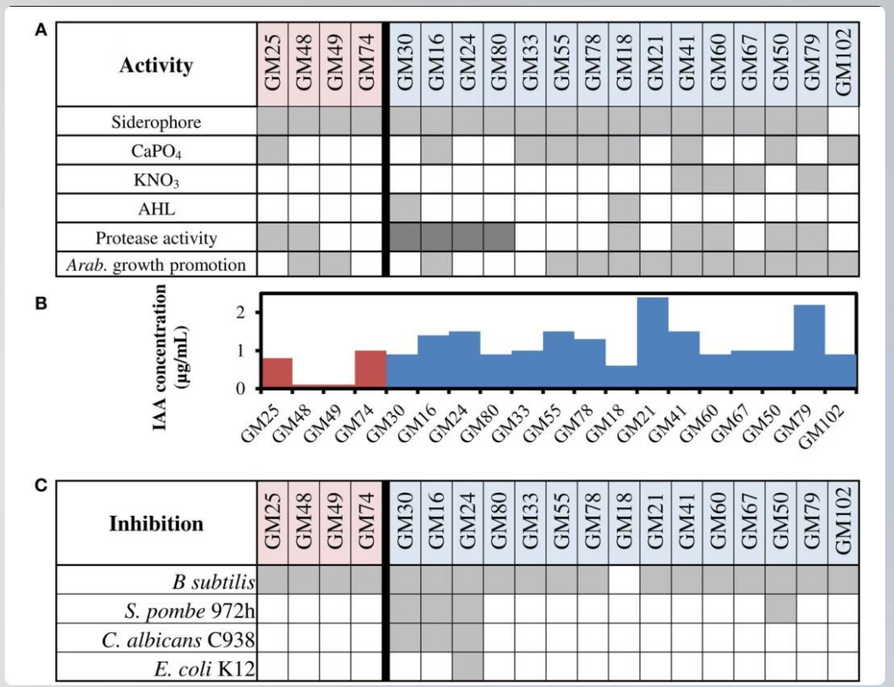
Each stack frame maintains a Stack Pointer (SP) and Frame Pointer (FP), which point to the top of the stack. A Program Counter (PC) points to the next instruction to be executed. The memory allocated for a function call in the stack exists only during the function's execution. Once completed, the variables become inaccessible. Stack frames support nested function calls, but excessive recursion can lead to stack overflow due to limited stack memory.

Component	Function
Stack Pointer (SP)	Points to top of stack
Frame Pointer (FP)	Stores address of whole stack frame
Program Counter (PC)	Points to next instruction
Local Variables	Stored within stack frame
Saved Registers	Temporarily stored in frame

5: Discuss the Importance of Stack Frames

Stack frames are fundamental components in the execution of function calls and memory management within computer systems. They play a vital role in ensuring efficient, organized, and secure program execution. From managing local variables to handling recursive functions, stack frames are essential for maintaining the integrity and flow of computer programs. Let's explore the key reasons why stack frames are so important:





5.1: Function Isolation and Local Variable Management

Dedicated Storage

Each function call has its own stack frame, storing local variables, function parameters, and return addresses.

Isolation

Ensures local variables of one function do not interfere with those of another, providing memory safety.

Temporary Access

Variables are only valid for the duration of the function's execution, helping to avoid memory corruption.

This isolation is crucial for maintaining the integrity of function execution and preventing unintended side effects between different parts of a program.

5.2: Efficient Management of Function Calls

1. **Function Call**

   When a function is called, a new stack frame is created and pushed onto the stack.

2. **Execution**

   The function executes within its dedicated stack frame.

3. **Completion**

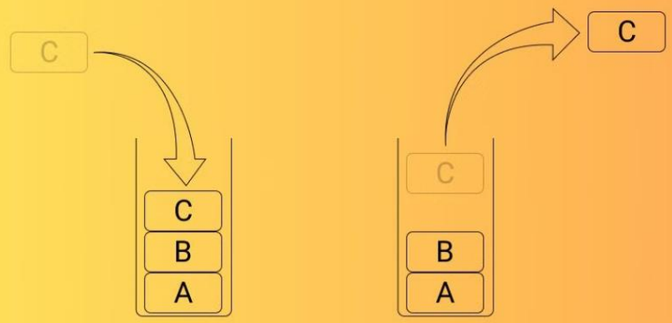
   Once the function completes, its stack frame is popped, returning control to the calling function.

This last-in, first-out (LIFO) organization ensures a smooth, predictable execution flow and enables nested function calls. It allows one function to call another and resume execution after the called function finishes, maintaining program coherence.

1/3

A call stack is composed of stack frames. They are essentially data structures that contain information about subroutines to call. A stack frame is just the entry and exit code for the routine, that pushes an anchor to the stack. This is very useful for debugging and error handling.
The contents of a stack frame are:

1. The function to be invoked and it's locals
2. The parameters for the function
3. An address to the callers frame (The current line number or return address)



@fejiroofficial

5.3: Handling Recursive Function Calls

Separate Instances

Each recursive call creates a new stack frame, allowing separate instances of local variables and return addresses for each call.

Preventing Overwrite

This structure prevents overwriting of previous values, ensuring correct results in recursive operations.

Stack Overflow Risk

Excessive recursion can lead to stack overflow if too many stack frames are created, highlighting the limitations of stack memory.

The ability to handle recursion effectively demonstrates both the power and the constraints of stack-based memory management in computer systems.

5.4: Program Counter and Flow Control

1

Store Address

Each stack frame contains a Program Counter (PC) storing the address of the next instruction.

2

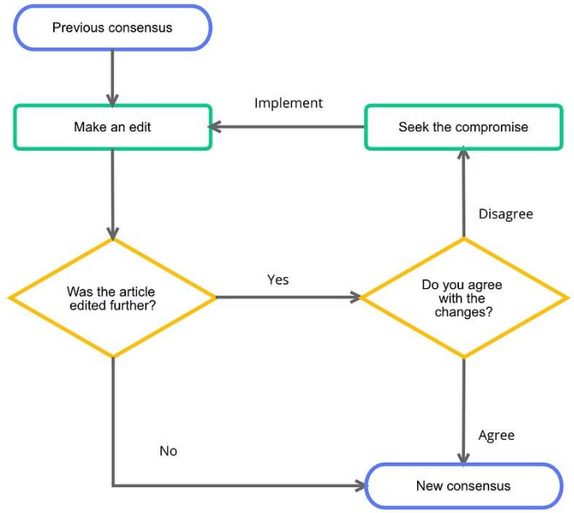
Manage Flow

Ensures proper management of control flow within the program.

3

Resume Execution

Allows the system to know where to continue execution after a function call.



The Program Counter is crucial for maintaining organized and coherent program execution, preventing disorganized or broken execution that could result from improper function call management.

5.5: Stack Pointer and Frame Pointer

Stack Pointer (SP)

Points to the top of the current stack frame.

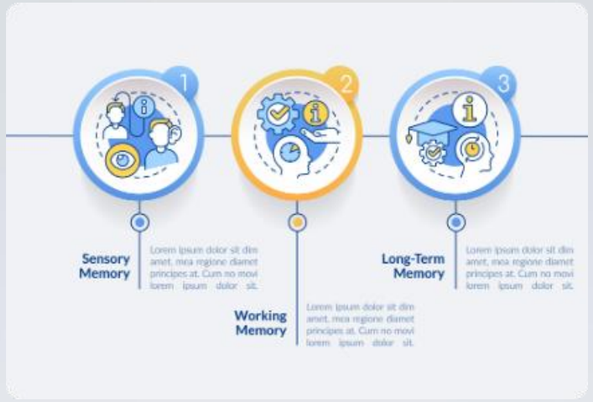
Frame Pointer (FP)

Points to the start of the current stack frame.

Tracking

Critical for tracking the location of function parameters, local variables, and return addresses in memory.

These pointers are essential for the dynamic growth and shrinkage of the stack as functions are called and completed. They ensure consistent and accurate access to stack frame components throughout program execution.



5.6: Memory Efficiency

1 **Automatic Allocation**

Memory is allocated for each function when it is called, ensuring resources are available when needed.

2 **Immediate Deallocation**

Memory is deallocated as soon as the function completes, freeing up resources for other operations.

3 **Dynamic Management**

The stack grows and shrinks as needed, providing efficient memory usage without manual management.

This automatic and dynamic memory management ensures optimal use of finite stack memory, contributing to overall system efficiency and performance.

5.7: Return to Calling Function

Function Call	Return Address Saved
Function Execution	Stack Frame Active
Function Completion	Return to Saved Address

After a function completes, the return address saved in the stack frame ensures that execution continues exactly where the function was called. This seamless transfer of control is essential for nested function calls and complex program flow. It allows programs to call multiple functions and subroutines while ensuring that each call returns control to the proper place, maintaining the integrity of program execution.

5.8: Stack Overflow and Limits

Stack Overflow

Occurs when too many recursive calls or deeply nested function calls are made without completing previous ones, exhausting the stack's limited memory.

Memory Limits

Highlights the importance of careful memory and function management in software development to prevent crashes and ensure stable program execution.

The importance of stack frames also highlights potential issues such as stack overflow. Because the stack has limited memory, if too many recursive calls or deeply nested function calls are made without completing previous ones, the stack can run out of space, leading to an overflow. This reinforces the importance of careful memory and function management in software development.

Conclusion

Stacks are not only a simple data structure but also play an important role in many areas of programming and data processing. The skillful application of operations such as push, pop, peek, and checking the stack state not only helps programmers manage data more easily but also optimizes the memory and performance of the program. From managing function calls, evaluating expressions to supporting features such as undo in applications, stacks have proven their usefulness and flexibility. Through this article, readers will have an overview and better understand the importance of stacks in programming and practical applications.