**Performance Analysis of Stack and Queue Implementations in C++**

## **Introduction**

This document presents a performance analysis of the stack and queue data structures implemented in C++ as part of a simulation of a back-and-forth conversation between two individuals.

The stack stores messages for one individual, while the queue handles the other's messages.

The key performance aspects analyzed include memory management, execution time, and data structure efficiency.

### **Memory Management**

* The stack and queue are dynamically implemented using linked lists, allowing flexible memory allocation.
* Each node in the stack and queue consumes memory for a message array and a pointer, leading to efficient memory usage without significant overhead.
* The dynamic nature of linked lists prevents the wastage of memory, unlike fixed-size arrays.

### **Execution Time**

* Stack operations (**push** and **pop**) operate in constant time, O(1), as they involve adding or removing elements from the top of the stack.
* Similarly, queue operations (**enqueue** and **dequeue**) also operate in O(1) time, as elements are added at the rear and removed from the front.
* The **displayConversation** function, which alternates messages from the stack and queue, has a time complexity proportional to the total number of messages, making it O(n).

### **Data Structure Efficiency**

* The stack, with its LIFO (Last In, First Out) principle, effectively reverses the order of one individual's messages.
* The queue, following FIFO (First In, First Out), maintains the chronological order of messages for the other individual.
* The combined use of these data structures in alternating messages simulates a real-life conversation flow effectively.

### **Using Stack and Queue:**

Eric's messages are stored in a stack and Amoh's in a queue.

### **Explanation of why this works:**

* The stack preserves the order of messages as they are pushed onto it, making it suitable for displaying messages in the order they were added.
* The queue, when implemented to store messages in reverse order, allows you to retrieve messages in the correct order for playback, as it follows the FIFO behavior.

## **API Documentation:**

* **push**: Adds a message to the stack.
* **pop**: Removes a message from the stack.
* **enqueue**: Adds a message to the queue.
* **dequeue**: Removes a message from the queue.
* **display**: Displays messages from a stack or queue.

#### **Conclusion**

The implemented stack and queue in C++ demonstrate efficient memory usage and execution time, making them suitable for applications requiring dynamic data handling like chat simulations.

The linked list-based implementation ensures flexible memory management, and the constant time complexity of basic operations guarantees fast execution.

The overall performance is therefore highly efficient and effective for the intended simulation purpose.