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++, which are used to simulate 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:

Person-1: Eric's messages are stored in a stack and Person-2: Amoh's in a queue.

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.

Test Cases for Edge Scenarios

A list of test cases that explore edge scenarios in our Stack and Queue implementations. This includes cases for normal operations, underflow, and overflow scenarios:

Test Case ID	Scenario	Input	Expected Output	Notes
TC001	Add to an empty stack	Push("Hello")	Stack contains one element: ["Hello"]	Basic functionality of stack insertion

Test Case ID	Scenario	Input	Expected Output	Notes
TC002	Add to a full stack	Fill stack to a predefined limit (e.g., 100) then Push("Extra")	Error: "Stack Overflow"	Add an overflow condition using a predefined limit
TC003	Remove from empty stack	Pop()	Error: "Stack is empty!"	Tests underflow condition in the stack
TC004	Add to an empty queue	Enqueue("World")	Queue contains one element: ["World"]	Basic functionality of queue insertion
TC005	Add to a full queue	Fill queue to a predefined limit (e.g., 100) then Enqueue("Extra")	Error: "Queue Overflow"	Add an overflow condition using a predefined limit
TC006	Remove from empty queue	Dequeue()	Error: "Queue is empty!"	Tests underflow condition in the queue
TC007	Display empty stack	Display()	Output: No messages to display	Ensures proper handling of empty stack
TC008	Display empty queue	Display()	Output: No messages to display	Ensures proper handling of empty queue
TC009	Large conversation dataset	Push and Enqueue 1,000,000 messages	Messages are successfully inserted and displayed in correct order	Tests the program's ability to handle large data
TC010	Alternate playback order	Stack and Queue with alternating conversation	Outputs the conversation alternately (stack	Ensures correct traversal for both structures simultaneously

Test Case ID	Scenario	Input	Expected Output	Notes
			message first, then queue message)	

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.