Report: Assignment 2 - Stacks and Queues (CSAI – 230, Fall 2024)

1. Assumptions

In this assignment, I was tasked with implementing a Queue ADT using two different approaches with the Stack ADT.

Assumptions made during this assignment are:

- In Big O analysis, we assumed that all data holds the same data type although the implementation uses ADT meaning the data type may vary affecting the Big O.
- In the second attempt we assumed that the linked list is independent of the stack, and that we are able to play with the order as we wish.

2. Design Approach

First Attempt:

In the first attempt, the Queue ADT was implemented using two stacks. The basic principle is:

1. **Enqueue operation**: Push the new element onto the first stack.
2. **Dequeue operation**: When performing a dequeue, if the second stack is empty, transfer all elements from the first stack to the second stack by repeatedly popping from the first stack and pushing to the second stack. Then, pop from the second stack to perform the dequeue.

This implementation works, but the dequeue operation has a time complexity of **O(n)** due to the need to transfer elements from one stack to the other. The more input we have the more this algorithm becomes inefficient.

Second Attempt (Optimised Approach):

To improve the performance, the second attempt introduces changes to the Stack implementation by:

1. **Adding a back pointer to the Stack**: The back pointer tracks the bottom of the stack, allowing for efficient access to the bottom of the stack.
2. **Maintaining a stack bottom pointer**: This pointer allows us to append new nodes directly at the stack bottom, making the **enqueue operation O(1)**.
3. **Optimising the dequeue operation**: Since the stack is reversed, we can pop the top element of the stack directly, making the **dequeue operation O(1)**.

Key Changes to the Stack:

- The **back pointer** allows traversal from the top to the bottom of the stack, making it possible to directly access the bottom for enqueue operations.
- The **stack bottom pointer** points to the bottom-most node in the stack. This pointer helps in appending new elements directly to the bottom of the stack during the enqueue operation, eliminating the need for costly stack reversals during enqueue.

By reversing the order of elements in the stack (with the bottom of the stack acting as the front of the queue), we can ensure that both enqueue and dequeue operations are **O(1)**.

3. Big-O Analysis

First Attempt:

1. **Enqueue Operation**:
    - **Time Complexity**: **O(1)**. This is because the enqueue operation only involves pushing an element onto the first stack.
    - **Space Complexity**: **O(1)**. We are only adding a single element to the stack.
2. **Dequeue Operation**:
    - **Time Complexity**: **O(n)**. In the worst case, if the second stack is empty, we need to transfer all the elements from the first stack to the second stack, which takes O(n) time.
    - **Space Complexity**: **O(n)**. We need extra space to hold all the elements temporarily in the second stack during the transfer.

Second Attempt:

1. **Enqueue Operation**:
    - **Time Complexity**: **O(1)**. With the optimised stack implementation, we append the new element directly to the bottom of the stack using the back pointer, making this operation constant time.
    - **Space Complexity**: **O(1)**. We only add one element to the stack each time.
2. **Dequeue Operation**:
    - **Time Complexity**: **O(1)**. Since the stack is reversed, the top element of the stack corresponds to the front of the queue. We can simply pop the top element in constant time.
    - **Space Complexity**: **O(1)**. The space complexity remains constant for dequeue, as it only removes the top element from the stack.

Comparison of Attempts:

- **First Attempt**: Enqueue is O(1), but Dequeue is O(n) due to the need for transferring elements between the stacks.
- **Second Attempt**: Both Enqueue and Dequeue operations are optimised to O(1) by using the back pointer and stack bottom pointer.

Thus, the second attempt provides a significant improvement in performance, reducing the time complexity of both operations to constant time.

4. Conclusion

The second attempt effectively optimises the Queue implementation by utilising an improved Stack with back and bottom pointers. This optimization reduces the time complexity of both **enqueue** and **dequeue** operations to **O(1)**, which makes the implementation more efficient compared to the first attempt, where **dequeue** was O(n).

While the second attempt with back pointers is highly optimised for this use case, there are additional optimizations that could be explored in other contexts. For instance, using a **single stack** with auxiliary data structures or introducing circular arrays could lead to further efficiency gains in specific scenarios.

By making these changes, I achieved the goal of the assignment, which was to enhance the performance of the Queue ADT while using only the primitive Stack operations (*push* and pop). This demonstrates the importance of careful data structure design and optimization techniques in achieving better runtime efficiency.