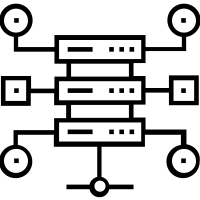
**Assignment No: \_1\_\_**

**Date: 26/2/2025**

**Title: Implementation and Performance Comparison of Stack, Queue, and Deque for Ticket Processing**

(Title based on the application domain and the data structure you will be implementing)

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment Type of Submission:** |  |  |  |
| **Group** | Yes | Yuxuan Song  24207239  Yujun Liu  24202574  Deepak Shelke  24208478 | **35**  **35**  **30** |

****

1. **Problem Domain Description:**

This project simulates a customer support ticketing system, where customers submit issues, and support agents process them in a first-come, first-served manner. However, urgent tickets (e.g., security issues, payment failures) must be processed before normal tickets (e.g., minor bugs, feature requests).

To achieve efficient ticket processing, we implemented three different data structures:

1. Queue-based approach: Uses two LinkedList-based queues for urgent and normal tickets.
2. Deque-based approach: Uses a single ArrayDeque, where urgent tickets are placed at the front, and normal tickets at the back.
3. Stack-based approach: Used for benchmarking, though not ideal for ticketing systems.

The system generates 10,000 random tickets and measures the performance of these three approaches.

1. **Theoretical Foundations of the Data Structure(s) utilised**

**This project models a customer support ticket system using Queue and Deque, ensuring efficient ticket processing while prioritizing urgent issues.**

Queue-Based Approach (LinkedList Implementation)

Concept: Two separate LinkedList-based queues process urgent tickets first, maintaining FIFO order within priority levels.

Operations:

offer(): Adds an element to the end (O(1))

poll(): Removes an element from the front (O(1))

Advantages: Simple priority management, clear separation of urgent and normal issues.

Deque-Based Approach (ArrayDeque Implementation)

Concept: Uses a single ArrayDeque, adding urgent tickets to the front and normal tickets to the back.

Operations:

addFirst(): Inserts an urgent ticket at the front (O(1))

addLast(): Inserts a normal ticket at the back (O(1))

pollFirst(): Processes from the front (O(1))

Advantages: Faster than LinkedList due to lower memory allocation overhead.

Stack-Based Approach (LinkedList Implementation)

Concept: Uses a single LinkedList-based stack, where tickets are processed last-in, first-out (LIFO).

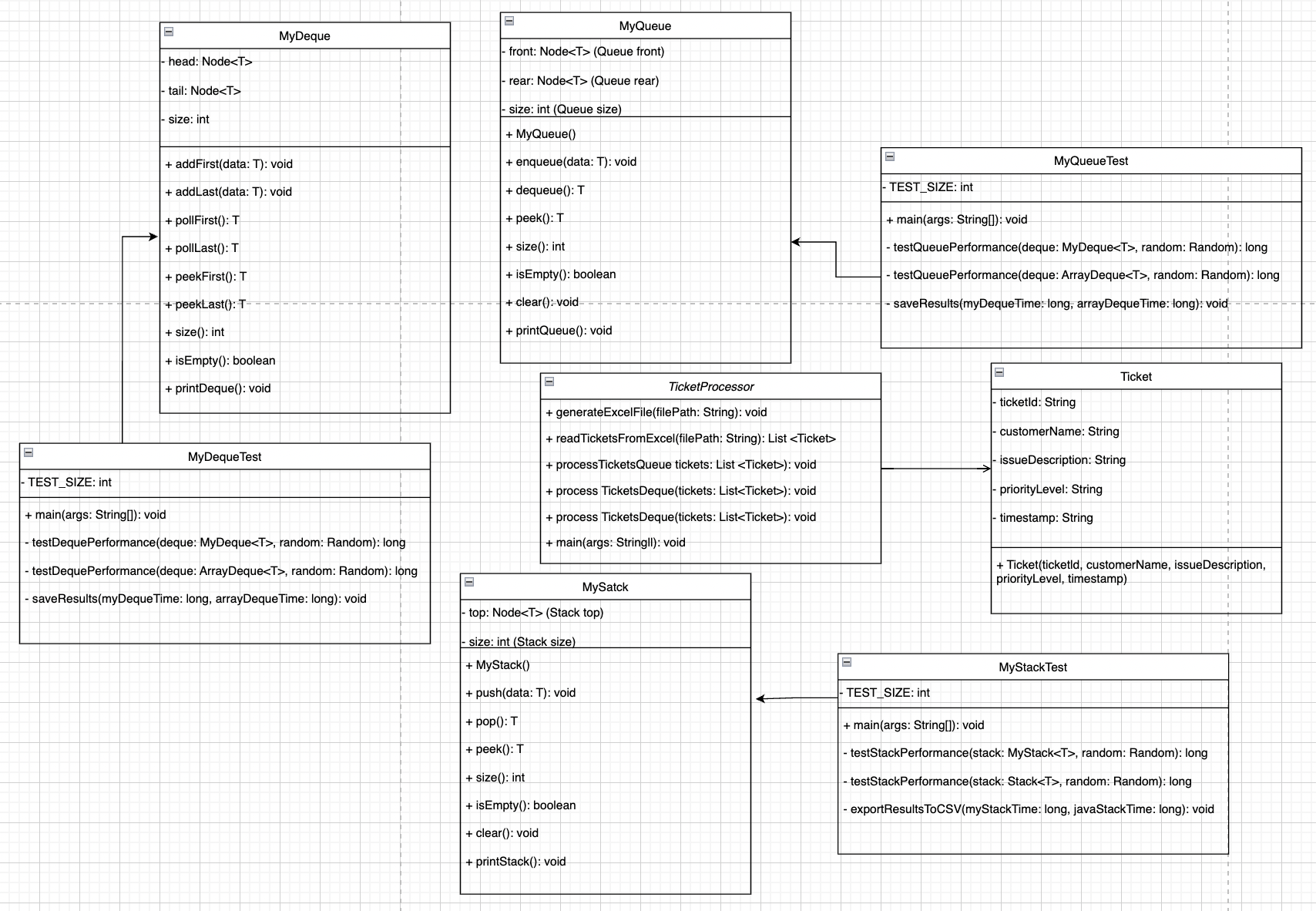
Operations:

push(): Inserts a ticket at the top (O(1))

pop(): Removes the most recent ticket (O(1))

Use Case: Mainly implemented for benchmarking, as LIFO is not suitable for ticketing systems.

1. **Analysis/Design (UML Diagram(s))**

****

1. **Code Implementation (please add your TA -** [Furqan.rustam1@gmail.com](mailto:Furqan.rustam1@gmail.com) **– as a collaborator)**

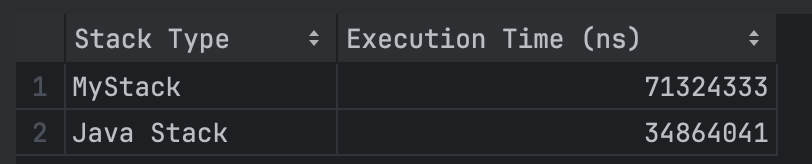
GitHub (link):https://github.com/WolfClarence/AdvancedDataStructure

1. Stack Implementation

MyStack is a custom implementation of a linked list-based stack, designed to support LIFO (Last-In-First-Out) operations efficiently. It maintains a top pointer, which allows for quick access to the most recently added element. Each node stores a reference to the next element, making operations like push(), pop(), and peek() efficient.

To evaluate performance, a stress test was conducted, where 1,000,000 random push and pop operations were performed on both MyStack and Java’s built-in Stack. The results showed that MyStack executed in 71,324,333 ns, while Java Stack completed in 34,864,041 ns. The built-in Stack was significantly faster due to internal optimizations and memory management, but both implementations achieved O(1) performance for all operations.

Result photo:

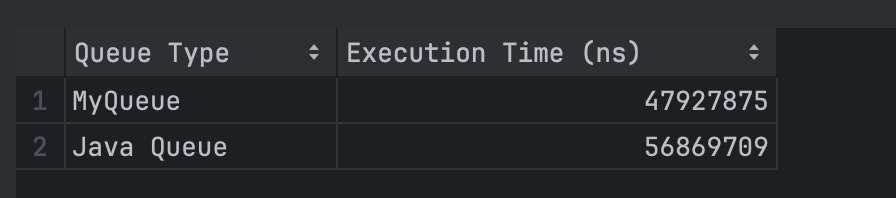


1. Queue Implementation

MyQueue is a linked list-based queue implementation, designed for FIFO (First-In- First-Out) processing. It maintains front and rear pointers, ensuring efficient insertion and removal. The enqueue() operation adds elements to the rear, while dequeue() removes elements from the front.

For performance evaluation, 1,000,000 enqueue and dequeue operations were tested on MyQueue and Java’s built-in Queue (LinkedList). The results showed that MyQueue executed in 47,927,875 ns, while Java’s Queue completed in 56,869,709 ns. The slight difference is due to Java’s optimized memory allocation, but both implementations maintain O(1) operations.

Result photo:

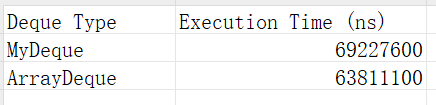


3) Deque Implementation

MyDeque is a custom implementation of a doubly linked list-based deque, designed to support efficient insertion and removal operations at both ends. It maintains head and tail pointers for quick access to the front and back, while each node has prev and next references to enable constant-time modifications. The size variable tracks the number of elements, ensuring that operations like isEmpty() and size() are efficient. Unlike ArrayDeque, which is backed by a resizable array, my implementation does not require resizing, making it more suitable for scenarios where memory reallocation could be a concern.

To evaluate performance, I conducted a stress test by performing 1,000,000 random operations on both MyDeque and Java’s built-in ArrayDeque. Each test involved a mix of addFirst, addLast, and pollFirst operations to simulate real-world usage. The results showed that MyDeque executed in 69,227,600 ns, while ArrayDeque completed in 63,811,100 ns. Although MyDeque performed well, ArrayDeque was slightly faster due to array-based memory locality, highlighting the trade-offs between linked list and array implementations.

Result photo:



1. **Video of the Implementation running**

Video link: https://www.youtube.com/watch?v=S2gMeLfYmU8

Comments:

This is a brief overview of the code execution and result display. The results are shown in the console and also saved in CSV files.