Name: Gülay Yamaner 240445701 Date: 09 08 2025

CENG 201: Data Structures and Algorithms - Homework 1 Analysis Report

Task 1 Report: Book List (Linked List Implementation) Big O Time Complexity Analysis addBook(Book book): The time complexity is O(1). By maintaining a tail pointer to the last node in the list, a new element can be added directly to the end in constant time without traversing the list. findBook(int id): The time complexity is O(n). In the worst-case scenario, the entire list of n nodes must be traversed to locate a book or to confirm it is not present. removeBook(int id): The time complexity is O(n). This operation is dominated by the initial search to find the book, which takes O(n) time. While the subsequent pointer adjustment for removal is an O(1) operation, the overall complexity remains linear due to the search phase.

Linked List vs. ArrayList Performance A Linked List is a suitable choice for this task, but an ArrayList offers different performance trade-offs. Addition: Adding an element to the end is O(1) for our Linked List with a tail pointer. For an ArrayList, this operation is typically O(1) in amortized time, but it becomes O(n) when the internal array's capacity is exceeded, requiring a resize and copy operation. Search and Removal: Searching by value (ID) is O(n) for both data structures, as each may require iterating through the elements. The key difference is in the removal step. After finding an element, removal from a Linked List is an efficient O(1) operation. In an ArrayList, removing an element from the middle or beginning necessitates shifting all subsequent elements, which is a costly O(n) operation. Suitability: A Linked List is well-suited for this task. Its strength lies in predictable O(1) insertion and efficient O(1) deletion once an element is located. An ArrayList would be superior if the primary operation was accessing books by a direct index (e.g., get(i)), which it performs in O(1) time. Given the requirements involve finding and removing by value, the performance is comparable in search, but the Linked List is more efficient at the deletion step itself.

Task 2 Report: Borrowing Queue (Queue Implementation) Suitability of a Queue A Queue is the ideal data structure for managing borrowing requests because it strictly enforces fairness through its First-In, First-Out (FIFO) principle. This ensures that users are served in the exact order their requests were received, which is the most logical and equitable way to manage a waiting list for a limited resource. Using a queue guarantees that the oldest request is processed first, preventing newer requests from unfairly "cutting in line."

Issues with Using a Stack Instead If a Stack were used, it would operate on a Last-In, First-Out (LIFO) basis. This would create a fundamentally unfair system where the most recent request is always the first to be processed. A user who has been waiting for an extended period would be continuously pushed further down the list as new requests arrive. This behavior is unacceptable for a borrowing system as it could lead to "starvation," where older requests are never fulfilled.

Big O Complexity Comparison When implemented with a linked list featuring both front and rear pointers: Queue Operations (enqueue, dequeue): Both are O(1). enqueue adds an element at the rear, and dequeue removes one from the front, both being simple pointer manipulations. Stack Operations (push, pop): Both are also O(1). push and pop operate on the top of the stack, which is also a constant-time operation. The decision between a Queue and a Stack is therefore not based on their Big O performance but on the logical model required by the problem. For managing user requests fairly, FIFO is the correct model, making the Queue the only appropriate choice.

Task 3 Report: Return Stack (Stack Implementation) Suitability of a Stack A Stack is a suitable data structure for managing return requests as it models a common real-world workflow: processing a pile of items. When books are returned, they are often placed into a cart or bin. The last book placed on the pile is the first one available to be picked up and re-shelved. This Last-In, First-Out (LIFO) model is efficient for library staff, as the most accessible item is processed first. This principle is fundamental in computing for managing processes like function calls (the call stack).

Issues with Using a Queue Instead If a Queue were used for returns, it would operate on a First-In, First-Out (FIFO) basis. This would mean the first book returned would be the first to be re-shelved.

While this is a valid and orderly processing method, it does not align with the LIFO behavior specified by the task to demonstrate an understanding of stacks. The primary "issue" is that it represents a different operational strategy than the "pile" analogy that the stack data structure perfectly models.

Big O Complexity Comparison When implemented with a linked list: Stack Operations (push, pop, peek): All have a time complexity of O(1) because they only interact with the top element. Queue Operations (enqueue, dequeue, peek): All have a time complexity of O(1), assuming an implementation with both front and rear pointers. The choice to use a Stack here is driven by the requirement to model a LIFO process, not by a performance advantage over a Queue, as their core operation complexities are identical.

Task 4 Detailed Report: Library System Integration Design Choice for Priority Borrowing To support priority borrowing, the chosen design utilizes two distinct queues: a priorityBorrowQueue and a normalBorrowQueue. This approach elegantly separates the two user groups while adhering to the assignment's constraints. When a borrow request is made, its isPriority flag determines which queue it enters. The processBorrowingRequest method contains the core logic. It first checks if the priority queue has any requests. If so, it dequeues and processes from there. Only when the priority queue is empty does the system begin processing requests from the normal queue. This design strictly ensures that all priority requests are handled before any normal ones. Crucially, since each queue maintains its own FIFO integrity, the order of requests is preserved within each group. This two-queue solution is simple, efficient, and avoids modifying the internal logic of a single queue, which could introduce complexity and bugs.

Big O Complexity of Priority Implementation addBorrowingRequest: The complexity is O(1). The method involves a boolean check followed by a standard enqueue operation, which is O(1). processBorrowingRequest: The complexity is O(1). The method performs a size() check and a dequeue() operation, both of which are constant-time operations for our linked list-based queue. This implementation is optimally efficient because its core operations do not depend on the number of elements in the queues.

Comparison to a Binary Heap-based Priority Queue A Binary Heap is a more traditional and flexible data structure for implementing a priority queue. Performance: A binary heap offers O(log n) complexity for insertion and deletion operations. This is highly efficient but marginally slower than our O(1) two-queue solution. Flexibility: The primary advantage of a binary heap is its ability to handle many different priority levels. If the system required "gold," "silver," and "bronze" priority tiers, a heap would manage this naturally. Our two-queue approach is hardcoded for exactly two tiers and would require significant modification (e.g., adding more queues and if-else logic) to support more levels. Implementation Complexity: Implementing a binary heap is more complex than using two standard queues. It involves managing an array and performing heapify operations to maintain the heap property after insertions or deletions.

Conclusion: For the problem as specified—with only two priority levels—our two-queue design is superior due to its simplicity and better O(1) performance. However, for a more general-purpose, scalable priority system, a Binary Heap would be the more robust and appropriate choice.