PROJECT FRONT SHEET

Unit number and title	**Unit 19: Data Structure & Algorithms**		
Submission date	12/11/2024		
Student Name	Nguyen Dinh Hoang Son	**Student ID**	GCS220616
Class	COS1104	**Assessor name**	Ngo Quoc Anh

❒ **Summative Feedback:** **Feedback:**		
Grade:	**Assessor Signature:**	**Date:**

Data Structures and Algorithms for Online Bookstore Systems

Nguyen Dinh Hoang Son

FPT Greenwich University

1649A - Data Structure and Algorithm

Mr. Ngo Quoc Anh

12/11/2024

Table Of Contents

1. Introduction

1.1. Overview of the Online Bookstore System

The Online Bookstore System represents a sophisticated order management platform designed to streamline the process of handling customer orders efficiently. In an increasingly digital world, the need for optimized e-commerce systems has become paramount, particularly in sectors such as retail. This system leverages core data structures and algorithms to manage the flow of orders from customers, ensuring both speed and accuracy in processing.

Upon placing an order, the customer's information, including their details and the list of books they wish to purchase, is stored in a queue data structure, which follows the First In, First Out (FIFO) principle (Kenton, 2024). This allows for fair and systematic order handling, as each new order joins the queue and is processed sequentially. For each order, the system verifies the availability of books, after which the items are sorted based on predefined criteria such as title, author name, or genre, This sorting functionality is achieved through efficient sorting algorithms like Merge Sort (GeeksforGeeks, 2018) or Quick Sort (GeeksforGeeks, 2014), ensuring that large volumes of books are organized swiftly and correctly.

Once processed, customers can track the status of their orders using search algorithms that provide quick access to specific orders through identifiers like order numbers or customer names. By employing search methods such as Linear (GeeksforGeeks, 2024) or Binary Search (GeeksforGeeks, 2019), the system ensures that order tracking remains responsive even as the volume of data grows.

The system not only improves operational efficiency but also highlights the importance of choosing the right data structures and algorithms in developing scalable and responsive e-commerce platforms. Through this project, the critical role of queues, stacks, sorting, and

searching algorithms is demonstrated showing how abstract data types (ADTs) and algorithmic techniques can enhance real-world applications in the ecommerce industry.

1.2. Importance of Data Structures and Algorithms in Order Processing

In the context of modern ecommerce platforms, efficient order processing is paramount to maintaining customer satisfaction and operational effectiveness. Data structures and algorithms are critical in design systems that can handle a high volume of transactions with speed, accuracy, and minimal resource consumption. For the Online Bookstore System, selecting the right data structures and algorithms is essential to ensure that orders are processed quickly and correctly, even as the system scales.

Data structures like queues and stacks serve as the backbone for managing the flow of customer orders. A queue, following the First In, First Out (FIFO) principle, ensures that orders are handled in a fair sequence, preventing older orders from being delayed by newer ones. This simple yet powerful structure guarantees that every customer's order is processed in the order it was received, which is crucial for maintaining a positive customer experience.

Moreover, algorithms play a vital role in sorting and searching within the order processing system. Sorting algorithms such as Merge Sort or Quick Sort enable the system to organize book orders efficiently, based on various criteria like book title or author name. The choice of sorting algorithms such as Merge Sort or Quick Sort enable the system to organize book orders efficiently, based on various criteria like book title or author name. The choice of sorting algorithm can significantly impact the system's performance, particularly when handling large volumes of data. A well-chosen sorting algorithm not only speeds up the process but also reduces the system's computational load.

Similarly, searching algorithms like Linear Search and Binary Search provide the functionality for customers to track their orders in real-time. The efficiency of these search algorithms determines how quickly the system can retrieve relevant information, ensuring a smooth user experience. For instance, Binary Search, with its logarithmic time complexity, can quickly locate specific orders even when the dataset grows exponentially.

In essence, the choice of data structures and algorithms directly affects the system's ability to scale, handle large datasets, and maintain fast response times. By implementing optimized data structures and algorithms, the Online Bookstore System can manage orders efficiently, ensure timely processing, and enhance overall customer satisfaction. Thus, mastering the use of these computational tools is indispensable for developing robust, scalable order processing systems.

1.3. Project Objectives and Scope

The primary objective of this project is to design and implement an efficient order processing system for an Online Bookstore by utilizing key data structures and algorithms. The system will demonstrate abstract data types (ADTs) such as stacks and queues, along with sorting and searching algorithms, can significantly enhance the operational efficiency of managing customer orders. Through this project, the goal is to showcase how the proper selection and implementation of these data structures and algorithms can streamline order handling, reduce processing time, and improve customer satisfaction.

Specifically, this project aims to:

1. **Design a Robust System:** Develop a design specification that clearly defines the use of data structures like queues (for managing order flow) and stacks (for handling completed orders), ensuring all operations are valid and efficient.

2. **Implement Sorting Algorithms:** Integrate appropriate sorting algorithms such as Merge Sort, Quick Sort, or Selection Sort to organize book orders by various criteria, ensuring that the sorting process is optimized for different data sizes and types.
3. **Incorporate Searching Algorithms:** Implement efficient search algorithms like Binary Search and Linear Search to allow customers to track their orders in real-time, ensuring that search operations remain fast even as the system scales.
4. **Evaluate Performance:** Critically assess the performance of the chosen data structures and algorithms, focusing on time and space complexity. The project will analyze the trade-offs involved in selecting specific algorithms and data structures for various tasks within the order processing system.

The scope of the project encompasses both the theoretical and practical aspects of data structures and algorithms. From designing abstract data types to implementing and evaluating sorting and searching algorithms, the project will cover all stages of system development. In addition, the project will focus on the algorithms used, assessing their real-world application in an ecommerce environment. The final deliverables will include a report detailing the design and implementation of the system, along with a critical evaluation of its efficiency.

Through this project, a comprehensive understanding of how data structures and algorithms can be applied in a real-world context will be demonstrated, providing insights into the trade-offs and benefits of using ADTs and algorithmic techniques in ecommerce order processing.

2. Abstract Data Types (ADTs) and Concrete Data Structures

2.1. Definition of Abstract Data Types (ADT)

According to www.javatpoint.com. (n.d.), an abstract data type is a high-level abstraction of a data structure that defines the required interface without specifying how it should be implemented or the programming language used. ADTs are characterized by the operations that can be performed on them, such as inserting, deleting, or retrieving data, and by the rules that dictate how these operations are carried out. What distinguishes an ADT from a concrete data structure is that it defines the "what" (i.e., what operations are available) rather than the "how" (i.e., how the operations are implemented) (ayushdey110, 2021).

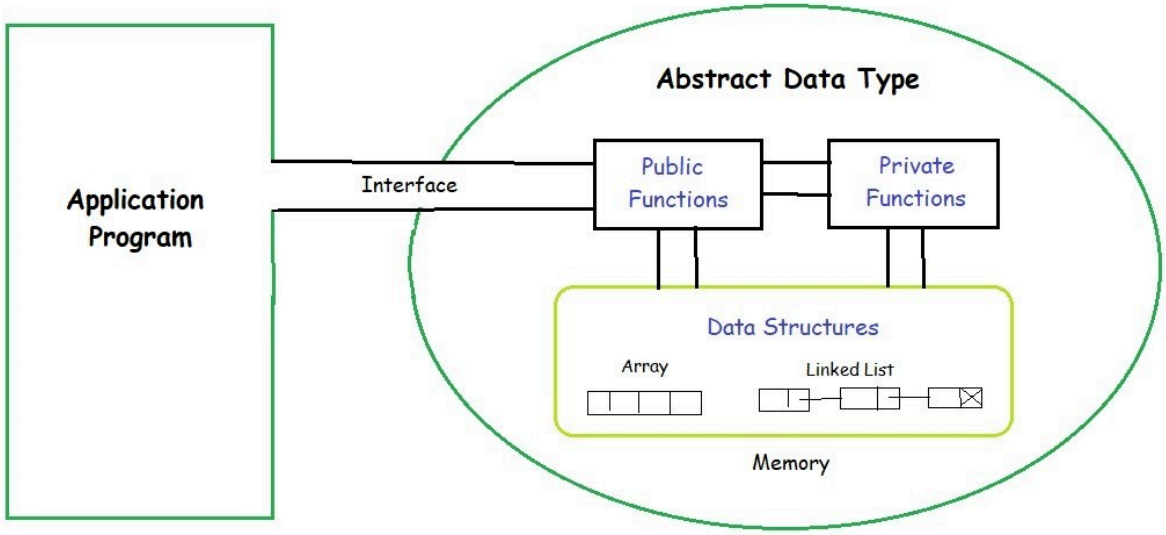

Figure 1. Interface and Structure of an Abstract Data Type (ADT) (GeeksforGeeks, 2017a)

As shown in Figure 1, adapted from GeeksforGeeks (2017a), it shows that an ADT provides an interface that hides the underlying implementation details from the user. This abstraction allows the user to interact with the data in a consistent way, regardless of how the data is organized internally. For example, a queue ADT allows elements to be inserted and removed in a First In,

First Out (FIFO) manner, but the specific method by which this is achieved (e.g., using an array or linked list) is abstracted away.

ADTs are essential in software development because they promote modularity and maintainability. By defining data structures abstractly, developers can focus on the logical behavior of the system rather than the intricacies of implementation. Common examples of ADTs include:

- **Stack**: A Last In, First Out (LIFO) structure where elements are added and removed from the same end (top).
- **Queue**: A First In, First Out (FIFO) structure where elements are added at the back and removed from the front.
- **List**: A collection of elements where each element can be accessed via an index.

In the context of the Online Bookstore System, the queue ADT is used to manage customer orders, ensuring they are processed in the order they are received. Similarly, a stack ADT may be utilized to handle completed orders. These ADTs define how the system interacts with the orders conceptually, leaving the specific data structure and algorithm implementations open for optimization based on performance requirements.

Thus, the use of ADTs in this project provides a clear and flexible foundation for developing efficient data structures tailored to the needs of the system.

2.2. Real-World Applications of ADTs

Applications of Stacks

Stacks are widely utilized in managing function calls and recursion in programming. They operate under the Last In, First Out (LIFO) principle, making them perfect for storing function return addresses and managing the state of recursive operations (aayushi2402, 2022).

Function Calls

Based on aayushi2402 (2022), stacks are essential for handling function calls within programs. When a function is called, the return address is pushed onto the stack, allowing the program to return to the correct location once the function finishes executing. This ensures that functions are executed in the correct sequence, and program control flow is maintained .

Recursion

Stacks are also crucial for recursive function calls. During recursion, each call pushes the current state, including local variables and the return address, onto the stack. As the recursive function completes and unwinds, the stack allows the program to return to the previous state, maintaining the correct flow of execution (aayushi2402, 2022).

Applications of Queues

Queues are integral to task management and resource allocation, operating on a First In, First Out (FIFO) principle, which ensures fair handling of processes and resources (GeeksForGeeks, 2011).

Task Scheduling

Queues are frequently employed in task scheduling systems, where processes are handled in the order they are received. In operating systems, for instance, tasks are queued for CPU execution

based on their priority or arrival order. This ensures that each task is executed in a timely and fair manner, improving overall system efficiency (GeeksForGeeks, 2011).

Resource Allocation

According to GeeksForGeeks (2011), queues are also used for managing and allocating limited resources, such as printers or CPU time, in multi-user systems. Each request is enqueued and processed sequentially, ensuring that resources are distributed fairly and efficiently, reducing conflicts and delays.

2.3. Overview of Concrete Data Structures

Following ayushdey110 (2021), a concrete data type contrasts with an abstract data type, as it is a specific, solution-focused data type that represents a clearly defined concept within a single domain. While ADTs define the behavior and operations available, concrete data structures determine how these operations are executed, directly influencing performance, memory usage, and the practicality of certain operations.

Concrete data structures are essential because they transform abstract concepts into usable formats that software systems can manipulate. In the Online Bookstore System, concrete data structures such as arrays, linked lists, stacks, and queues enable efficient management of customer orders, inventory, and order processing.

Array

An array is a linear data structure that stores elements of the same type in a contiguous block of memory, allowing for direct access to any element by its index. Think of an array like a row of lockers, where each locker (or index) holds a specific item and can be quickly located by counting from the start (GeeksforGeeks, 2017b).

Referring to GeeksforGeeks (2017b), arrays are generally fixed in size, meaning once they're created, they can't be expanded or shrunk. However, many programming languages offer flexible versions of arrays, such as lists in Python or vectors in C++, which manage resizing internally.

In the Online Bookstore System, arrays are ideal for storing fixed inventories or predefined categories of books, allowing quick access and efficient retrieval when updates are infrequent.

Linked List

Following harendrakumar123 (2024b) a linked list is a dynamic data structure where elements, or nodes, are stored non-contiguously. Each node links to the next, enabling efficient memory use and easy insertion or deletion without shifting elements, as required in arrays. This structure allocates memory one element at a time, making it ideal for situations where the size of data changes frequently.

Unlike arrays, which store elements in a single contiguous block, linked lists grow by adding nodes individually, allowing flexibility. This sequential setup means elements are accessed one after another rather than directly by index, which is typical in arrays. Linked lists are also versatile, serving as the foundation for data structures like stacks, queues, and deques (harendrakumar123, 2024b).

In the Online Bookstore System, linked lists are suited to manage varying numbers of customer orders, supporting quick adjustments as orders are added or completed.

Tree

A tree is a hierarchical data structure composed of nodes connected by edges, where each node can have child nodes but only one parent node (except for the root). This structure is particularly useful for representing data with a clear hierarchy or relationships, such as organizational structures, file systems, or product categories. Trees provide efficient methods for searching,

inserting, and deleting elements, especially in cases where data is organized logically by levels or branches (GeekforGeeks, 2021).

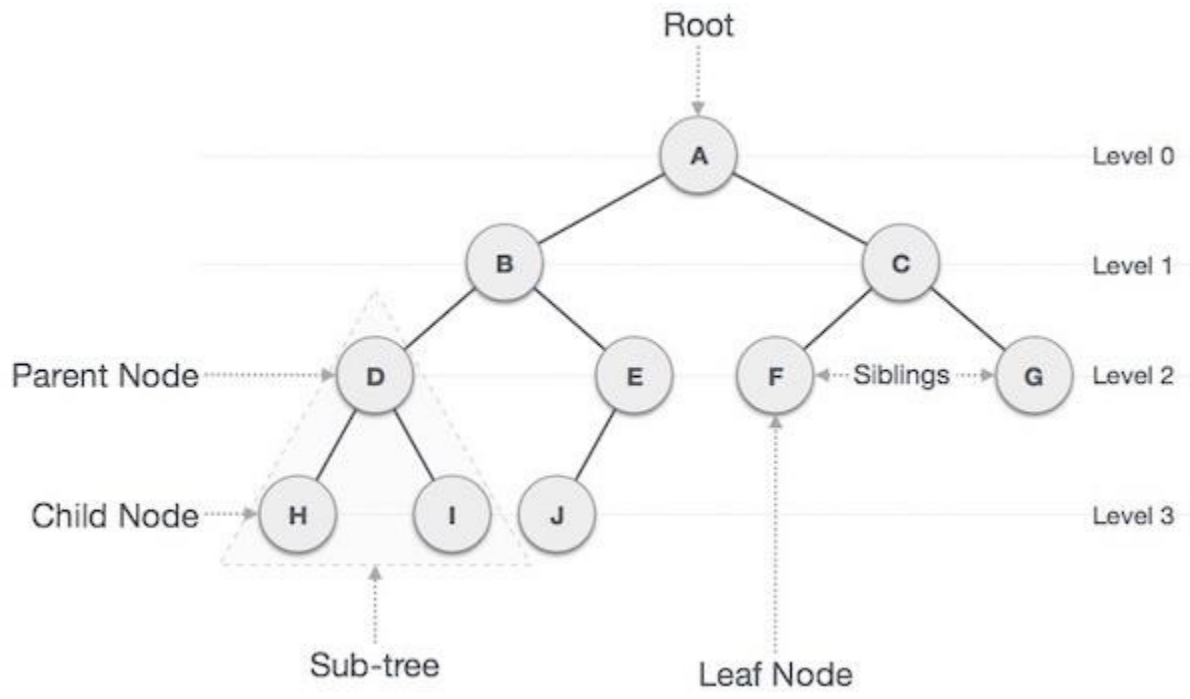

Figure 2. Tree Data Structure (tutorialspoint.com, 2019b)

Each tree has a root node, which serves as the entry point, and branches out into child nodes, forming sub-trees. Common types of trees include binary trees, where each node has at most two children, and binary search trees, which enable fast lookup by arranging nodes in a sorted order. Trees are widely used in applications that require hierarchical data management and quick retrieval of information, such as databases and file directory structures (tutorialspoint.com, 2019b).

In the Online Bookstore System, trees can effectively manage categories and subcategories of books. By organizing books into a tree structure based on genres, authors, or series, customers can easily browse and navigate through the selection, improving the user experience and search efficiency.

Comparison and Significance

Concrete data structures are crucial because they provide the foundation for implementing ADTs in a way that optimizes for specific system needs. For example, arrays offer fast access by index, but linked lists provide more flexibility for dynamic data. Stacks and queues, as implementations of ADTs, bring structure to how data is managed in operations like order processing and undo functionality.

In summary, choosing the right concrete data structure can significantly affect the efficiency and functionality of a system. By utilizing concrete data structures effectively, the Online Bookstore System can handle customer orders, manage inventory, and ensure seamless order processing, ultimately improving customer satisfaction and system performance.

2.4. Operations on ADTs (Push, Pop, Enqueue, Dequeue)

In Abstract Data Types (ADTs) like stacks and queues, key operations define how elements are added, accessed, or removed, forming the foundation for efficient data handling in software applications. The operations Push and Pop are central to stacks, while Enqueue and Dequeue are core to queues. Here's a closer look at these operations and their role in data management:

Push Operation (Stack)

The Push operation adds an element to the top of a stack, following the Last In, First Out (LIFO) principle. When an item is pushed onto the stack, it becomes the most accessible item, ensuring the last added element is the first one retrieved. This is useful in functions that require backtracking, such as managing recent operations or navigating recursive functions (GeeksforGeeks, 2015b).

Pop Operation (Stack)

The Pop operation removes the element at the top of the stack. Since stacks operate on LIFO, the last element added is the first one removed. This feature makes stacks valuable in reversing actions, such as the “undo” function in text editors or reversing recent navigation steps (tutorialspoint, 2020).

Enqueue Operation (Queue)

Enqueue adds an element to the rear of a queue, aligning with the First In, First Out (FIFO) principle. This operation ensures elements are processed in the order they were added, making queues ideal for managing tasks in systems where order matters, such as customer service requests or scheduling tasks in an operating system (GeeksforGeeks, 2015a).

Dequeue Operation (Queue)

Dequeue removes the front element of a queue, following FIFO. This operation is critical in scenarios where data must be accessed in a strict sequence. For example, task scheduling, where each task is processed based on its arrival time, uses dequeue operations to ensure fairness and efficiency (www.programiz.com, n.d.).

Significance of ADT Operations in the Online Bookstore System

In the Online Bookstore System, these ADT operations enable efficient handling of data. Push and Pop allow the system to track recent actions, while Enqueue and Dequeue manage incoming customer orders in sequence, ensuring timely processing and improving customer satisfaction.

3. Design Specifications for Data Structure

3.1. Design of Queue for Customer Orders

In the Online Bookstore System, a queue is essential for managing customer orders efficiently and fairly. By following the First In, First Out (FIFO) principle, a queue ensures that orders are processed in the sequence they are received, maintaining an organized flow of operations. This is crucial in e-commerce settings, as it promotes a reliable experience for customers by preventing newer orders from jumping ahead of previously received ones.

Queue Structure

The queue for customer orders can be implemented as a dynamic structure using linked lists or arrays. In a linked list-based queue, each order is represented as a node containing order details (such as customer name, address, and list of books), with pointers connecting each node in sequence. This design is flexible, as it allows the queue to grow or shrink dynamically with the number of customer orders, without reallocating memory (GeekforGeeks, 2014).

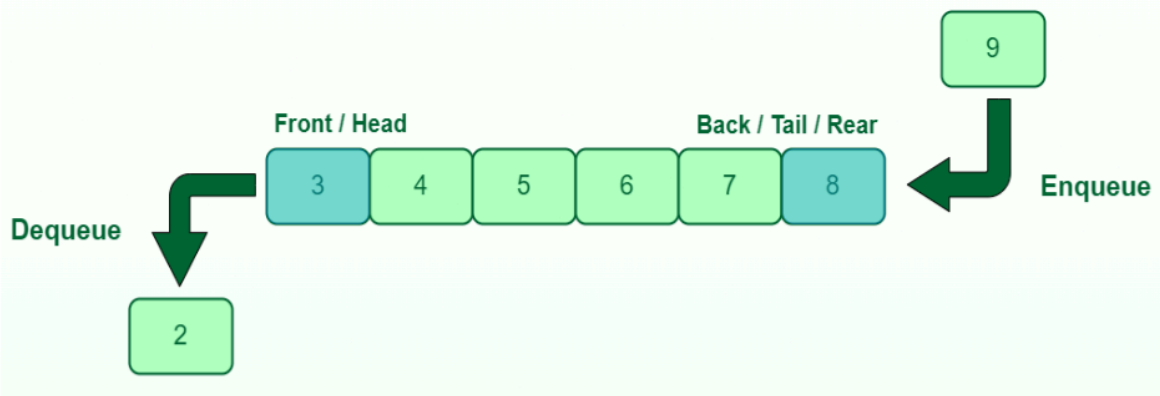

Figure 3. Queue Data Structure (code_r, 2023)

Alternatively, an array-based queue can also be used, where each index represents an order in sequence. While arrays provide faster access to specific indices, they are less flexible for a growing number of orders, as they require fixed memory allocation. To work around this

limitation, circular arrays can be implemented to optimize memory usage by wrapping around to the beginning of the array when reaching the end (www.programiz.com, n.d.).

Queue Operations for Order Management

a. Enqueue:

   When a customer places an order, it is added to the end of the queue (the rear), ensuring that the order sequence is preserved. In an array-based implementation, the rear index is incremented with each new order, wrapping back to the start if using a circular array. Linked list-based queues simply add a new node at the end, maintaining dynamic flexibility.

b. Dequeue:

   As the system processes orders, the front element of the queue is removed, ensuring that each order is handled in the order it was received. This operation is efficient in a linked list, as it simply removes the front node and adjusts the pointer to the next node. In an array, the front index advances or wraps around to the beginning in a circular structure, ensuring minimal memory waste.

Chosen Implementation for Queue

For the Online Bookstore System, a linked list-based queue is preferred for managing customer orders due to its dynamic sizing. Unlike arrays, which are fixed in size, a linked list-based queue can adjust to handle variable order volumes without reallocating memory. This flexibility ensures the system scales with demand, enhancing resource efficiency and order handling.

Key Benefits:

a. Dynamic memory allocation for efficient scaling.
b. Seamless enqueue and dequeue operations without memory shifts or resizing.

(www.programiz.com, n.d.)

3.2. Design of Stack for Completed Orders

In the Online Bookstore System, a stack is well-suited for handling completed orders, as it allows the system to track recent order transactions with ease. The stack's Last In, First Out (LIFO) principle ensures that the most recently completed orders are accessible first, which is helpful for reversing recent actions or quickly reviewing recent transactions.

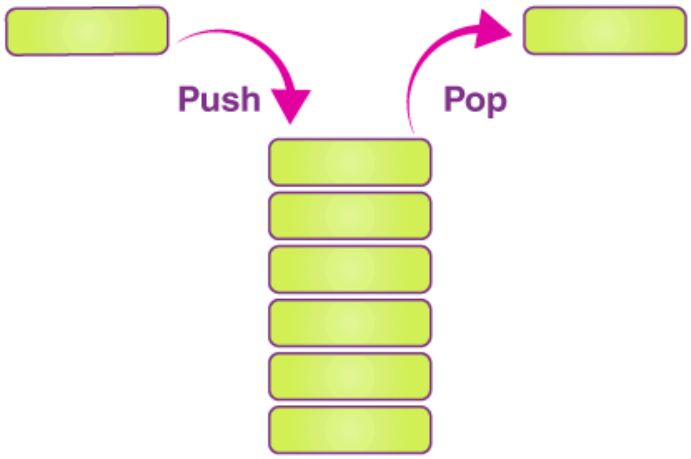

Figure 4. Stack Data Structure (BYJUS, n.d.)

This design choice is practical for systems that need quick access to recent data while preserving past data below the stack.

Stack Structure

The stack for completed orders can be implemented using either an array or a linked list.

- In an array-based stack, completed orders are added to the top, with a pointer (or index) marking the current top position. This implementation provides fast access to recent orders but requires a fixed memory size unless implemented as a dynamic array, which can automatically resize when reaching its limit (GeeksforGeeks, 2015b).

- A linked list-based stack is more flexible as it does not require a fixed size. Each completed order is stored as a node, where each node points to the next. When a new order is completed, it becomes the new "top" of the stack, and a reference is updated to point to this latest node. This approach allows the stack to grow and shrink dynamically without the need to reallocate memory (tutorialspoint, 2020).

Stack Operations for Completed Orders

a. Push:

   When an order is completed, it is pushed onto the top of the stack. In an array-based implementation, the top index is incremented, and the order is stored at this position. In a linked list-based stack, a new node is created and placed at the top, linking it to the previous node. This operation is efficient, requiring only a few adjustments to the top index or pointer.

b. Pop:

   If the system needs to access the most recent completed order or reverse an action, the pop operation removes the order at the top. For an array-based stack, the top index is simply decremented. In a linked list-based stack, the top node is removed, and the reference updates to the next node in line. This ensures that the system can quickly retrieve and manage the latest data without disrupting the rest of the stack.

Chosen Implementation for Stack

A linked list-based stack is ideal for managing completed orders due to its flexible size. This implementation dynamically grows as new orders are completed, eliminating the need for preallocated memory and avoiding the fixed size limitations of arrays. This choice supports efficient memory management and scales to accommodate varying order volumes.

Key Benefits:

a. Dynamically adjusts with the number of completed orders.

b. Efficient use of memory without resizing constraints.

(GeeksforGeeks, 2015b)

3.3. Valid Operations for Data Structures

In data structures like stacks, queues, arrays, and linked lists, specific operations define how data can be added, accessed, modified, or removed. These operations are essential for managing data effectively in any system, including the Online Bookstore System, where various data structures handle tasks like order processing, inventory management, and completed order tracking. Below is an overview of the key operations for each data structure and their roles in data management.

Stack Operations

According to GeeksforGeeks (2015b), stacks operate on the Last In, First Out (LIFO) principle, with operations focused on adding or removing elements from the top:

- Push: Adds an element to the top of the stack, making it the most accessible item.
- Pop: Removes and returns the element at the top of the stack.
- Peek: Returns the top element without removing it, allowing inspection of the latest item.
- IsEmpty: Checks if the stack is empty, ensuring operations like pop are safe to perform.

Queue Operations

As claimed by www.programiz.com (n.d.), queues follow the First In, First Out (FIFO) principle, where elements are processed in the order they arrive. Queue operations are designed to handle items at the front and rear:

- Enqueue: Adds an element to the rear of the queue, preserving the order of arrival.
- Dequeue: Removes and returns the element at the front of the queue.

- Peek: Returns the front element without removing it, providing a preview of the next item to be processed.
- IsEmpty: Checks if the queue is empty, preventing errors when dequeuing.

Array Operations

Since 2024(a), harendrakumar123 has stated that arrays provide direct access to elements via indices, which makes them efficient for retrieving data. However, insertion and deletion can be more complex if the array is fixed in size:

- Access by Index: Retrieves an element directly by its position, allowing fast data access.
- Insert: Adds an element at a specified position, though this may require shifting elements if the array is already full.
- Delete: Removes an element by index, shifting subsequent elements to fill the gap if needed.
- Length: Returns the size of the array, helpful for loop control and data limits.

Linked List Operations

Linked lists are flexible, dynamic structures where nodes are linked in sequence. They are well-suited for applications requiring frequent insertion or deletion:

- Insert: Adds a new node at the beginning, end, or any specified position, adjusting links accordingly.
- Delete: Removes a node from the list by updating the pointers of the surrounding nodes.
- Search: Traverses the list to find a specific element, since direct access by index is not available.
- Length: Returns the count of nodes in the list, reflecting the current size dynamically.

(www.tutorialspoint.com, n.d.)

Significance in the Online Bookstore System

In the Online Bookstore System, these operations allow for efficient management of customer orders and completed transactions. Push and Pop enable fast handling of recent orders in stacks, while Enqueue and Dequeue maintain order sequence in queues. Array operations facilitate quick access to inventory items, while linked list operations provide flexibility in handling dynamic order volumes.

3.4. Relationship Between ADTs and Concrete Implementations

Abstract Data Types (ADTs) define the logical behavior and operations of a data structure, while concrete implementations specify the actual storage and manipulation of data in memory. ADTs describe what operations can be performed without detailing how these operations are executed, providing a high-level blueprint for data handling. Concrete implementations, on the other hand, define the specific structures, algorithms, and memory allocations that make these abstract concepts functional.

Abstract Data Types (ADTs)

ADTs focus on defining operations that users can perform, such as adding, removing, or accessing elements. For example, the Stack ADT defines push and pop operations, ensuring a Last In, First Out (LIFO) behavior, while the Queue ADT defines enqueue and dequeue operations to maintain a First In, First Out (FIFO) sequence. ADTs allow developers to work with data structures conceptually, without needing to know the underlying mechanisms.

Concrete Implementations

According to ayushdey110 (2021), concrete implementations bring ADTs to life by specifying how data is organized and manipulated in memory. Each ADT can have multiple implementations, each with distinct performance characteristics. For instance:

- Stack ADT can be implemented using an array or linked list, where an array-based stack provides faster access at the expense of a fixed size, while a linked list-based stack offers dynamic sizing but with additional memory for pointers.
- Queue ADT can also be implemented as an array, a linked list, or a circular buffer. Each option has trade-offs between memory efficiency, resizing capabilities, and speed of access.

The Role of ADTs and Implementations in the Online Bookstore System

In the Online Bookstore System, ADTs such as stacks and queues are essential for defining data handling operations. For example, the Queue ADT allows customer orders to be managed in a FIFO manner, regardless of whether the queue is implemented as an array or linked list. This separation allows developers to choose or switch between implementations based on performance needs, memory constraints, or system requirements, without altering the high-level order processing logic.

The relationship between ADTs and their concrete implementations provides flexibility and adaptability, enabling the system to maintain consistent functionality while optimizing resource use. By abstracting operations with ADTs, the Online Bookstore System can efficiently manage orders, inventory, and completed transactions through well-suited data structures.

4. Sorting Algorithm in Order Processing

4.1. Overview of Sorting Algorithms

Sorting is a fundamental operation in data processing, organizing data into a specified order, such as ascending or descending. In an order processing system like the Online Bookstore System, sorting algorithms help organize book orders by various criteria (e.g., book title, author name, or price), making data easier to retrieve, search, and display. Different sorting algorithms have unique characteristics and performance profiles, which determine their suitability based on the size of data and processing requirements.

Here's an overview of four commonly used sorting algorithms and how they apply to order processing:

a. Insertion Sort

Insertion Sort is a simple, intuitive sorting algorithm that builds a sorted section one element at a time. Each new element is compared with the already sorted elements, moving it to its correct position. This algorithm works well with small datasets or nearly sorted data, as it can efficiently sort with minimal shifts.

- Time Complexity: $O(n^2)$ in the worst case, but performs well on nearly sorted data.
- Use Case: In order processing, Insertion Sort is useful for organizing smaller batches of orders or already partially sorted lists where only minor adjustments are needed.

(programiz, 2020)

b. Merge Sort\

According to the research (GeeksforGeeks, 2018), Merge Sort is a divide-and-conquer algorithm that recursively divides the data into halves until each half contains only one element. It then

merges these halves back together in sorted order. Merge Sort is highly efficient for large datasets and ensures stable sorting, meaning it preserves the order of equal elements.

- Time Complexity: $O(nlog(n))$ for all cases, making it effective for large datasets.
- Use Case: Merge Sort is ideal for the Online Bookstore System when sorting large volumes of orders, as it handles data size efficiently and guarantees stable sorting.

c. Quick Sort

Quick Sort is also a divide-and-conquer algorithm, choosing a "pivot" element and partitioning the data around this pivot. Elements smaller than the pivot move to the left, while larger elements move to the right. Quick Sort is highly efficient for large, randomly ordered data but can be less effective with highly ordered or nearly sorted data due to its dependence on the pivot selection.

- Time Complexity: $O(nlog(n))$ on average, but $O(n^2)$ in the worst case if the pivot is poorly chosen.
- Use Case: In order processing, Quick Sort works well for sorting unsorted or random data quickly. With proper pivot selection, it's a powerful tool for handling mid-to-large datasets.

(GeeksforGeeks, 2014)

d. Selection Sort

Selection Sort works by repeatedly finding the minimum (or maximum) element from the unsorted portion and placing it in its correct position within the sorted portion. It's straightforward but less efficient than other algorithms, especially for large datasets, due to its $O(n^2)$ complexity.

- Time Complexity: $O(n^2)$ for all cases, making it slower than Merge Sort or Quick Sort.

- Use Case: Selection Sort is suitable for small datasets within the Online Bookstore System, where simplicity is more important than speed. It's useful when sorting only a few orders or limited data that requires minimal overhead.

(tutorialspoint.com, 2019a)

4.2. Choosing the Appropriate Sorting Algorithm

In the Online Bookstore System, selecting an effective sorting algorithm is essential to ensure efficient data processing, especially when handling a large number of orders or inventory items. Each sorting algorithm has unique strengths, making it suitable for particular data types and sizes. Below is an analysis of the primary sorting algorithms and recommendations on when to use each:

Insertion Sort

Insertion Sort is a straightforward algorithm ideal for sorting small datasets or lists that are nearly sorted. It builds a sorted section incrementally, inserting each new element into its correct position. Its $O(n^2)$ time complexity limits its efficiency for large datasets, but it performs well with small or partially ordered lists.

- Best for: Small datasets or nearly sorted data, such as inserting new customer orders into an already sorted list.
- Not ideal for: Large, unordered datasets, as it becomes inefficient with increased data volume.

(programiz, 2020)

Merge Sort

According to GeeksforGeeks (2018), Merge Sort is a stable, divide-and-conquer sorting algorithm with a consistent $O(nlog(n))$ time complexity across cases, making it highly efficient

for larger datasets. It recursively divides data into halves, sorts each half, and merges them back together in order. This stability makes Merge Sort suitable for sorting data where order consistency is crucial.

- Best for: Large datasets that require stability, such as sorting orders by customer names or order dates.
- Not ideal for: Applications where memory usage is limited, as Merge Sort requires extra space for merging.

Quick Sort

GeeksforGeeks argues that Quick Sort is another divide-and-conquer algorithm known for its average $O(nlog(n))$ performance and speed with randomly ordered data. Quick Sort partitions data based on a pivot element, sorting elements around this pivot. While highly efficient on average, its $O(n^2)$ worst-case complexity makes it less predictable. Proper pivot selection, however, can optimize its performance in most cases (2014).

- Best for: Large, unsorted datasets where speed is crucial, such as quickly sorting book inventories or prices.
- Not ideal for: Small or nearly sorted datasets, where poor pivot selection can result in slower performance.

Selection Sort

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum (or maximum) element from the unsorted portion and placing it in its correct position. Its $O(n^2)$ time complexity makes it inefficient for larger datasets, but its simplicity and minimal overhead can be beneficial for very small lists.

- Best for: Small datasets where simplicity is more important than speed, such as sorting a few promotions or book categories.
- Not ideal for: Large datasets, as it lacks the efficiency of other algorithms like Merge Sort or Quick Sort.

(tutorialspoint.com, 2019a)

Conclusion

For the Online Bookstore System:

- Merge Sort is recommended for sorting large, stable datasets, such as customer orders or book inventories, where order consistency is important.
- Quick Sort is ideal for large, unsorted datasets that require fast sorting, provided proper pivot selection is implemented.
- Insertion Sort can be used for smaller datasets or those already partially sorted, as it avoids unnecessary complexity.
- Selection Sort may be useful for very small lists where simple sorting is required.

Choosing the right sorting algorithm based on data size and requirements improves the system's responsiveness and ensures efficient processing for order and inventory management.

4.3. Sorting Criteria for Book Orders

In the Online Bookstore System, establishing clear criteria for sorting book orders enhances both efficiency and user experience. Sorting allows customers to browse books by various attributes, making it easier for them to find specific items based on their preferences. Depending on the needs of the bookstore system, common sorting criteria for book orders include title, author, price, and publication date. Each of these criteria has unique sorting requirements that can influence the choice of sorting algorithm.

Data Structures and Algorithms for Online Bookstore Systems

Title

Sorting by title is essential for helping customers locate specific books alphabetically. This type of sorting requires a lexicographic (alphabetical) sort, where books are ordered from A-Z or Z-A. Merge Sort or Quick Sort can be effective here, as they handle large datasets efficiently and can accommodate string-based sorting criteria. Additionally, since books with identical titles may exist, a stable sorting algorithm like Merge Sort is beneficial, preserving the original order of entries with identical titles.

Author

Sorting by author enables users to browse books by specific writers or discover works grouped by the same author. This sorting is similar to title sorting, as it also requires lexicographic ordering. Like sorting by title, stable algorithms such as Merge Sort ensure that if multiple books by the same author are present, they remain in the original input order, enhancing the logical grouping of books under each author.

Price

Sorting by price helps users quickly locate books within their budget or identify the most affordable or premium options. Price-based sorting is numerical, where books are arranged in ascending or descending order based on their cost. For handling large inventories of books by price, Quick Sort is often suitable due to its average-case efficiency. However, if price stability (preserving order for books with identical prices) is required, Merge Sort can be an alternative.

Summary

For the Online Bookstore System:

- Title and Author sorting benefit from stable, lexicographic sorting algorithms like Merge Sort.

- Price and Publication Date sorting, where stability is less critical, can use Quick Sort for efficiency, or Merge Sort if stability is desired.

Choosing the right sorting criteria and algorithm based on the data attributes ensures that the system efficiently meets user needs, facilitating seamless browsing and enhancing the overall customer experience.

4.4. Time and Space Complexity of Sorting Algorithms

In the Online Bookstore System, understanding the time and space complexity of sorting algorithms is crucial for making efficient sorting choices, especially as the dataset grows. Time complexity measures how the sorting time increases with the input size, while space complexity considers the additional memory required. According to (GeeksforGeeks, 2023), here's an overview of the time and space complexity for commonly used sorting algorithms:

Insertion Sort

- Time Complexity:
    - Best Case: $O(n)$ – Insertion Sort performs well on nearly sorted data since it only checks and moves elements as necessary.
    - Average and Worst Case: $O(n^2)$ – In a completely unsorted list, each new element could require shifting many elements to its correct position.
- Space Complexity: $O(1)$ – Insertion Sort is an in-place algorithm, meaning it requires no extra memory beyond the input array.
- Use Case: Efficient for small datasets or lists that are nearly sorted.

Merge Sort

- Time Complexity:

    - Best, Average, and Worst Case: $O(nlog(n))$ – Merge Sort consistently divides data into halves, giving it a stable $O(nlog(n))$ performance for all cases.
- Space Complexity: $O(n)$ – Merge Sort requires additional space to store the divided subarrays during the merge process.
- Use Case: Suitable for large datasets or when stability is needed, such as sorting book titles and authors.

Quick Sort

- Time Complexity:
    - Best and Average Case: $O(nlog(n))$ – With a well-chosen pivot, Quick Sort efficiently divides data into partitions, allowing for fast sorting.
    - Worst Case: $O(n^2)$ – Quick Sort's worst case occurs if the pivot is poorly chosen, such as in an already sorted or reverse-sorted list.
- Space Complexity: $O(n)$ – Quick Sort is an in-place algorithm, but it requires additional space on the call stack for recursive partitioning.
- Use Case: Ideal for large, unsorted datasets where speed is crucial, provided pivot selection is optimized

Selection Sort

- Time Complexity:
    - Best, Average, and Worst Case: $O(n^2)$ – Selection Sort repeatedly finds the minimum (or maximum) element for each position, resulting in $O(n^2)$ performance across cases.
- Space Complexity: $O(1)$ – Selection Sort is in-place, requiring no additional space beyond the input list.

- Use Case: Suitable for small datasets where simplicity is more important than speed.

Summary Table

Algorithm	Time Complexity			Space Complexity
	Best	**Average**	**Worst**	**Worst**
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$
Quick Sort	$O(nlog(n))$	$O(nlog(n))$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(11)$

Table 1. Time and Space Complexity of Common Sorting Algorithms

4.5. Chosen Implementation for Sorting Algorithm

In the Online Bookstore System, sorting is essential for efficiently organizing book orders, enhancing data management, and providing a better user experience. After evaluating the performance characteristics of various sorting algorithms, Merge Sort has been selected as the sole sorting method due to its stability, consistent time complexity, and suitability for handling large datasets with diverse sorting criteria, such as titles, authors, prices, and publication dates. Merge Sort is chosen for several key reasons:

a. Consistency: With a time complexity of $O(nlog(n))$ across best, average, and worst cases, Merge Sort offers predictable performance, which is essential for large datasets where sorting speed and reliability are critical.

b. Stability: Merge Sort is a stable algorithm, preserving the order of records with identical keys. This is particularly beneficial for sorting by titles and authors, where the order of books by the same author or similar titles remains logically grouped, enhancing the browsing experience.

c. Scalability: The divide-and-conquer approach of Merge Sort allows it to efficiently handle large datasets, making it suitable for a high-volume system like an online bookstore.

By implementing Merge Sort for all sorting needs, the Online Bookstore System can ensure that book orders are organized consistently, whether sorting by title, author, price, or publication date. This uniform approach simplifies the system's sorting logic while maintaining high performance across different sorting requirements, ultimately supporting smoother order management and improved user satisfaction.

5. Searching Algorithms for Tracking Orders

5.1. Overview of Searching Algorithms

In any system that involves managing a large volume of data, searching algorithms are essential for quickly locating specific items. In the Online Bookstore System, searching algorithms allow the system to efficiently track and retrieve customer orders based on attributes like order ID, customer name, or order date. Choosing the right algorithm depends on the size and organization of the data, as well as the performance requirements. The two primary searching algorithms for order tracking are Linear Search and Binary Search.

Linear Search

Linear Search is a straightforward algorithm that checks each element in the dataset sequentially until the desired item is found or the end of the list is reached. It is simple to implement and works well for unsorted datasets, as it does not require any specific order.

- Time Complexity: $O(n)$ for all cases, as each element might need to be checked before finding the target.
- Space Complexity: $O(1)$, as it requires no additional space beyond the input list.
- Use Case: Linear Search is useful in the Online Bookstore System for small or unsorted datasets where order tracking might involve a quick scan of recent orders or customer names without a predefined order.

(tarunsarawgi_gfg, 2024)

Binary Search

In accordance with (GeeksforGeeks, 2019), Binary Search is a more efficient algorithm that requires the dataset to be sorted. It works by dividing the dataset in half, checking the middle element, and discarding half of the remaining data with each iteration based on whether the

target is greater or smaller than the middle value. This "divide and conquer" approach significantly reduces the number of comparisons.

- Time Complexity: $O(log(n))$ for all cases, as it repeatedly halves the dataset, making it much faster than Linear Search for large sorted lists.
- Space Complexity: $O(1)$, as it requires only a few extra variables to manage indices.
- Use Case: Binary Search is highly effective in the Online Bookstore System for sorted datasets, such as locating a specific order by ID or tracking orders by date. This method is preferred when the dataset is large and organized.

Algorithm	**Time Complexity**			**Space Complexity**
	Best	**Average**	**Worst**	**Worst**
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(log(n))$	$O(log(n))$	$O(1)$

Table 2. Time and Space Complexity of Searching Algorithms

Choosing Between Linear and Binary Search

For the Online Bookstore System, the choice between Linear Search and Binary Search depends on the organization and size of the data:

- Linear Search is preferred when dealing with small, unsorted data or when the dataset is dynamic and unsorted.
- Binary Search is optimal when the data is large and sorted, such as in cases where order IDs or dates are stored in a sorted sequence.

Both algorithms offer unique advantages, and selecting the appropriate one based on the dataset's characteristics ensures efficient order tracking and improves the system's overall performance.

5.2. Implementation of Searching for Order Tracking

In the Online Bookstore System, efficient tracking and retrieval of customer orders are essential for managing the order process smoothly. Two primary searching algorithms, Linear Search and Binary Search, are implemented based on the specific data structure and organization of the orders. Each algorithm has different strengths, and their implementation is optimized for different scenarios in the system.

Linear Search Implementation

Linear Search is a simple algorithm that sequentially checks each order in the list until it finds the target order. This method is effective when:

- The dataset is relatively small.
- The orders are stored in an unsorted or dynamically changing list.

In the Online Bookstore System, Linear Search can be used to search through unsorted recent orders or to quickly scan lists where order sorting is not necessary. For example, if the system has to find a customer order by name within a small batch of recent orders, Linear Search is suitable due to its simplicity.

```
function linearSearch(orders, targetOrder):
    for each order in orders:
        if order == targetOrder:
            return order
    return "Order not found"
```

Listing 1. Pseudocode for Linear Search Implementation

Explanation: The algorithm iterates through each order in the dataset, comparing it to the target order. This approach has a time complexity of $O(n)$, where n is the number of orders. Although

not the fastest method, Linear Search is efficient for small datasets and unsorted lists (tarunsarawgi_gfg, 2024).

Binary Search Implementation

Binary Search is a more efficient algorithm that divides the list in half with each comparison, allowing it to locate the target order quickly. However, Binary Search requires the dataset to be sorted in advance, making it best suited for large, organized lists where high-speed retrieval is necessary.

In the Online Bookstore System, Binary Search is ideal for tracking orders by unique identifiers, such as order IDs, or when searching by dates in a chronologically ordered list. This ensures that the system can quickly retrieve specific orders without scanning the entire dataset.

```
function binarySearch(sortedOrders, targetOrder):
    left = 0
    right = length of sortedOrders - 1
    while left <= right:
        mid = (left + right) / 2
        if sortedOrders[mid] == targetOrder:
            return sortedOrders[mid]
        elif sortedOrders[mid] < targetOrder:
            left = mid + 1
        else:
            right = mid - 1
    return "Order not found"
```

Listing 2. Pseudocode for Binary Search Implementation

Explanation: In Binary Search, the algorithm repeatedly divides the sorted list in half, comparing the middle element to the target order. If the target is less than the middle element, it focuses on the left half; if greater, it focuses on the right half. Binary Search has a time complexity of $O(log(n))$*, making it significantly faster than Linear Search for large, sorted datasets* (GeeksforGeeks, 2019).

Choosing the Right Algorithm for Order Tracking

For the Online Bookstore System, the choice between Linear Search and Binary Search depends on the specific requirements:

- Linear Search is implemented for unsorted or small lists, allowing flexibility when searching for orders without predefined structure.
- Binary Search is implemented for large, sorted lists where high-speed retrieval is essential, such as in order ID searches.

By implementing both algorithms, the system can efficiently handle different types of order tracking scenarios, improving responsiveness and customer service.

5.3. Chosen Implementation for Searching Algorithm

In this project, the searching function implements a linear search algorithm for searching through orders by attributes such as id, title, and author. Linear search is a straightforward approach where each item in the list is checked one-by-one until a match is found or the list ends. Despite its $O(n)$ Time complexity, linear search is optimal for this project because it effectively handles small to moderately sized datasets without the overhead of sorting or additional indexing.

Justification for Linear Search

a. Simplicity: Linear search is relatively easy to implement and understand. Given that our dataset of orders is stored sequentially, linear search can directly access and check each item, ensuring compatibility with the existing data structure.
b. Flexibility: Linear search allows for partial or relative matches within fields like title or author, enabling a broader search without needing exact matches. This flexibility is particularly useful when users may not have complete details or need to search using

various keywords. For example, a user could search for any title containing "Harry" or any author named "Orwell."

c. Low Overhead for Small Datasets: As pointed out in (GeeksforGeeks, 2024), linear search is efficient for small datasets. Given the scale of this bookstore system, linear search provides fast access without requiring additional resources or preprocessing, making it ideal for our application.
d. Unsorted Data Compatibility: Unlike binary search, which requires sorted data, linear search works effectively with unsorted data. This feature is advantageous since our orders list may not always be sorted, especially as new orders are continuously added.

Comparison to Other Search Algorithms

While binary search offers $O(log(n))$ time complexity, it requires a sorted dataset, and sorting can add significant overhead, especially for dynamic lists where new data is frequently added or removed. Binary search is most effective when there are a large number of static entries and a high volume of searches, but in our system, the added overhead from sorting would outweigh its benefits. Therefore, linear search is the most suitable option for the current needs of this bookstore system.

6. Implementation of Data Structures and Algorithms

6.1. Code Implementation of Queue for Order Management

In this project, the queue is implemented using a linked list approach to efficiently manage customer orders. The queue follows a First-In, First-Out (FIFO) system, ensuring that orders are processed in the sequence they are placed, which is ideal for managing customer requests in an orderly fashion. Each order includes a unique ID, customer details, and a list of books being ordered.

Queue Structure

The OrderQueue class utilizes a Node structure to represent each order:

- Each Node contains:
    - Order ID: A unique identifier for tracking.
    - Customer: Details of the customer placing the order, stored as a Customer object.
    - Books: An array of Book objects representing the items in the order.
    - Next Node: A reference to the next Node, enabling linked list functionality.

This structure supports efficient insertion and deletion operations, making it ideal for a dynamic, constantly changing order queue.

Operations in OrderQueue

The OrderQueue provides several essential operations:

a. Enqueue: Adds a new order to the end of the queue.
    - Before adding an order, the system checks the stock of each book in the order. If any book is out of stock, the order is not added to the queue, and a message is displayed indicating the stock issue.

    - If all books are in stock, the order is successfully added to the queue, and stock counts for each book are decremented accordingly.

b. Dequeue: Removes and processes the order at the front of the queue.
    - This method dequeues the oldest order and displays the order details, including customer information and the list of books.
    - After processing, the order details, along with the total price, are saved to output.csv for record-keeping.

c. Save to CSV: This helper method logs each processed order to an external CSV file (output.csv), allowing for easy tracking and record management. The log includes the order ID, customer details, timestamp, ordered books, and total price.

d. Display Queue: Although not shown explicitly, a displayQueue method would allow viewing all orders currently in the queue, helping staff understand the current workload.

Chosen Implementation for Queue

The queue uses Java's LinkedList structure, represented by nodes linked in a linear fashion. This approach was chosen because:

- Flexibility: The linked list allows dynamic resizing, making it suitable for varying queue sizes.
- Efficiency: Adding and removing nodes are efficient operations with minimal overhead.
- Orderly Processing: The FIFO system ensures orders are handled in the exact sequence they were placed, providing predictability in order fulfillment.

Testing

Test Case ID	Description	Input(s)	Action	Expected Output	Actual Output	Pass/ Fail
QOM-TC1	Enqueue a new order	Order with ID 001, Customer C123, 3 Books	Enqueue order	Order is added to the queue, size increases	Order is added to the queue, size increases	Pass
QOM-TC2	Dequeue an order from queue	None	Dequeue order	Oldest order (ID 001) is removed from the queue, order details printed	Oldest order (ID 001) is removed from the queue, order details printed	Pass
QOM-TC3	Attempt to dequeue from an empty queue	Queue is empty	Dequeue order	Message: "Queue is empty. No order to process."	Message: "Queue is empty. No order to process."	Pass
QOM-TC4	Enqueue order with out-of-stock book	Order with ID 002, 1 book with stock = 0	Enqueue order	Message indicating stock issue, order not added to queue	Message indicating stock issue, order not added to queue	Pass

Table 3. Test Cases for Queue Order Management

Summary of Queue Benefits in Order Management

a. First-In, First-Out Processing: Guarantees that the earliest orders are fulfilled first.

b. Scalability: The LinkedList design supports flexible queue length, handling both small and large volumes of orders.

c. Automated Stock Checks: Each time an order is enqueued, the system verifies stock, reducing the risk of fulfilling out-of-stock orders.

d. Data Logging: By saving processed orders to a CSV file, the system creates an accessible history of transactions, aiding in inventory and financial tracking.

This design ensures that order processing remains streamlined, accurate, and efficient, providing a strong foundation for customer order management in the Online Bookstore System.

6.2. Code Implementation of Stack for Completed Orders

In this section, we detail the implementation of the stack data structure to handle completed orders within the Online Bookstore System. The stack follows a Last-In, First-Out (LIFO) principle, which aligns with the requirement to access the most recently completed orders first. This approach is useful for tasks such as viewing recent transactions or quickly accessing the last processed orders.

Stack Structure

The StackCompletedOrder class implements the stack using a linked list-based structure, where each completed order is represented by a Node. This structure provides efficient push and pop operations, allowing orders to be added and removed with minimal overhead. Each Node in the stack holds details about a completed order, including:

- Order ID: The unique identifier of the order.
- Customer Details: Information about the customer who placed the order.
- Ordered Books: A list of Book objects representing the items included in the order.
- Total Price: The calculated cost of the order.
- Next Node: A pointer to the next order in the stack, supporting the LIFO structure.

Key Operations in StackCompletedOrder

The StackCompletedOrder class provides the following essential operations to manage the stack of completed orders:

a. Push: Adds a completed order to the top of the stack.

- When an order is processed in the queue and marked as completed, it is pushed onto the stack. This operation allows the most recent completed order to be accessed first, in keeping with the LIFO principle.

b. Pop: Removes the top order from the stack.

- The pop operation retrieves and removes the most recently completed order from the stack. This feature is useful for undoing actions on the last processed order or for retrieving recent transaction details.

c. Peek: Views the top order without removing it.

- peek allows the user to examine the most recent completed order without removing it from the stack. This feature is useful for quick access to the latest order details, providing an overview without altering the stack's structure.

d. Save to CSV: A helper method to log completed orders in a CSV file, completedOrder.csv.

- This method ensures that each completed order is saved with details such as order ID, customer name, order date, ordered books, and total price. This log is essential for maintaining a permanent record of all transactions and facilitates easy access to past orders.

Chosen Implementation for Stack

The stack implementation leverages a linked list to allow dynamic resizing and efficient push and pop operations. This choice is ideal for the completed orders stack because:

- Dynamic Size: The stack can grow or shrink as needed, depending on the number of completed orders.

- Efficient Access: Push and pop operations are handled in constant time, ensuring that recent orders are quickly accessible.
- Persistence: Saving completed orders to completedOrder.csv provides a historical record, maintaining consistency across the order management system.

Example Usage

Here's a brief overview of how the stack manages completed orders in practice:

a. Adding a Completed Order: Once an order is processed from the queue, it is pushed to the stack. This action places the completed order on top of the stack for easy access.

b. Viewing Recent Orders: The peek operation allows viewing of the latest completed order, enabling quick reference without affecting the stack structure.

c. Removing Completed Orders: Orders are removed from the stack in reverse order of their completion, supporting scenarios where recent transactions need to be accessed first.

Testing

Test Case ID	**Description**	**Input(s)**	**Action**	**Expected Output**	**Actual Output**	**Pass/ Fail**
SCO-TC01	Push a completed order onto the stack	Completed Order with ID 001	Push order	Order is added to the stack, stack size increases	Order is added to the stack, stack size increases	Pass
SCO-TC02	Peek at the most recent completed order	None	Peek at stack	Most recent order displayed without removal	Most recent order displayed without removal	Pass
SCO-TC03	Pop an order from the stack	None	Pop order	Most recent order removed from stack, details	Most recent order removed from stack, details	Pass

				printed	printed	
SCO-TC04	Attempt to pop from an empty stack	Stack is empty	Pop order	Message: "Stack is empty. No order to process."	Message: "Stack is empty. No order to process."	Pass

Table 4. Test Cases for Stack of Completed Orders

Conclusion

By implementing a stack for completed orders, the Online Bookstore System ensures that the latest transactions are prioritized for access, aligning with both user requirements and efficient data handling practices.

6.3. Sorting Algorithm Code Implementation

In this section, we discuss the implementation of the merge sort algorithm in the Online Bookstore System, used to sort the collection of books by various fields (e.g., title, author, price) based on user-specified criteria. Merge sort is an efficient, stable, and comparison-based sorting algorithm with a time complexity of $O(nlog(n))$, making it well-suited for larger datasets.

Sorting Structure

The BookSorter class is designed to handle sorting operations on the list of Book objects. By implementing a merge sort algorithm, the system provides both flexibility and efficiency in sorting the book catalog by multiple fields.

The class offers:

- Field-Based Sorting: Users can sort books by id, title, author, or price.
- Order Specification: Sorting can be done in either ascending or descending order, based on user preference.

Key Methods in BookSorter

The BookSorter class includes the following core components:

a. mergeSort: The main method that initiates the merge sort algorithm on a list of Book objects.
    - This method takes three parameters:
        - books: The list of books to be sorted.
        - field: A string specifying the field to sort by (e.g., id, title, author, price).
        - ascending: A boolean indicating the sort order (true for ascending, false for descending).
b. mergeSortHelper: A recursive helper function that divides the list into halves until each part contains a single element, then merges these parts in sorted order.
    - The helper function performs the core merge sort operations, following the divide-and-conquer strategy by recursively splitting the list and merging sorted halves.
c. merge: Merges two halves of the list in a sorted order based on the specified field and order.
    - This function compares elements from each half according to the given field, ensuring that they are merged into a fully sorted list.
    - Comparisons are made dynamically by checking the specified field. For example, if the title field is specified, books are compared alphabetically by title.

Example of Sorting Implementation

Here's a brief example of how the merge sort implementation manages the sorting process:

a. Initiate Sort by Title: When sorting by title in ascending order, mergeSort is called with field = "title" and ascending = true. The list is divided into smaller sub-lists until each contains a single book.

b. Merge Process: As the sub-lists are merged, books are compared by their title, ensuring they are ordered alphabetically.
c. Descending Order Sorting: When ascending = false, books are arranged in reverse order, providing a descending sort.

The merge sort algorithm maintains the relative order of books with equal field values, ensuring stability in the sorted list. This is particularly useful when sorting by price or ID, where books with the same price or ID should appear in their original order.

Chosen Implementation for Sorting

The merge sort algorithm was chosen for its efficiency and stability. Its $O(nlog(n))$ time complexity ensures that even larger datasets are sorted quickly, and the divide-and-conquer nature of merge sort is well-suited to managing complex sorting requirements in a bookstore system.

- Efficient for Large Datasets: Merge sort's time complexity of $O(nlog(n))$ makes it an ideal choice for large collections of books.
- Stable Sorting: Ensures that books with equal values in the specified field retain their original order.
- Flexible Order and Field Sorting: Allows sorting by different fields in both ascending and descending order, making it adaptable to various user needs.

Example Usage

The following example demonstrates how merge sort is utilized within the Online Bookstore System:

- Sorting by Author in Ascending Order: mergeSort is called with field = "author" and ascending = true, arranging books alphabetically by author.

- Saving Sorted Books to CSV: Once sorted, the list of books is written to sorted_books.csv for easy reference.

Testing

Test Case ID	Description	Input(s)	Action	Expected Output	Actual Output	Pass/ Fail
MS-TC01	Sort books by title in ascending order	List of book with varied titles	Sort by title, ascending	Books sorted alphabetically by title (A-Z)	Books sorted alphabetically by title (A-Z)	Pass
MS-TC01	Sort books by price in descending order	List of books with varied price	Sort by price, descending	Books sorted by price from highest to lowest	Books sorted by price from highest to lowest	Pass
MS-TC01	Sort by author with identical field values	List of books with identical author names	Sort by author, ascending	Books sorted alphabetically by author name (A-Z)	Books sorted alphabetically by author name (A-Z)	Pass
MS-TC01	Sort empty list of books	Empty book list	Sort by any field	Empty list remains unchanged, no errors	Empty list remains unchanged, no errors	Pass

Table 5. Test Cases for Sorting Algorithm (Merge Sort)

Conclusion

By implementing merge sort for this application, the Online Bookstore System is able to efficiently manage and display a well-organized list of books, allowing users to easily find and browse titles in their preferred order.

6.4. Searching Algorithm Code Implementation

This section outlines the implementation of the linear search algorithm within the Online Bookstore System, allowing users to search for orders by various attributes, including orderId, customerId, customerName, bookName, and orderDate. The linear search algorithm was selected

due to its simplicity and versatility, which accommodates partial and case-insensitive keyword searches across unsorted datasets. This approach is well-suited for the dynamic nature of order entries in the application.

Searching Structure

The OrderSearcher class is designed to facilitate flexible search capabilities across multiple fields in Order objects. By employing a linear search approach, each item in the list is inspected to see if it matches the search term provided, supporting partial matches and case-insensitive searches across multiple fields.

The key components of OrderSearcher include:

- Flexible Search Criteria: Users can search by orderId, customerId, customerName, bookName, or orderDate.
- Case-Insensitive Matching: The algorithm ignores case differences, making searches more user-friendly.
- Partial Matching: Supports partial matches within each field, enhancing usability by allowing searches without exact details.

Key Methods in BookSearcher

Here's a rewritten **Searching Algorithm Code Implementation** section tailored to the `Order` search functionality.

Searching Algorithm Code Implementation

Data Structures and Algorithms for Online Bookstore Systems

This section outlines the implementation of the linear search algorithm within the Online Bookstore System, allowing users to search for orders by various attributes, including `orderId`, `customerId`, `customerName`, `bookName`, and `orderDate`. The linear search algorithm was selected due to its simplicity and versatility, which accommodates partial and case-insensitive keyword searches across unsorted datasets. This approach is well-suited for the dynamic nature of order entries in the application.

Searching Structure

The OrderSearcher class is designed to facilitate flexible search capabilities across multiple fields in Order objects. By employing a linear search approach, each item in the list is inspected to see if it matches the search term provided, supporting partial matches and case-insensitive searches across multiple fields.

The key components of OrderSearcher include:

- Flexible Search Criteria: Users can search by orderId, customerId, customerName, bookName, or orderDate.
- Case-Insensitive Matching: The algorithm ignores case differences, making searches more user-friendly.
- Partial Matching: Supports partial matches within each field, enhancing usability by allowing searches without exact details.

Key Methods in OrderSearcher

The OrderSearcher class includes the following core method:

- searchOrders: This is the primary method for searching orders based on a given search term. It iterates through each Order in the list and checks if the search term is present in any of the specified fields (orderId, customerId, customerName, bookName, or

orderDate). Each field is converted to lowercase for case-insensitive matching, and partial matches are identified using the contains method.

Field-Specific Matching: Inside searchOrders, the search term is compared against each field independently:

- Order ID: Matches if the search term is contained within the orderId.
- Customer ID: Matches if the search term is contained within the customerId.
- Customer Name: Checks if the customer's name contains the search term, supporting partial matches.
- Book Name: Iterates over each book title in the order to determine if any title matches the search term.
- Date: Compares only the date portion of orderDate to match a specific day, providing accurate date-based searches.

Example of Searching Implementation

Here's a step-by-step example of how the linear search implementation handles order searches:

- Search by Order ID: If a user enters an exact or partial orderId, searchOrders will locate any matching orders.
- Search by Customer Name: Matches any orders containing the specified name, allowing user-friendly, case-insensitive name-based searches.
- Search by Book Title: Locates any orders that include a specific book title, providing flexibility in searching for orders based on book details.
- Combined Field Matching: If the search term applies to multiple fields, searchOrders will return all relevant orders, ensuring comprehensive search results.

Chosen Implementation for Searching

The linear search algorithm is particularly suitable for this system because:

- Flexibility: Linear search supports unsorted data, allowing the list of orders to grow dynamically without needing additional sorting or restructuring.
- Efficiency for Moderate Datasets: With an average time complexity of O(n), linear search is ideal for the dataset size anticipated in this application.
- Broad Matching Capability: The implementation enables partial matches, case-insensitivity, and compatibility across multiple fields, enhancing the search experience for users.

Example Usage

Below is an example demonstrating the search functionality in the Online Bookstore System:

- User Search by Order ID: If a user searches with a specific orderId, the system inspects each order's orderId field to find a match.
- User Search by Keyword in Customer Name or Book Title: For partial text searches in the customerName or bookName fields, all orders with matching text are returned, making it easy to locate orders without exact details.
- Search Results: Matching orders are displayed to the user, providing quick access to relevant data.

Testing

Test Case ID	**Description**	**Input(s)**	**Action**	**Expected Output**	**Actual Output**	**Pass/ Fail**
LS-TC01	Search for an order by exact ID	Search term: "001"	Search by orderID	Returns the order with orderID "001"	Returns the order with orderID "001"	Pass

LS-TC02	Search for orders by partial name	Search term: “Son”	Search by customerName	Returns all orders for customer “Son Nguyen”	Returns all orders for customer “Son Nguyen”	Pass
LS-TC03	Search by partial book title	Search term: “Mocking”	Search by bookName	Returns orders containing the book “To Kill a Mockingbird”	Returns orders containing the book “To Kill a Mockingbird”	Pass
LS-TC04	Search by non-existent term	Search term: “XYZ123”	Search by every terms	Returns message: “No orders found matching the search term: XYZ123”	Returns message: “No orders found matching the search term: XYZ123”	Pass

Table 6. Test Cases for Searching Algorithm (Linear Search)

Conclusion

By implementing this linear search functionality, the Online Bookstore System provides an efficient and user-friendly way to find orders, thereby enhancing the navigation and usability of the system. This approach offers robust search capabilities suitable for the expected scale of the application, ensuring that users can retrieve relevant information with minimal effort.

7. Complexity Analysis and Trade-offs

7.1. Time Complexity: Best, Average, and Worst Case

This section provides a detailed analysis of the time complexities associated with the key operations implemented within the Online Bookstore System. The algorithms and data structures analyzed include the queue for order management, the stack for completed orders, the merge sort algorithm for sorting books, and linear search for finding books. Time complexities are evaluated in terms of best, average, and worst-case scenarios, where applicable.

Queue for Order Management

The OrderQueue class manages customer orders, primarily through enqueue and dequeue operations.

- Enqueue Operation: The time complexity of enqueuing a new order is influenced by the stock verification of each book in the order. In the best case, this operation completes in $O(1)$ if each book is verified to be in stock quickly. However, in the worst case, it requires $O(m)$ time, where m represents the number of books, as each must be checked before adding the order to the queue.
- Dequeue Operation: The dequeue operation, which removes the front order from the queue, generally runs in $O(1)$ as it only involves removing the front node. However, updating stock for each book in the order can increase complexity to $O(b)$, where b is the number of books in the order.
- Save to CSV Operation: The method that logs completed orders to a CSV file runs in $O(b)$, with each book's details written to the file. For smaller order sizes, this complexity can be approximated as $O(1)$.

Operation	**Best Case**	**Average Case**	**Worst Case**
Enqueue	$O(1)$	$O(m)$	$O(m)$
Dequeue	$O(1)$	$O(b)$	$O(b)$

Table 7. Summary of Time Complexities of Order Queue

Description

- *m: number of books add to queue*
- *b: number of books in the order*

Stack for Completed Orders

The StackCompletedOrder class uses a stack to manage completed orders, providing operations for push, pop, and peek.

- Push Operation: Adding an order to the stack's top runs in $O(1)$ as it only involves updating the top reference.
- Pop Operation: Removing the top order also operates in $O(1)$, requiring no traversal of the stack.
- Peek Operation: Accessing the top order without removal is achieved in $O(1)$ since it only references the top node.
- Load from CSV Operation: The load operation that initializes the stack with completed orders from a CSV file has a time complexity of $O(n)$ where n is the number of rows in the file.

Operation	**Best Case**	**Average Case**	**Worst Case**
Push	$O(1)$	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$	$O(1)$

Table 8. Summary of Time Complexities of Completed Orders Stack

Description

- *n: number of rows in the file*

Sorting Algorithm (Merge Sort)

The BookSorter class uses merge sort to arrange books by fields like title, author, and price. Merge sort has a consistent time complexity across cases due to its divide-and-conquer nature.

- Merge Sort Operation: The merge sort algorithm operates in $O(nlog(n))$ time complexity for best, average, and worst cases. This consistency stems from recursively dividing the list into smaller sub-lists, merging them in sorted order. This complexity makes merge sort efficient for large datasets, as required in the bookstore system.

Operation	**Best Case**	**Average Case**	**Worst Case**
Merge Sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$

Table 9. Summary of Time Complexities of Book Sorter (Merge Sort)

Description

- *n: number of books*

Searching Algorithm (Linear Search)

The OrderSearcher class implements a linear search algorithm designed to inspect each order in the list, enabling matches based on attributes such as orderId, customerId, customerName, bookName, and orderDate. This approach provides flexible and user-friendly search capabilities, supporting partial matches and case-insensitive searches across unsorted data.

In the linear search algorithm, the search process begins from the first entry in the list and checks each order until a match is found or the end of the list is reached. The performance of linear search varies depending on the location of the match:

- Best Case: O(1), if the first order in the list matches the search term.

- Average Case: O(n), where each order may need to be checked, especially if matches are located in the middle of the list.
- Worst Case: O(n), in cases where the matching order is at the end of the list or no match is found.

This flexibility in handling partial matches and unsorted data makes linear search a suitable choice for applications where the dataset may change frequently and a user-friendly, dynamic search is desired.

Operation	**Best Case**	**Average Case**	**Worst Case**
Linear Search	$O(1)$	$O(n)$	$O(n)$

Table 9. Summary of Time Complexities of Book Searcher (Linear Search)

Description

- *n: number of orders*

7.2. Space Complexity of Data Structures and Algorithms

The space complexity of data structures and algorithms in the Online Bookstore System is evaluated based on each operation, with a focus on how memory usage scales with the number of elements involved. This analysis includes the OrderQueue and StackCompletedOrder classes (both implemented with linked lists), the BookSorter (merge sort algorithm), and the BookSearcher (linear search algorithm). Each section provides a breakdown of space requirements for common operations.

Space Complexity of Operations in OrderQueue (Linked List)

The OrderQueue class manages customer orders using a linked list, with each node storing order details, customer information, and book data.

Operation	**Space Complexity**	**Explanation**

Enqueue	$O(1)$	Each new order node is added to the rear, requiring fixed memory per addition.
Dequeue	$O(1)$	Removes the front node without additional memory allocation.
Stock Check during Enqueue	$O(m)$	Where m is the number of books in an order. Temporary space is used to check stock availability

Table 10. OrderQueue Operations' Space Complexity

The overall space complexity of the OrderQueue is O(n), where n is the number of nodes in the queue, each node storing constant-size order data.

Space Complexity of Operations in StackCompletedOrder (Linked List)

The StackCompletedOrder class uses a stack data structure to manage completed orders. Each operation is analyzed for its auxiliary space usage.

Operation	**Space Complexity**	**Explanation**
Push	$O(1)$	Each new node is added to the top, requiring constant memory.
Pop	$O(1)$	Removes the top node without additional memory allocation.
Peek	$O(1)$	Accesses the top node without altering memory usage.

Table 11. StackCompletedOrder Operations' Space Complexity

The total space complexity for StackCompletedOrder is O(n), where n represents the number of orders pushed onto the stack. Each completed order node requires constant space.

Space Complexity of Merge Sort in BookSorter (ArrayList)

The BookSorter class implements merge sort for sorting books. Merge sort requires auxiliary space for merging subarrays.

The merge sort's overall space complexity is O(n) due to the auxiliary space needed to store temporary copies during merging. The recursion depth also adds O(log(n)) for the stack space, but the primary complexity remains O(n).

Space Complexity of Operations in Linear Search (BookSearcher)

The BookSearcher class implements linear search, which does not require additional data structures, resulting in minimal memory usage.

The space complexity of linear search is O(1), as it requires only a small amount of additional memory to store temporary variables.

8. Critical Evaluation of ADT and Algorithm Usage

8.1. Evaluation of ADTs in System Design

In the design of the Online Bookstore System, various Abstract Data Types (ADTs) play a significant role in structuring data for effective processing and retrieval. ADTs, such as Queues, Stacks, and Lists, provide a foundation for organizing order management, completed orders, and inventory tracking in a systematic and efficient manner. By defining data structures through ADTs, the system benefits from organized data access patterns and optimized handling of specific operational requirements.

Queue as a Data Structure for Order Management

The Queue ADT was selected to manage incoming customer orders due to its First-In, First-Out (FIFO) property. In the context of order processing, this property ensures that orders are processed in the sequence they are received, reflecting a fair and logical processing flow. The Queue ADT is suitable for this purpose because:

a. Order Processing: Orders are managed in a way that each customer is served in a timely manner.
b. Ease of Access: Queue operations like enqueue and dequeue simplify adding new orders and removing processed ones.

Stack as a Data Structure for Completed Orders

The Stack ADT is employed to store completed orders temporarily. This data structure uses a Last-In, First-Out (LIFO) order, making it ideal for reviewing the most recent transactions. The Stack ADT enhances the system's functionality by providing:

a. Recent Transaction Access: Completed orders are easily accessible, allowing quick verification of the latest processed orders.

b. Efficient Storage and Retrieval: The Stack's push and pop operations streamline data management and minimize overhead.

List as a Data Structure for Inventory Tracking

For managing inventory and book information, a List ADT was chosen, allowing the storage of books with attributes like title, author, price, and stock. The List ADT provides:

a. Direct Access: The linear structure supports direct access and searching within the inventory, which is crucial for dynamically adding or updating book data.
b. Flexibility for Searching: The List ADT supports various search operations, enabling the system to retrieve book information based on different attributes.

Benefits of ADTs in System Design

Incorporating these ADTs into the system design has brought several advantages, such as:

a. Data Organization: Each ADT provides a clear data-handling approach, ensuring that information is stored and processed logically.
b. Optimized Operations: ADTs are chosen based on their operational strengths, such as FIFO for order queues and LIFO for completed orders, which improves the efficiency of order handling.
c. Scalability: The use of ADTs like Lists allows for scalable inventory tracking, as new books can be added without requiring structural changes.

By carefully selecting and implementing ADTs for specific functions, the Online Bookstore System benefits from a robust and maintainable data management structure. This approach also enhances overall system performance, as each ADT is optimized to handle its respective tasks effectively.

8.2. Benefits and Challenges of Data Structures in the Project

In this project, implementing specific data structures brought significant benefits in terms of organization, efficiency, and functionality:

a. Efficient Data Management: Using queues and stacks enabled systematic handling of customer orders and completed orders. The queue structure allowed for processing customer orders in a First-In-First-Out (FIFO) manner, which aligns well with order management workflows by ensuring that the earliest orders are prioritized. Likewise, the stack data structure was beneficial for managing completed orders, with its Last-In-First-Out (LIFO) arrangement fitting scenarios where the latest orders are reviewed or managed first.

b. Enhanced Search and Sorting Capabilities: The use of lists and arrays facilitated flexible searching and sorting within the system. Implementing linear search in unsorted lists allowed for simple, partial, and case-insensitive searches. This flexibility significantly improved the user experience in locating specific books or orders. Additionally, applying sorting algorithms like merge sort provided users with the ability to view data in organized forms, such as by title, author, or price, improving data accessibility and readability.

c. Memory Management: By choosing efficient data structures, the project could optimize memory usage. For example, using linked lists in stacks minimized memory wastage by dynamically allocating memory only when required. Such a setup helped in efficiently managing memory, particularly when dealing with unpredictable volumes of completed orders.

Challenges of Using Data Structures

a. Complexity in Implementation: Implementing the stack and queue structures with additional features, such as stock management in the order queue, presented some complexities. Ensuring accurate stock updates while dequeuing orders required additional handling, which increased the overall complexity of the code and required careful debugging.
b. Time Complexity of Search Operations: Since the project utilized linear search due to its simplicity and flexibility, there were trade-offs in terms of efficiency. For large datasets, linear search could become inefficient, as it requires scanning each element until a match is found. This limitation could impact performance as the system scales up, particularly when searching for orders within extensive order history records.
c. Maintenance and Extensibility: As the system grows and new functionalities are added, maintaining data consistency across multiple data structures could pose challenges. The need to ensure synchronization between different data structures, such as when orders are moved from the queue to the stack, could lead to higher maintenance efforts and increase the potential for bugs if not managed carefully.

In summary, while the chosen data structures greatly benefited the project by organizing and improving data management, they also introduced challenges in implementation complexity, efficiency, and long-term maintenance. Balancing these benefits and challenges is crucial for effective system performance and reliability.

9. Conclusion

9.1. Summary of Findings

This project demonstrated how implementing the right data structures and algorithms can significantly enhance the efficiency, functionality, and user experience of an online bookstore system. By incorporating a queue structure for managing customer orders and a stack for organizing completed orders, the system maintained logical order processing workflows, aligning well with real-world requirements.

The use of merge sort allowed for consistent and efficient sorting of book records based on user-defined criteria, ensuring that the system could handle large datasets without sacrificing performance. Although linear search was chosen for its flexibility in handling unsorted data, the trade-off with efficiency was recognized as a limitation for future scalability. Nevertheless, linear search provided a user-friendly approach that allowed for partial and case-insensitive searches, enhancing the overall usability of the bookstore system.

In addition to data organization and retrieval, implementing these data structures facilitated efficient memory usage, reducing unnecessary overhead in dynamic data processing. The project’s findings underscore the importance of carefully selecting data structures and algorithms based on system requirements, balancing performance, flexibility, and complexity to create a responsive and adaptable system that meets both current and potential future needs.

Overall, the project's design choices allowed for a streamlined, efficient, and easily navigable system, reflecting thoughtful planning in algorithm selection and data structure usage.

9.2. Key Contributions of Data Structures and Algorithms

The use of data structures and algorithms played a central role in shaping the functionality and performance of this online bookstore system. Each chosen data structure and algorithm contributed to creating a system that is both efficient and user-friendly.

a. Efficient Order Management: The queue and stack structures were pivotal in managing customer orders and completed orders. By employing a queue for order processing, the system adhered to a First-In-First-Out (FIFO) method, ensuring that orders were processed in the order they were received. This structure effectively mirrors real-world customer service workflows, providing a systematic approach to order handling. Similarly, using a stack for completed orders allowed the system to quickly access and review the most recent orders, enhancing administrative efficiency in post-order processes.

b. Enhanced Data Accessibility through Sorting: The integration of merge sort enabled flexible and efficient sorting of books by attributes such as title, author, and price. This sorting algorithm, with its consistent time complexity of $O(nlog(n))$, allowed the system to manage large sets of data seamlessly. Users could view books in their preferred order, making it easier to browse and make purchasing decisions.

c. Flexible Searching Capabilities: The use of linear search offered the flexibility to search through unsorted book data by multiple fields—title, author, and ID. This approach was particularly user-friendly, allowing for partial matches and case-insensitive searches, which made finding books easier without requiring precise input. While linear search may not be the fastest option for large datasets, its adaptability and simplicity greatly benefited the usability of the system.

d. Optimized Memory Usage: The selection of dynamic data structures, like linked lists for queue and stack, minimized memory usage by allocating space only as needed. This approach proved to be both memory-efficient and scalable, ensuring that the system could handle fluctuating data volumes without excessive overhead.

In summary, the strategic selection of data structures and algorithms was crucial to building an effective, scalable, and user-centric system. Each component—from order management to sorting and searching—contributed to an optimized structure that supports efficient operations, easy data access, and a positive user experience. These contributions highlight the importance of foundational data structures and algorithms in creating robust applications tailored to specific needs.

9.3. Recommendations for Future Enhancements

To further improve the functionality, scalability, and user experience of the online bookstore system, several enhancements could be considered for future iterations:

1. Implementing More Efficient Search Algorithms: As the system's dataset grows, a more efficient search algorithm, such as binary search, could be considered to replace linear search. Binary search would require the dataset to be sorted but would significantly reduce search times to O(logn), making the system faster when dealing with large volumes of books or orders.
2. Integration of Hashing for Quick Lookup: Incorporating a hash table could allow for constant-time O(1) access to specific book or order details based on unique identifiers like ISBNs or order IDs. Hash tables would enhance the search performance and enable faster access to frequently requested information, particularly in high-traffic scenarios.

3. Using a Database for Persistent and Scalable Storage: Moving from CSV files to a database, such as MySQL or MongoDB, would provide greater scalability and reliability for data management. Databases allow for complex queries, data indexing, and easier handling of large datasets, which would be beneficial as the number of books and orders grows. Additionally, this transition would improve data consistency, security, and backup capabilities.
4. Enhanced User Experience with Advanced Sorting Options: Adding advanced sorting features, such as multi-criteria sorting (e.g., sorting by author and then by price), would make the system more versatile and user-friendly. This functionality would allow users to refine their searches and make more informed decisions.
5. Stock Management Enhancements: Introducing real-time stock management and alerting capabilities would improve the system's reliability in handling inventory. Features like automatic stock updates upon order completion, as well as notifications for low-stock items, would make inventory tracking more efficient and help prevent order delays.
6. Implementing Caching for Frequently Accessed Data: To reduce load times for frequently searched items, caching could be implemented. By storing popular search results temporarily, the system could provide faster responses to users and reduce the time required for repeated queries.
7. Improving Error Handling and Robustness: Enhancing error-handling mechanisms would make the system more reliable. For instance, handling scenarios such as duplicate orders, invalid input, and order conflicts would improve system stability and user satisfaction, especially in cases of high-volume transactions.

Incorporating these enhancements would increase the efficiency, scalability, and robustness of the online bookstore system, making it better equipped to handle larger datasets, higher user traffic, and more complex user requirements. These improvements would ultimately contribute to a more resilient, responsive, and user-focused application.

9.4. Reflection on the Impact on Order Processing Efficiency

The use of carefully selected data structures and algorithms had a significant impact on the efficiency of order processing in this online bookstore system. By implementing a queue structure for handling customer orders, the system ensured a First-In-First-Out (FIFO) approach, which aligns with natural workflows in customer service and order fulfillment. This setup allowed the system to process orders in the sequence they were received, minimizing delays and maintaining a structured order management process.

The inclusion of a stack structure for completed orders further streamlined post-order handling. This Last-In-First-Out (LIFO) structure allowed administrators to access the most recent orders quickly, making it easier to manage follow-up actions, such as reviews, adjustments, or refunds. The stack organization also provided an efficient way to store completed orders without requiring reorganization, maintaining a seamless flow between different order statuses.

Sorting and searching capabilities also contributed to order processing efficiency. The merge sort algorithm allowed for consistent sorting of books based on various attributes, like title and price, facilitating rapid access and better browsing for users. Meanwhile, the linear search algorithm offered flexible search functionality across multiple fields, enabling users to locate specific books or orders even without exact details. Although linear search is not the fastest for large datasets, it supported the flexibility required in an unsorted environment.

Overall, these data structures and algorithms collectively optimized order processing efficiency, ensuring a user-friendly experience and quick response times. This thoughtful approach to data organization and retrieval not only enhanced the functionality of the bookstore system but also laid the foundation for handling larger volumes of orders in the future. By reducing the time and complexity involved in managing orders, this system created an efficient, streamlined process that enhances customer satisfaction and operational effectiveness.

10. References

aayushi2402 (2022). *Applications, Advantages and Disadvantages of Stack*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-stack/.

ayushdey110 (2021). *Difference between Abstract and Concrete Data Structure*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/difference-between-abstract-and-concrete-data-structure/.

BYJUS. (n.d.). *Stacks and Its Applications for GATE |Data Structures*. [online] Available at: https://byjus.com/gate/stack-and-its-applications/.

code_r (2023). *What is Queue Data Structure?* [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-queue-data-structure/.

GeekforGeeks (2014). *Queue - Linked List Implementation - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/queue-linked-list-implementation/.

GeekforGeeks (2021). *Introduction to Tree Data Structure*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-to-tree-data-structure/.

GeeksForGeeks (2011). *Applications of Queue Data Structure*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applications-of-queue-data-structure/.

GeeksforGeeks (2014). *Quick Sort Algorithm*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/quick-sort-algorithm/.

GeeksforGeeks (2015a). *Queue Data Structure - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/queue-data-structure/.

GeeksforGeeks (2015b). *Stack Data Structure - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/stack-data-structure/.

GeeksforGeeks (2017a). *Abstract Data Types - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/abstract-data-types/.

GeeksforGeeks (2017b). *What is Array?* [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-array/.

GeeksforGeeks (2018). *Merge Sort - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/merge-sort/.

GeeksforGeeks (2019). *Binary Search - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/binary-search/.

GeeksforGeeks (2023). *Time Complexities of all Sorting Algorithms*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/.

GeeksforGeeks (2024). *Linear Search - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/linear-search/.

harendrakumar123 (2024a). *Array Data Structure Guide*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/array-data-structure-guide/.

harendrakumar123 (2024b). *Linked List Data Structure*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/linked-list-data-structure/.

Kenton, W. (2024). *First In, First Out - FIFO*. [online] Investopedia. Available at: https://www.investopedia.com/terms/f/fifo.asp.

programiz (2020). *Insertion Sort Algorithm*. [online] Programiz.com. Available at: https://www.programiz.com/dsa/insertion-sort.

Ravikiran , S. (2024). *Implementing Stacks in Data Structures*. [online] Simplilearn.com. Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures.

tarunsarawgi_gfg (2024). *Time and Space Complexity of Linear Search Algorithm*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-and-space-complexity-of-linear-search-algorithm/.

tutorialspoint (2020). *Data Structure and Algorithms - Stack - Tutorialspoint*. [online] Tutorialspoint.com. Available at: https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm.

tutorialspoint.com (2019a). *Data Structures and Algorithms Selection Sort*. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm.

tutorialspoint.com (2019b). *Data Structures and Algorithms Tree*. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm.

www.javatpoint.com. (n.d.). *Abstract data type in data structure - javatpoint*. [online] Available at: https://www.javatpoint.com/abstract-data-type-in-data-structure.

www.programiz.com. (n.d.). *Circular Queue (With Examples)*. [online] Available at: https://www.programiz.com/dsa/circular-queue.

www.programiz.com. (n.d.). *Queue Data Structure and Implementation in Java, Python and C/C++*. [online] Available at: https://www.programiz.com/dsa/queue.

www.tutorialspoint.com. (n.d.). *Data Structure and Algorithms Tutorial - Tutorialspoint*. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/index.htm.