**1649A\_coursework\_Pham Thanh Truc\_GCD220368**

**Github link:** <https://github.com/Truc4p/BooksApplication.git>

Online Bookstore Order Processing System

Table of Contents

[Task 1: 1](#_Toc182681996)

[1. Introduction 1](#_Toc182681997)

[2. System Overview 1](#_Toc182681998)

[3. Functional Requirements 1](#_Toc182681999)

[4. Non-Functional Requirements 4](#_Toc182682000)

[Task 2: 5](#_Toc182682001)

[Use case diagram 5](#_Toc182682002)

[Class diagram 7](#_Toc182682003)

[Code 10](#_Toc182682004)

[Run the app 53](#_Toc182682005)

[Task 3: 67](#_Toc182682006)

[Data Structures 67](#_Toc182682007)

[Algorithms 67](#_Toc182682008)

[Task 4: 68](#_Toc182682009)

[1. searchOrdersAsAdmin 68](#_Toc182682010)

[2. Search book history 68](#_Toc182682011)

[3. Method to get all orders as a queue 69](#_Toc182682012)

[4. searchOrdersAsCustomer 69](#_Toc182682013)

[5. addNewBook 70](#_Toc182682014)

[6. Changing Quantity in Cart 71](#_Toc182682015)

## Task 1:

### 1. Introduction

**1.1 Purpose**

The purpose of this document is to detail the requirements for the Online Bookstore Order Processing System. This system will utilize various data structures and algorithms to efficiently manage customer orders, allowing users to add books to their shopping cart, track orders, and perform searches on book inventory and order status. This SRS serves as a guide for the development team to understand and implement the specified functionalities.

**1.2 Scope**

The Online Bookstore Order Processing System will allow users to browse books, place orders, add books to their cart, and track the status of their orders. Key functionalities include adding books to the cart using an arraylist, saving search book history using a stack, managing orders with a queue, sorting book lists, and searching for orders. The system will optimize user experience through efficient data structures and algorithms.

**1.3 Definitions, Acronyms, and Abbreviations**

* **FIFO**: First-In-First-Out, used in queue operations.
* **LIFO**: Last-In-First-Out, used in stack operations.
* **ADT**: Abstract Data Type, an abstract model for data structures.
* **Queue**: Data structure to store orders in a sequential manner.
* **Stack**: Data structure to store recently added search book history.

### 2. System Overview

The system is designed to manage customer orders by using appropriate data structures and algorithms. A queue data structure will be employed to process customer orders sequentially, while a stack will handle search book history. Book lists will be sorted by title using quicksort, and binary search will allow users to track orders based on sorted order numbers.

### 3. Functional Requirements

**3.1. Queue Data Structure for Order Processing**

* **Description**: A queue follows the **First-In-First-Out (FIFO)** principle, ideal for handling customer orders sequentially.
* **Valid Operations**:
  + **Enqueue**: Add a new order to the queue.
  + **Dequeue**: Process the order at the front.
  + **Peek/Front**: View the next order to process.
  + **IsEmpty**: Check if the queue is empty.
  + **Size**: Check the number of orders in the queue.

**3.2. Stack ADT for Search book history**

* **Description**: A stack operates on a **Last-In-First-Out (LIFO)** basis, is used to keep track of the search history, allowing to view and restore previous search queries.
* **Valid Operations**:
  + **Push**: Add a new search history to the top of the stack.
  + **Pop**: Remove the top search history.
  + **Peek/Top**: Look at the top search history.
  + **IsEmpty**: Check if the stack is empty.
  + **Size**: Get the stack’s current size.

**3.3. List Data Structure for Managing Books in Each Order**

* **Description**: A list is a collection of items arranged in a sequence, suitable for storing and organizing book details within each customer order.
* **Valid Operations**:
  + **Add**: Append a new book to the list within an order.
  + **Remove**: Remove a book from the list if the customer cancels a selection.
  + **Insert**: Place a book at a specific position in the list.
  + **Get**: Retrieve details of a specific book.
  + **Size**: Count the number of books in the list.
  + **IsEmpty**: Check if the list has any books.
  + **Sort**: Organize the list of books (by title, author, etc.).
  + **Search**: Locate a book within the list based on criteria (title, author).

**3.4. Sorting Algorithms**

* **Description**: Sorting is used on the list of books within each order for better organization.
* **Available Algorithms**:
  + **Insertion Sort**: Efficient for small or nearly sorted lists.
  + **Selection Sort**: Picks the minimum element to place in order.
  + **Quick Sort**: Efficient for large datasets.
  + **Merge Sort**: Stable sorting for large lists, though it uses extra space.

A table with different colored boxes

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence

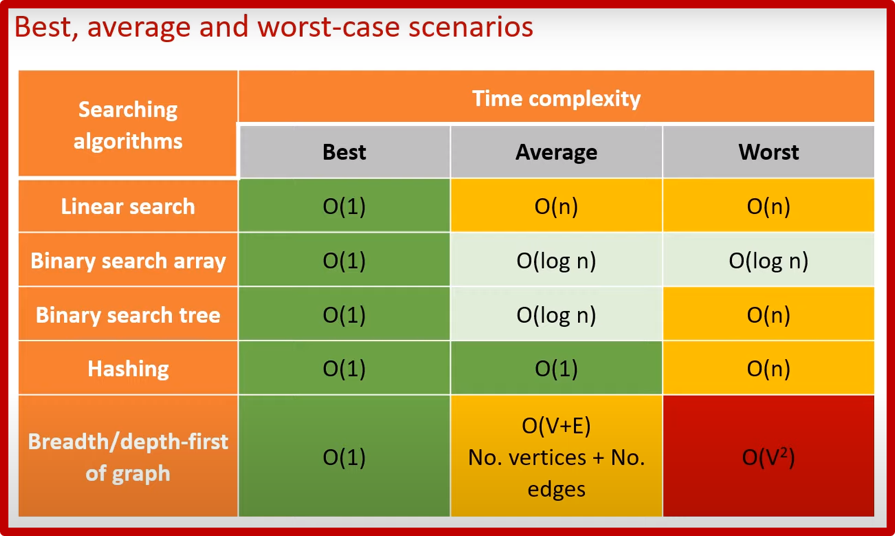
* **Operations**:
  + **Sort by Criteria**: Sort books based on title, author.

Using Algorithm: **Quicksort**

* Reason: Quicksort typically performs very well for string-based data like book titles because of its O(nlogn) average-case time complexity. It’s an in-place algorithm, so it doesn’t need extra memory (except for recursion overhead). Quicksort is often the fastest choice for large datasets.

**3.5. Searching Algorithms**

* **Description**: Searching helps customers find specific orders or book details.
* **Available Algorithms**:
  1. **Linear Search**: Sequential search, works on unsorted lists.
  2. **Binary Search**: Efficient on sorted lists.



* **Operations**:
  1. **Search by Order Number**: Locate an order by number.

Using Algorithm: **Binary Search**

Reason: If the order numbers are sorted, binary search will be efficient with a time complexity of O(logn). It significantly reduces the number of comparisons needed to find the order compared to a linear search (O(n)).

* 1. **Search by Book Details**: Within the book list, search by title or author.

Using Algorithm: **Linear Search**

Reason: If I use binary search, the list of books have to be sorted by title or author everytime before the searching or I need to create a sorted list by title or author and update the list everytime the book list changed, which make it even slower than linear search so I decided to use linear search.

### 4. Non-Functional Requirements

**4.1 Performance**

* The system should process each operation (enqueue, dequeue, push, pop) within milliseconds.
* Sorting and searching should be optimized for performance to handle large lists efficiently.

**4.2 Reliability**

* The system should handle multiple orders concurrently and maintain data integrity.
* Operations like enqueue and dequeue should maintain consistency in the queue even during system interruptions.

**4.3 Usability**

* Users should be able to perform tasks like adding books to a cart, viewing orders, and searching seamlessly.
* The UI should clearly display sorted and filtered book lists.

**4.4 Scalability**

* The system should scale to accommodate growing data sets (e.g., increasing number of orders, books).
* Sorting and searching algorithms should support efficient processing of large data volumes.

**5. Design Constraints**

* The system must use a FIFO queue for order processing and LIFO stack for search book history management.
* Sorting operations must prioritize efficiency, especially for larger book lists, using algorithms like quicksort and merge sort.
* Search functionality must rely on binary search where data is sorted, otherwise revert to linear search.

**6. Assumptions and Dependencies**

* The book inventory is assumed to be pre-loaded and updated regularly.
* Orders are assumed to be queued for processing as they are placed.
* Customers can only search for orders if they have an order number or specific criteria (like name or shipping address).

**7. Evaluation and Measurement**

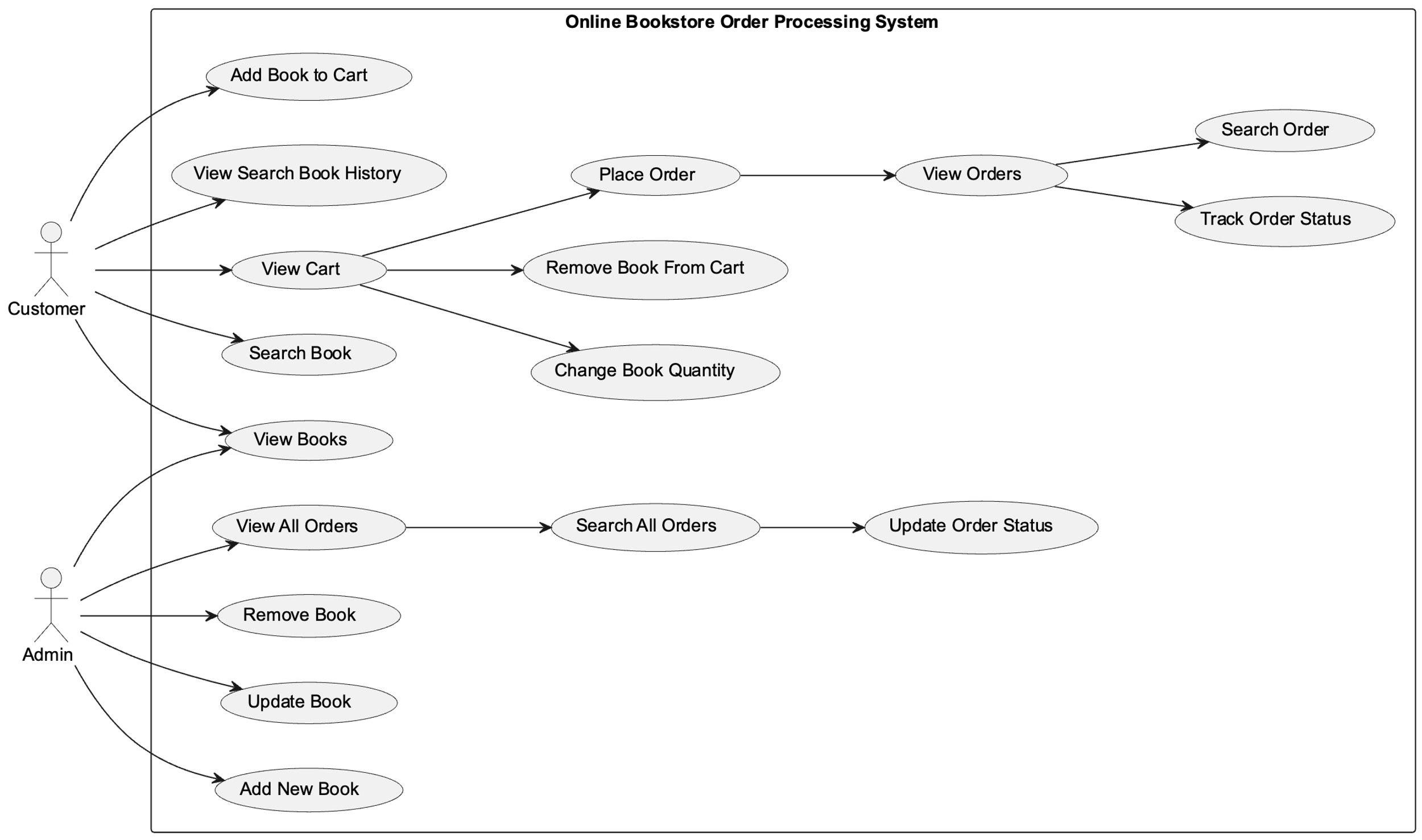
* **Efficiency Measurement**: Evaluate the time complexity of enqueue, dequeue, and push/pop operations.
* **Complexity Analysis**: Assess the time complexity of sorting (e.g., quicksort O(nlogn)) and searching algorithms (e.g., binary search O(logn)).

**8. Future Enhancements**

* Implementation of additional filters for book searches (e.g., genre, publication date).
* Enhanced sorting criteria, allowing users to sort by multiple attributes, such as title and author simultaneously.
* Real-time order tracking to allow customers to see live updates.

## Task 2:

### Use case diagram



**Explanation:**

The diagram illustrates the interactions between two primary actors: Customer and Admin.

**Use Cases for Customers**

Customers can engage in the following actions:

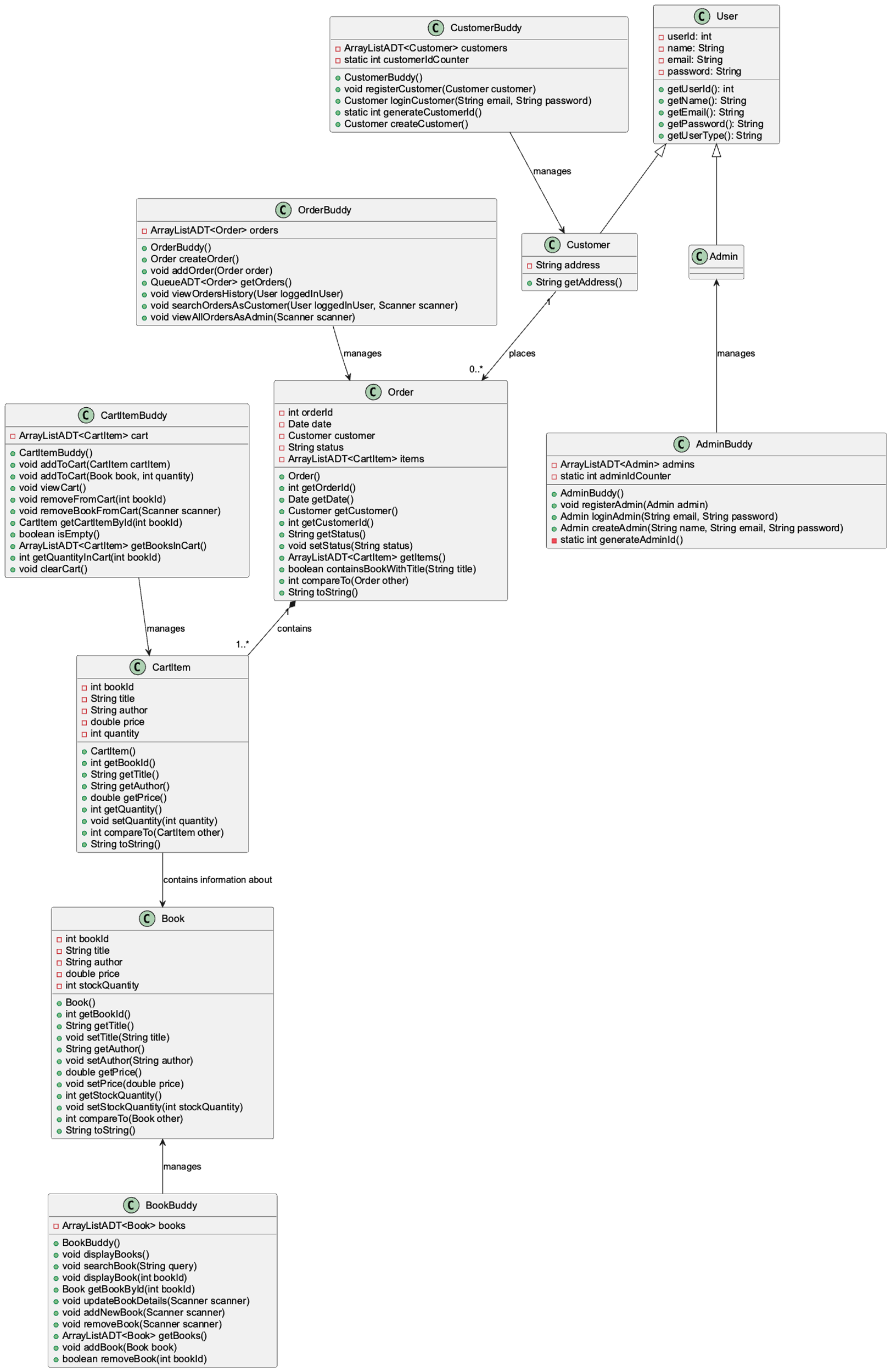
1. Search Book: Customers can search for books based on specific criteria like title, author, or ISBN.
2. View Books: Customers can browse through the available books in the system.
3. Add Book to Cart: Customers can add desired books to their shopping cart.
4. View Cart: Customers can review the items in their shopping cart.
5. Change Book Quantity: Customers can modify the quantity of books in their cart.
6. Remove Book from Cart: Customers can remove unwanted books from their cart.
7. Place Order: Customers can finalize their purchase by placing an order.
8. View Order: Customers can access their order history and view details of past purchases.
9. Search Their Order: Customers can search their order by order ID or a book title in the order.
10. Track Order Status: Customers can monitor the progress of their current orders.

**Use Cases for Admins**

Admins can engage in the following actions:

1. View All books: Admins can view a comprehensive list of all books.
2. Update Order Status: Admins can modify the status of orders to reflect changes in their processing or shipment.
3. Add New Book: Admins can add new books to the system's inventory.
4. Update Book: Admins can modify existing book information, such as title, author, price, or quantity.
5. Remove Book: Admins can remove books from the system's inventory.
6. View All Orders: Admins can view all orders of all customers.
7. Search All Orders: Admins can search for specific orders by order ID.

### Class diagram



**Explanation:**

UML Relationships:

**Association**

1. The relationship between the Book and CartItem classes is an association relationship. This means that a CartItem is associated with a Book. Specifically, a CartItem contains information about a Book, such as its ID, title, author, price, and quantity.
2. The relationship between the Book and BookBuddy classes is an association relationship. The BookBuddy class manages a collection of Book objects. Specifically, the BookBuddy class contains an ArrayListADT<Book> to store and manage the list of books. (same relationship for CartItem and CartItemBuddy, Order and OrderBuddy)

**Composition**

the relationship between the Order and CartItem classes is a composition relationship.

Order: The Order class has an attribute items which is an ArrayListADT<CartItem>. This means that an Order is composed of multiple CartItem objects.

CartItem: The CartItem class represents an item in the order.

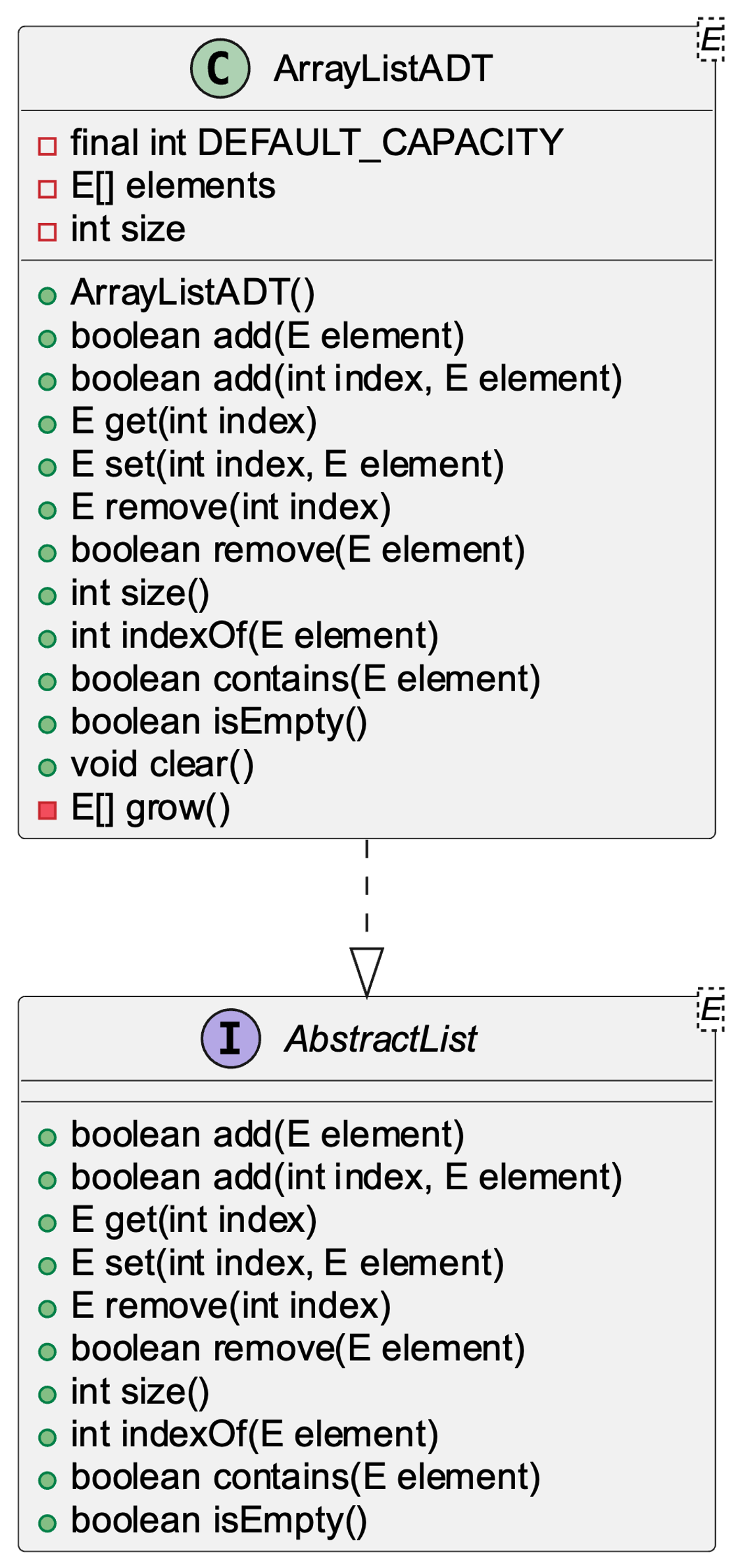
**Inheritance**

customer and admin inherit from user class

attributes methods

**Realization**

The relationship between ArrayListADT and AbstractList is an implementation relationship. The ArrayListADT class implements the AbstractList interface. This means that ArrayListADT provides concrete implementations for the methods defined in the AbstractList interface. (same relationship for QueueADT and AbstractQueue, StackADT and AbstractStack)



### Code

Folder structure:

phamthanhtruc@m BooksApp % tree

.

├── Main.java

├── admin

│ ├── Admin.java

│ └── AdminBuddy.java

├── adt

│ ├── AbstractList.java

│ ├── AbstractQueue.java

│ ├── AbstractStack.java

│ ├── ArrayListADT.java

│ ├── QueueADT.java

│ └── StackADT.java

├── algo

│ ├── BinarySearch.java

│ ├── LinearSearch.java

│ └── QuickSort.java

├── customer

│ ├── Customer.java

│ └── CustomerBuddy.java

└── models

├── Book.java

├── BookBuddy.java

├── CartItem.java

├── CartItemBuddy.java

├── Order.java

├── OrderBuddy.java

└── User.java

6 directories, 21 files

Code files:

*// FILE: Main.java*

package **BooksApp**;

import **java**.**util**.**Scanner**;

import **java**.**util**.**Date**;

import **java**.**io**.**Console**;

import **BooksApp**.**models**.**BookBuddy**;

import **BooksApp**.**models**.**CartItemBuddy**;

import **BooksApp**.**models**.**OrderBuddy**;

import **BooksApp**.**models**.**User**;

import **BooksApp**.**admin**.**Admin**;

import **BooksApp**.**admin**.**AdminBuddy**;

import **BooksApp**.**customer**.**Customer**;

import **BooksApp**.**customer**.**CustomerBuddy**;

import **BooksApp**.**adt**.**StackADT**;

import **BooksApp**.**algo**.**BinarySearch**;

import **BooksApp**.**algo**.**LinearSearch**;

import **BooksApp**.**algo**.**QuickSort**;

import **BooksApp**.**models**.**Book**;

import **BooksApp**.**adt**.**ArrayListADT**;

import **BooksApp**.**models**.**CartItem**;

import **BooksApp**.**models**.**Order**;

import **BooksApp**.**adt**.**QueueADT**;

public class **Main** {

public static void **main**(**String**[] args) {

**Scanner** scanner = new **Scanner**(**System**.in);

**Console** console = **System**.**console**(); *// Get the system console*

**BookBuddy** bookBuddy = new **BookBuddy**();

**CartItemBuddy** cartItemBuddy = new **CartItemBuddy**();

**OrderBuddy** orderBuddy = new **OrderBuddy**();

**AdminBuddy** adminBuddy = new **AdminBuddy**();

**CustomerBuddy** customerBuddy = new **CustomerBuddy**();

**StackADT**<**String**> searchHistory = new **StackADT**<>(); *// Stack to store search queries*

boolean running = true;

**User** loggedInUser = null;

*// ArrayListADT to simulate database*

**ArrayListADT**<**User**> usersDatabase = new **ArrayListADT**<>();

while (running) {

**System**.out.**println**("----------------------");

**System**.out.**println**("Book Store Application");

**System**.out.**println**("----------------------");

if (loggedInUser == null) {

**System**.out.**println**("1. Register");

**System**.out.**println**("2. Login");

**System**.out.**println**("3. Exit");

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter your choice: ");

int choice = **Integer**.**parseInt**(scanner.**nextLine**());

switch (choice) {

case 1:

*// Register*

**System**.out.**println**("1. Register as Customer");

**System**.out.**println**("2. Register as Admin");

**System**.out.**print**("Enter your choice: ");

int registerChoice = **Integer**.**parseInt**(scanner.**nextLine**());

**System**.out.**print**("Enter your name: ");

**String** name = scanner.**nextLine**();

**System**.out.**print**("Enter your email: ");

**String** email = scanner.**nextLine**();

**String** password;

if (console != null) {

*// Use Console.readPassword() to read the password without echoing*

char[] passwordArray = console.**readPassword**("Enter your password: ");

password = new **String**(passwordArray);

} else {

*// Fallback to Scanner if Console is not available (e.g., in some IDEs)*

**System**.out.**print**("Enter your password: ");

password = scanner.**nextLine**();

}

if (registerChoice == 1) {

**System**.out.**print**("Enter your address: ");

**String** address = scanner.**nextLine**();

**Customer** customer = customerBuddy.**createCustomer**(name, email, password, address);

customerBuddy.**registerCustomer**(customer);

usersDatabase.**add**(customer); *// Save to database*

**System**.out.**println**("Customer registered successfully!");

} else if (registerChoice == 2) {

**System**.out.**print**("Enter the admin key: ");

**String** adminKey = scanner.**nextLine**();

if (adminKey.**equals**("abc")) {

**Admin** admin = adminBuddy.**createAdmin**(name, email, password);

adminBuddy.**registerAdmin**(admin);

usersDatabase.**add**(admin); *// Save to database*

**System**.out.**println**("Admin registered successfully!");

} else {

**System**.out.**println**("Invalid admin key. Registration as Admin failed.");

}

} else {

**System**.out.**println**("Invalid choice.");

}

break;

case 2:

*// Login*

**System**.out.**println**("1. Login as Customer");

**System**.out.**println**("2. Login as Admin");

**System**.out.**print**("Enter your choice: ");

int loginChoice = **Integer**.**parseInt**(scanner.**nextLine**());

**System**.out.**print**("Enter your email: ");

**String** loginEmail = scanner.**nextLine**();

**String** loginPassword;

if (console != null) {

*// Use Console.readPassword() to read the password without echoing*

char[] passwordArray = console.**readPassword**("Enter your password: ");

loginPassword = new **String**(passwordArray);

} else {

*// Fallback to Scanner if Console is not available (e.g., in some IDEs)*

**System**.out.**print**("Enter your password: ");

loginPassword = scanner.**nextLine**();

}

if (loginChoice == 1) {

loggedInUser = customerBuddy.**loginCustomer**(loginEmail, loginPassword);

} else if (loginChoice == 2) {

loggedInUser = adminBuddy.**loginAdmin**(loginEmail, loginPassword);

} else {

**System**.out.**println**("Invalid choice.");

}

if (loggedInUser != null) {

**System**.out.**println**("Login successful! Welcome, " + loggedInUser.**getName**());

} else {

**System**.out.**println**("Invalid email or password.");

}

break;

case 3:

*// Exit*

running = false;

break;

default:

**System**.out.**println**("Invalid choice. Please enter your choice again!");

break;

}

} else {

if (loggedInUser instanceof **Customer**) {

*// Customer menu*

int choice = 0;

while (true) {

**System**.out.**println**("1. Display all books");

**System**.out.**println**("2. Add book to cart");

**System**.out.**println**("3. Search book");

**System**.out.**println**("4. Go to cart");

**System**.out.**println**("5. Orders History");

**System**.out.**println**("6. Search book history");

**System**.out.**println**("7. Logout");

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter your choice: ");

try {

choice = **Integer**.**parseInt**(scanner.**nextLine**());

if (choice >= 1 && choice <= 7) {

break;

} else {

**System**.out.**println**("Invalid choice. Please enter a number between 1 and 7.");

}

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a number between 1 and 7.");

}

}

switch (choice) {

case 1:

*// Display all books*

bookBuddy.**displayBooks**();

break;

case 2:

*// Add book to cart*

**System**.out.**println**("----------------------");

int bookIdToAdd = 0;

while (true) {

try {

**System**.out.**print**("Enter the book ID to add to cart: ");

bookIdToAdd = **Integer**.**parseInt**(scanner.**nextLine**());

break;

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid book ID.");

}

}

**Book** book = bookBuddy.**getBookById**(bookIdToAdd); *// Assume this method retrieves the book by*

*// ID*

if (book != null) {

int currentQuantityInCart = cartItemBuddy.**getQuantityInCart**(bookIdToAdd);

int availableQuantity = book.**getStockQuantity**() - currentQuantityInCart;

if (availableQuantity <= 0) {

**System**.out.**println**("This book is out of stock.");

} else {

**System**.out.**print**("Enter the quantity to add to cart (available: "

+ availableQuantity + "): ");

int quantity = 0;

boolean validQuantity = false;

while (!validQuantity) {

try {

quantity = **Integer**.**parseInt**(scanner.**nextLine**());

if (quantity > 0 && quantity <= availableQuantity) {

validQuantity = true;

} else {

**System**.out.**println**(

"Invalid quantity. Please enter a quantity between 1 and "

+ availableQuantity + ".");

}

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid quantity.");

}

}

cartItemBuddy.**addToCart**(book, quantity); *// Add to cart with specified quantity*

**System**.out.**println**("Book(s) added to cart!");

}

} else {

**System**.out.**println**("No book found with ID: " + bookIdToAdd);

}

break;

case 3:

*// Search book*

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the ID, title or author of the book: ");

**String** searchQuery = scanner.**nextLine**();

searchHistory.**push**(searchQuery); *// Save search query to history*

bookBuddy.**searchBook**(searchQuery); *// Assume searchBook handles searching by ID, title, or*

*// author*

break;

case 4:

*// View cart*

**System**.out.**println**("----------------------");

cartItemBuddy.**viewCart**();

**System**.out.**println**("1. Place Order");

**System**.out.**println**("2. Go back");

**System**.out.**println**("3. Remove book from cart");

**System**.out.**println**("4. Change quantity");

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter your choice: ");

int choiceCart = **Integer**.**parseInt**(scanner.**nextLine**());

switch (choiceCart) {

case 1:

*// Place Order*

if (cartItemBuddy.**isEmpty**()) {

**System**.out.**println**(

"Your cart is empty. Add some books to the cart before checking out.");

} else {

**Customer** customer = (**Customer**) loggedInUser;

**ArrayListADT**<**CartItem**> booksInCart = new **ArrayListADT**<>();

for (int i = 0; i < cartItemBuddy.**getBooksInCart**().**size**(); i++) {

booksInCart.**add**(cartItemBuddy.**getBooksInCart**().**get**(i));

}

**Order** order = orderBuddy.**createOrder**(new **Date**(), customer, "Pending",

booksInCart);

orderBuddy.**addOrder**(order);

*// Subtract stock quantity of the books*

for (int i = 0; i < booksInCart.**size**(); i++) {

**CartItem** cartItem = booksInCart.**get**(i);

**Book** bookToBuy = bookBuddy.**getBookById**(cartItem.**getBookId**());

bookToBuy.**setStockQuantity**(

bookToBuy.**getStockQuantity**() - cartItem.**getQuantity**());

}

**System**.out.**println**("----------------------");

**System**.out.**println**("Order placed successfully!");

**System**.out.**println**(order);

*// Clear the cart after Place Order*

cartItemBuddy.**clearCart**();

}

break;

case 2:

*// Go back*

break;

case 3:

*// Remove book from cart*

cartItemBuddy.**removeBookFromCart**(scanner);

break;

case 4:

*// Change quantity*

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the book ID to change quantity or 'n' to cancel: ");

**String** inputCart2 = scanner.**nextLine**();

if (!inputCart2.**equalsIgnoreCase**("n")) {

try {

int bookId = **Integer**.**parseInt**(inputCart2);

**CartItem** cartItem = cartItemBuddy.**getCartItemById**(bookId);

if (cartItem != null) {

**Book** bookQuantityChange = bookBuddy.**getBookById**(bookId); *// Assume this method retrieves the book by* *ID*

if (bookQuantityChange != null) {

**System**.out.**print**("Enter the new quantity (available: "

+ bookQuantityChange.**getStockQuantity**() + "): ");

int newQuantity = 0;

boolean validQuantity = false;

while (!validQuantity) {

try {

newQuantity = **Integer**.**parseInt**(scanner.**nextLine**());

if (newQuantity > 0 && newQuantity <= bookQuantityChange

.**getStockQuantity**()) {

validQuantity = true;

} else {

**System**.out.**println**(

"Invalid quantity. Please enter a quantity between 1 and "

+ bookQuantityChange.**getStockQuantity**()

+ ".");

}

} catch (**NumberFormatException** e) {

**System**.out.**println**(

"Invalid input. Please enter a valid quantity.");

}

}

cartItem.**setQuantity**(newQuantity);

**System**.out.**println**("Quantity updated successfully!");

} else {

**System**.out.**println**("No book found with ID: " + bookId);

}

} else {

**System**.out.**println**("No book found with ID: " + bookId);

}

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid book ID.");

}

}

break;

}

break;

case 5:

*// View Orders History as customer*

orderBuddy.**viewOrdersHistory**(loggedInUser);

*// Search Orders as customer*

orderBuddy.**searchOrdersAsCustomer**(loggedInUser, scanner);

break;

case 6:

*// Search book history*

**System**.out.**println**("----------------------");

**System**.out.**println**("Search History:");

if (searchHistory.**isEmpty**()) {

**System**.out.**println**("No search history found.");

} else {

**System**.out.**println**(searchHistory.**toString**());

}

break;

case 7:

*// Logout*

loggedInUser = null;

**System**.out.**println**("Logged out successfully.");

break;

default:

**System**.out.**println**("Invalid choice. Please enter your choice again!");

break;

}

} else if (loggedInUser instanceof **Admin**) {

*// Admin menu*

**System**.out.**println**("1. Display all books");

**System**.out.**println**("2. Add new book");

**System**.out.**println**("3. Update book details");

**System**.out.**println**("4. Remove book");

**System**.out.**println**("5. View all orders");

**System**.out.**println**("6. Logout");

**System**.out.**println**("----------------------");

int choice = 0;

while (true) {

try {

**System**.out.**print**("Enter your choice: ");

choice = **Integer**.**parseInt**(scanner.**nextLine**());

if (choice < 1 || choice > 6) {

throw new **NumberFormatException**();

}

break;

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid choice. Please enter a number between 1 and 6.");

}

}

switch (choice) {

case 1:

*// Display all books*

bookBuddy.**displayBooks**();

break;

case 2:

*// Add new book*

bookBuddy.**addNewBook**(scanner);

break;

case 3:

*// Update book details*

bookBuddy.**updateBookDetails**(scanner);

break;

case 4:

*// Remove book*

bookBuddy.**removeBook**(scanner);

break;

case 5:

*// View all orders as admin*

orderBuddy.**viewAllOrdersAsAdmin**(scanner);

break;

case 6:

*// Logout*

loggedInUser = null;

**System**.out.**println**("Logged out successfully.");

break;

default:

**System**.out.**println**("Invalid choice. Please enter your choice again!");

break;

}

}

}

}

scanner.**close**();

**System**.out.**println**("Exiting ...");

}

}

*// FILE: adt/AbstractList.java*

package **BooksApp**.**adt**;

*// Interface for a generic list*

public interface **AbstractList**<**E**> {

*// Adds an element to the end of the list*

boolean **add**(**E** element);

*// Adds an element at a specific index in the list*

boolean **add**(int index, **E** element);

*// Retrieves the element at a specific index*

**E** **get**(int index);

*// Replaces the element at a specific index with a new element*

**E** **set**(int index, **E** element);

*// Removes the element at a specific index*

**E** **remove**(int index);

*// Removes the first occurrence of a specific element*

boolean **remove**(**E** element); *// New method to remove an element by its value*

*// Returns the number of elements in the list*

int **size**();

*// Returns the index of the first occurrence of a specific element*

int **indexOf**(**E** element);

*// Checks if the list contains a specific element*

boolean **contains**(**E** element);

*// Checks if the list is empty*

boolean **isEmpty**();

}

*// FILE: adt/ArrayListADT.java*

package **BooksApp**.**adt**;

import **java**.**util**.**Arrays**;

import **java**.**util**.**NoSuchElementException**;

public class **ArrayListADT**<**E**> implements **AbstractList**<**E**> {

*// Default capacity of the array*

private final int DEFAULT\_CAPACITY = 10;

*// Array to store elements*

private **E**[] elements;

*// Number of elements in the list*

private int size;

*// Constructor to initialize the array and size*

public **ArrayListADT**() {

this.elements = (**E**[]) new **Object**[DEFAULT\_CAPACITY];

this.size = 0;

}

*// Adds an element to the end of the list*

@**Override**

public boolean **add**(**E** element) {

*// Check if the array needs to be resized*

if (this.size == this.elements.length) {

this.elements = **grow**();

}

*// Add the new element*

this.elements[size] = element;

this.size++;

return true;

}

*// Resizes the array to double its current size*

private **E**[] **grow**() {

return **Arrays**.**copyOf**(this.elements, this.elements.length \* 2);

}

*// Adds an element at a specific index in the list*

@**Override**

public boolean **add**(int index, **E** element) {

*// Check if the index is valid*

if (index < 0 || index > this.**size**()) {

throw new **IndexOutOfBoundsException**();

}

*// Check if the array needs to be resized*

if (this.**size**() == this.elements.length) {

this.elements = **grow**();

}

*// Shift the remaining elements to the right*

for (int i = size; i >= index + 1; i--) {

elements[i] = elements[i - 1];

}

*// Add the new element*

this.elements[index] = element;

this.size++;

return true;

}

*// Retrieves the element at a specific index*

@**Override**

public **E** **get**(int index) {

*// Check if the index is valid*

if (index < 0 || index >= this.**size**()) {

throw new **IndexOutOfBoundsException**();

}

return this.elements[index];

}

*// Replaces the element at a specific index with a new element*

@**Override**

public **E** **set**(int index, **E** element) {

**E** oldElement = this.**get**(index);

this.elements[index] = element;

return oldElement;

}

*// Removes the element at a specific index*

@**Override**

public **E** **remove**(int index) {

**E** removedElement = this.**get**(index);

*// Shift the remaining elements to the left*

for (int i = index; i < this.**size**() - 1; i++) {

elements[i] = elements[i + 1];

}

this.size--;

this.elements[size] = null;

return removedElement;

}

*// Removes the first occurrence of a specific element*

public boolean **remove**(**E** element) {

int index = this.**indexOf**(element);

if (index != -1) {

this.**remove**(index);

return true;

}

return false;

}

*// Returns the number of elements in the list*

@**Override**

public int **size**() {

return this.size;

}

*// Returns the index of the first occurrence of a specific element*

@**Override**

public int **indexOf**(**E** element) {

for (int i = 0; i < this.**size**(); i++) {

if (this.elements[i] == element) {

return i;

}

}

return -1;

}

*// Checks if the list contains a specific element*

@**Override**

public boolean **contains**(**E** element) {

for (int i = 0; i < this.**size**(); i++) {

if (this.elements[i] == element) {

return true;

}

}

return false;

}

*// Checks if the list is empty*

@**Override**

public boolean **isEmpty**() {

return this.**size**() == 0;

}

*// Returns a string representation of the list*

@**Override**

public **String** **toString**() {

**StringBuilder** result = new **StringBuilder**();

result.**append**("[");

for (int i = 0; i < this.size; i++) {

result.**append**(this.**get**(i));

if (i < this.size - 1) {

result.**append**(", ");

}

}

result.**append**("]");

return result.**toString**();

}

*// Clears the list*

public void **clear**() {

for (int i = 0; i < this.size; i++) {

this.elements[i] = null;

}

this.size = 0;

}

}

*// FILE: adt/AbstractQueue.java*

package **BooksApp**.**adt**;

public interface **AbstractQueue**<**E**> {

*// Inserts the specified element into the queue*

void **offer**(**E** element);

*// Retrieves and removes the head of this queue, or returns null if this queue is empty*

**E** **poll**();

*// Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty*

**E** **peek**();

*// Returns the number of elements in this queue*

int **size**();

*// Returns true if this queue contains no elements*

boolean **isEmpty**();

}

*// FILE: adt/QueueADT.java*

package **BooksApp**.**adt**;

*// Implementation of a generic queue*

public class **QueueADT**<**E**> implements **AbstractQueue**<**E**> {

*// Inner class to represent a node in the queue*

private class **Node**<**E**> {

private **E** element; *// The element stored in the node*

private **Node**<**E**> next; *// The next node in the queue*

*// Constructor to initialize the node with an element*

private **Node**(**E** element) {

this.element = element;

this.next = null;

}

}

private **Node**<**E**> head; *// The head of the queue*

private int size; *// The number of elements in the queue*

*// Constructor to initialize the queue*

public **QueueADT**() {

this.head = null;

this.size = 0;

}

*// Inserts the specified element into the queue*

@**Override**

public void **offer**(**E** element) {

**Node**<**E**> newNode = new **Node**<>(element);

if (this.**isEmpty**()) {

this.head = newNode;

} else {

**Node**<**E**> tempNode = this.head;

while (tempNode.next != null) {

tempNode = tempNode.next;

}

tempNode.next = newNode;

}

this.size++;

}

*// Retrieves and removes the head of this queue, or returns null if this queue is empty*

@**Override**

public **E** **poll**() {

if (this.**isEmpty**()) {

throw new **IllegalStateException**("Queue is currently empty.");

}

**E** oldElement = this.head.element;

**Node**<**E**> tempNode = head;

this.head = this.head.next;

tempNode.next = null;

this.size--;

return oldElement;

}

*// Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty*

@**Override**

public **E** **peek**() {

if (this.**isEmpty**()) {

throw new **IllegalStateException**("Queue is currently empty.");

}

return this.head.element;

}

*// Returns the number of elements in this queue*

@**Override**

public int **size**() {

return this.size;

}

*// Returns true if this queue contains no elements*

@**Override**

public boolean **isEmpty**() {

return this.head == null;

}

*// Returns a string representation of the queue*

@**Override**

public **String** **toString**() {

**StringBuilder** result = new **StringBuilder**("[");

**Node**<**E**> tempNode = head;

while (tempNode != null) {

result.**append**(tempNode.element);

if (tempNode.next != null) {

result.**append**(", ");

}

tempNode = tempNode.next;

}

result.**append**("]");

return result.**toString**();

}

}

*// Runner class to test the QueueADT*

class **QueueADTRunner** {

public static void **main**(**String**[] args) {

*// Main method for testing*

}

}

*// FILE: adt/AbstractStack.java*

package **BooksApp**.**adt**;

public interface **AbstractStack**<**E**> {

*// Pushes an element onto the top of the stack*

void **push**(**E** element);

*// Removes and returns the top element from the stack*

**E** **pop**();

*// Returns the top element of the stack without removing it*

**E** **peek**();

*// Returns the number of elements in the stack*

int **size**();

*// Checks if the stack is empty*

boolean **isEmpty**();

}

*// FILE: adt/StackADT.java*

package **BooksApp**.**adt**;

import **java**.**util**.**NoSuchElementException**;

*// Implementation of a generic stack*

public class **StackADT**<**E**> implements **AbstractStack**<**E**> {

*// Inner class to represent a node in the stack*

private class **Node**<**E**> {

private **E** element; *// The element stored in the node*

private **Node**<**E**> next; *// The next node in the stack*

*// Constructor to initialize the node with an element*

private **Node**(**E** element) {

this.element = element;

this.next = null;

}

}

private int size; *// The number of elements in the stack*

private **Node**<**E**> top; *// The top of the stack*

*// Constructor to initialize the stack*

public **StackADT**() {

this.top = null;

this.size = 0;

}

*// Pushes an element onto the top of the stack*

@**Override**

public void **push**(**E** element) {

**Node**<**E**> newNode = new **Node**<>(element);

newNode.next = top;

this.top = newNode;

this.size++;

}

*// Removes and returns the top element from the stack*

@**Override**

public **E** **pop**() {

if (this.**isEmpty**()) {

throw new **NoSuchElementException**("Stack is empty.");

}

**E** oldElement = this.top.element;

*// Option 1*

*/\*Node<E> tempNode = top;*

*this.top = this.top.next;*

*tempNode.next = null;*

*this.size--;\*/*

*// Option 2*

**Node**<**E**> tempNode = top.next;

this.top.next = null;

this.top = tempNode;

this.size--;

return oldElement;

}

*// Returns the top element of the stack without removing it*

@**Override**

public **E** **peek**() {

if (this.**isEmpty**()) {

throw new **NoSuchElementException**("Stack is empty.");

}

return this.top.element;

}

*// Returns the number of elements in the stack*

@**Override**

public int **size**() {

return this.size;

}

*// Checks if the stack is empty*

@**Override**

public boolean **isEmpty**() {

return this.size == 0;

}

*// Returns a string representation of the stack*

@**Override**

public **String** **toString**() {

**StringBuilder** result = new **StringBuilder**("[");

**Node**<**E**> tempNode = top;

while (tempNode != null) {

result.**append**(tempNode.element);

if (tempNode.next != null) {

result.**append**(", ");

}

tempNode = tempNode.next;

}

result.**append**("]");

return result.**toString**();

}

}

*// Runner class to test the StackADT*

class **StackADTRunner** {

public static void **main**(**String**[] args) {

*// Main method for testing*

}

}

*// FILE: algo/BinarySearch.java*

package **BooksApp**.**algo**;

import **BooksApp**.**adt**.**ArrayListADT**;

public class **BinarySearch**<**T** extends **Comparable**<**T**>> {

*// Public method to initiate binary search on the list*

public int **search**(**ArrayListADT**<**T**> list, **T** key) {

return **search**(list, key, 0, list.**size**() - 1);

}

*// Private recursive method to perform binary search*

private int **search**(**ArrayListADT**<**T**> list, **T** key, int start, int end) {

if (start > end) {

return -1; *// Key not found*

}

int mid = start + (end - start) / 2; *// Calculate mid index*

**T** midVal = list.**get**(mid); *// Get value at mid index*

if (midVal.**compareTo**(key) == 0) {

return mid; *// Key found*

} else if (midVal.**compareTo**(key) > 0) {

return **search**(list, key, start, mid - 1); *// Search left half*

} else {

return **search**(list, key, mid + 1, end); *// Search right half*

}

}

}

*// FILE: algo/LinearSearch.java*

package **BooksApp**.**algo**;

import **BooksApp**.**adt**.**ArrayListADT**;

public class **LinearSearch**<**T**> {

*// Searches for an element that exactly matches the key using the equals method.*

*// Example: Finding a book with a specific ID.*

public int **search**(**ArrayListADT**<**T**> list, **T** key) {

for (int i = 0; i < list.**size**(); i++) {

*// Check if the current element equals the key*

if (list.**get**(i).**equals**(key)) {

return i; *// Key found*

}

}

return -1; *// Key not found*

}

*// Searches for an element that matches a custom condition defined by the predicate.*

*// Example: Finding a book with a title that contains a specific word.*

public int **search**(**ArrayListADT**<**T**> list, **T** key, **java**.**util**.**function**.**Predicate**<**T**> predicate) {

for (int i = 0; i < list.**size**(); i++) {

*// Check if the current element matches the predicate condition*

if (predicate.**test**(list.**get**(i))) {

return i; *// Key found*

}

}

return -1; *// Key not found*

}

}

*// FILE: algo/QuickSort.java*

package **BooksApp**.**algo**;

import **BooksApp**.**adt**.**ArrayListADT**;

public class **QuickSort**<**T** extends **Comparable**<**T**>> {

*// Public method to initiate the quicksort on the list*

public void **sort**(**ArrayListADT**<**T**> list) {

**sort**(list, 0, list.**size**() - 1);

}

*// Private recursive method to perform quicksort*

private void **sort**(**ArrayListADT**<**T**> list, int start, int end) {

*// Base condition to end recursion*

if (start >= end)

return;

*// Partition the list and get the boundary index*

int boundary = **partition**(list, start, end);

*// Sort the left part of the list*

**sort**(list, start, boundary - 1);

*// Sort the right part of the list*

**sort**(list, boundary + 1, end);

}

*// Private method to partition the list*

private int **partition**(**ArrayListADT**<**T**> list, int start, int end) {

**T** pivot = list.**get**(end); *// Choose the pivot element*

int boundary = start - 1; *// Initialize the boundary index*

for (int i = start; i <= end; i++) {

*// Compare current element with the pivot*

if (list.**get**(i).**compareTo**(pivot) <= 0) {

boundary++;

*// Swap two elements*

**T** temp = list.**get**(i);

list.**set**(i, list.**get**(boundary));

list.**set**(boundary, temp);

}

}

return boundary; *// Return the boundary index*

}

}

*// FILE: models/User.java*

package **BooksApp**.**models**;

public abstract class **User** {

protected int userId; *// Unique identifier for the user*

protected **String** name; *// Name of the user*

protected **String** email; *// Email of the user*

protected **String** password; *// Password of the user*

*// Constructor to initialize the user object*

public **User**(int userId, **String** name, **String** email, **String** password) {

this.userId = userId;

this.name = name;

this.email = email;

this.password = password;

}

*// Getter method for userId*

public int **getUserId**() {

return userId;

}

*// Getter method for name*

public **String** **getName**() {

return name;

}

*// Getter method for email*

public **String** **getEmail**() {

return email;

}

*// Getter method for password*

public **String** **getPassword**() {

return password;

}

*// Abstract method to get the user type*

public abstract **String** **getUserType**();

}

*// FILE: customer/Customer.java*

package **BooksApp**.**customer**;

import **BooksApp**.**models**.**User**;

public class **Customer** extends **User** {

private **String** address; *// Address of the customer*

*// Constructor to initialize the customer object*

public **Customer**(int userId, **String** name, **String** email, **String** password, **String** address) {

super(userId, name, email, password); *// Call the superclass constructor*

this.address = address; *// Set the address*

}

*// Getter method to retrieve the address*

public **String** **getAddress**() {

return address;

}

*// Override method to return the user type*

@**Override**

public **String** **getUserType**() {

return "Customer";

}

*// Override method to return a string representation of the customer object*

@**Override**

public **String** **toString**() {

return "Customer{" +

"userId=" + userId +

", name='" + name + '\'' +

", email='" + email + '\'' +

", address='" + address + '\'' +

'}';

}

}

*// FILE: customer/CustomerBuddy.java*

package **BooksApp**.**customer**;

import **BooksApp**.**adt**.**ArrayListADT**;

public class **CustomerBuddy** {

*// List to store customer objects*

private **ArrayListADT**<**Customer**> customers;

*// Static counter to generate unique customer IDs*

private static int customerIdCounter = 1;

*// Constructor to initialize the customer list*

public **CustomerBuddy**() {

customers = new **ArrayListADT**<>();

}

*// Method to register a new customer*

public void **registerCustomer**(**Customer** customer) {

customers.**add**(customer);

}

*// Method to login a customer by email and password*

public **Customer** **loginCustomer**(**String** email, **String** password) {

*// Iterate through the customer list*

for (int i = 0; i < customers.**size**(); i++) {

**Customer** customer = customers.**get**(i);

*// Check if the email and password match*

if (customer.**getEmail**().**equals**(email) && customer.**getPassword**().**equals**(password)) {

return customer; *// Customer found*

}

}

return null; *// Customer not found*

}

*// Static method to generate a unique customer ID*

private static int **generateCustomerId**() {

return customerIdCounter++;

}

*// Method to create a new customer with the given details*

public **Customer** **createCustomer**(**String** name, **String** email, **String** password, **String** address) {

int customerId = **generateCustomerId**();

return new **Customer**(customerId, name, email, password, address);

}

}

*// FILE: admin/Admin.java*

package **BooksApp**.**admin**;

import **BooksApp**.**models**.**User**;

public class **Admin** extends **User** {

*// Constructor for the Admin class, which calls the superclass (User) constructor*

public **Admin**(int userId, **String** name, **String** email, **String** password) {

super(userId, name, email, password);

}

*// Overrides the getUserType method from the User class to return "Admin"*

@**Override**

public **String** **getUserType**() {

return "Admin";

}

}

*// FILE: admin/AdminBuddy.java*

package **BooksApp**.**admin**;

import **BooksApp**.**adt**.**ArrayListADT**;

import **BooksApp**.**customer**.**Customer**;

public class **AdminBuddy** {

*// List to store admin objects*

private **ArrayListADT**<**Admin**> admins;

*// Static counter to generate unique admin IDs*

private static int adminIdCounter = 1;

*// Constructor to initialize the admin list*

public **AdminBuddy**() {

admins = new **ArrayListADT**<>();

}

*// Method to register a new admin*

public void **registerAdmin**(**Admin** admin) {

admins.**add**(admin);

}

*// Method to login an admin by email and password*

public **Admin** **loginAdmin**(**String** email, **String** password) {

for (int i = 0; i < admins.**size**(); i++) {

**Admin** admin = admins.**get**(i);

if (admin.**getEmail**().**equals**(email) && admin.**getPassword**().**equals**(password)) {

return admin;

}

}

return null;

}

*// Static method to generate a unique admin ID*

private static int **generateAdminId**() {

return adminIdCounter++;

}

*// Method to create a new admin with the given details*

public **Admin** **createAdmin**(**String** name, **String** email, **String** password) {

int adminId = **generateAdminId**();

return new **Admin**(adminId, name, email, password);

}

}

*// FILE: models/Book.java*

package **BooksApp**.**models**;

public class **Book** implements **Comparable**<**Book**> {

private int bookId; *// Unique identifier for the book*

private **String** title; *// Title of the book*

private **String** author; *// Author of the book*

private double price; *// Price of the book*

private int stockQuantity; *// Quantity of the book in stock*

*// Constructor to initialize the book object*

public **Book**(int bookId, **String** title, **String** author, double price, int stockQuantity) {

this.bookId = bookId;

this.title = title;

this.author = author;

this.price = price;

this.stockQuantity = stockQuantity;

}

*// Getter method for bookId*

public int **getBookId**() {

return bookId;

}

*// Getter method for title*

public **String** **getTitle**() {

return title;

}

*// Setter method for title*

public void **setTitle**(**String** title) {

this.title = title;

}

*// Getter method for author*

public **String** **getAuthor**() {

return author;

}

*// Setter method for author*

public void **setAuthor**(**String** author) {

this.author = author;

}

*// Getter method for price*

public double **getPrice**() {

return price;

}

*// Setter method for price*

public void **setPrice**(double price) {

this.price = price;

}

*// Getter method for stockQuantity*

public int **getStockQuantity**() {

return stockQuantity;

}

*// Setter method for stockQuantity*

public void **setStockQuantity**(int stockQuantity) {

this.stockQuantity = stockQuantity;

}

*// Override compareTo method to compare books by bookId*

@**Override**

public int **compareTo**(**Book** other) {

return **Integer**.**compare**(this.bookId, other.bookId);

}

*// Override toString method to return a string representation of the book object*

@**Override**

public **String** **toString**() {

return **String**.**format**("Book ID: %d\tTitle: %s\tAuthor: %s\tPrice: %.2f\tStock Quantity: %d",

bookId, title, author, price, stockQuantity);

}

}

*// FILE: models/BookBuddy.java*

package **BooksApp**.**models**;

import **BooksApp**.**adt**.**ArrayListADT**;

import **BooksApp**.**algo**.**QuickSort**;

import **BooksApp**.**algo**.**BinarySearch**;

import **BooksApp**.**algo**.**LinearSearch**;

import **java**.**util**.**Scanner**;

public class **BookBuddy** {

private **ArrayListADT**<**Book**> books; *// List to store book objects*

private int maxBookId; *// Variable to keep track of the maximum book ID*

public **BookBuddy**() {

*// Initialize the books list with some sample data*

this.books = new **ArrayListADT**<>();

books.**add**(new **Book**(1, "The Catcher in the Rye", "J.D. Salinger", 10.99, 5));

books.**add**(new **Book**(2, "Brave New World", "Aldous Huxley", 9.99, 3));

books.**add**(new **Book**(3, "To Kill a Mockingbird", "Harper Lee", 12.99, 7));

books.**add**(new **Book**(4, "The Great Gatsby", "F. Scott Fitzgerald", 7.99, 2));

*// Initialize maxBookId to the highest ID in the initial list*

this.maxBookId = books.**size**();

}

*// Method to display books*

public void **displayBooks**() {

if (books.**isEmpty**()) {

**System**.out.**println**("No books to display.");

} else {

*// Sort books using QuickSort*

**QuickSort**<**Book**> sorter = new **QuickSort**<>();

sorter.**sort**(books);

*// Display sorted books*

**System**.out.**println**("Book List:");

for (int i = 0; i < books.**size**(); i++) {

**Book** currentBook = books.**get**(i);

**System**.out.**println**(currentBook);

}

}

}

*// Method to search for books by ID, title, or author*

public void **searchBook**(**String** query) {

**System**.out.**println**("Searching for books matching: " + query);

boolean found = false;

*// Sort books using QuickSort*

**QuickSort**<**Book**> sorter = new **QuickSort**<>();

sorter.**sort**(books);

*// Search by ID using BinarySearch*

try {

int bookId = **Integer**.**parseInt**(query);

**BinarySearch**<**Book**> binarySearch = new **BinarySearch**<>();

int index = binarySearch.**search**(books, new **Book**(bookId, "", "", 0, 0));

if (index != -1) {

**System**.out.**println**(books.**get**(index));

found = true;

}

} catch (**NumberFormatException** e) {

*// If query is not an integer, continue to search by title or author*

}

*// Search by title or author using LinearSearch*

if (!found) {

**LinearSearch**<**Book**> linearSearch = new **LinearSearch**<>();

int index = linearSearch.**search**(books, null,

book -> book.**getTitle**().**toLowerCase**().**contains**(query.**toLowerCase**()) ||

book.**getAuthor**().**toLowerCase**().**contains**(query.**toLowerCase**()));

if (index != -1) {

**System**.out.**println**(books.**get**(index));

found = true;

}

}

if (!found) {

**System**.out.**println**("No books found matching: " + query);

}

}

*// Method to display a book's details by its ID*

public void **displayBook**(int bookId) {

**Book** book = **getBookById**(bookId);

if (book != null) {

**System**.out.**println**(book);

} else {

**System**.out.**println**("No book found with ID: " + bookId);

}

}

*// Method to get a book by its ID*

public **Book** **getBookById**(int bookId) {

for (int i = 0; i < books.**size**(); i++) {

**Book** book = books.**get**(i);

if (book.**getBookId**() == bookId) {

return book;

}

}

return null; *// If no book found, return null*

}

*// Method to update book details*

public void **updateBookDetails**(**Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the book ID to update: ");

int bookIdToUpdate = **Integer**.**parseInt**(scanner.**nextLine**());

**Book** bookToUpdate = **getBookById**(bookIdToUpdate);

if (bookToUpdate != null) {

**System**.out.**print**("Enter the new title (leave blank to keep current): ");

**String** newTitle = scanner.**nextLine**();

**System**.out.**print**("Enter the new author (leave blank to keep current): ");

**String** newAuthor = scanner.**nextLine**();

**System**.out.**print**("Enter the new price (leave blank to keep current): ");

**String** newPriceStr = scanner.**nextLine**();

**System**.out.**print**("Enter the new stock quantity (leave blank to keep current): ");

**String** newStockQuantityStr = scanner.**nextLine**();

if (!newTitle.**isEmpty**()) {

bookToUpdate.**setTitle**(newTitle);

}

if (!newAuthor.**isEmpty**()) {

bookToUpdate.**setAuthor**(newAuthor);

}

if (!newPriceStr.**isEmpty**()) {

double newPrice = **Double**.**parseDouble**(newPriceStr);

bookToUpdate.**setPrice**(newPrice);

}

if (!newStockQuantityStr.**isEmpty**()) {

int newStockQuantity = **Integer**.**parseInt**(newStockQuantityStr);

bookToUpdate.**setStockQuantity**(newStockQuantity);

}

**System**.out.**println**("Book details updated successfully!");

} else {

**System**.out.**println**("No book found with ID: " + bookIdToUpdate);

}

}

*// Method to add a new book*

public void **addNewBook**(**Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the book title: ");

**String** title = scanner.**nextLine**();

**System**.out.**print**("Enter the book author: ");

**String** author = scanner.**nextLine**();

double price = 0;

while (true) {

try {

**System**.out.**print**("Enter the book price: ");

price = **Double**.**parseDouble**(scanner.**nextLine**());

break;

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid price.");

}

}

int stockQuantity = 0;

while (true) {

try {

**System**.out.**print**("Enter the stock quantity: ");

stockQuantity = **Integer**.**parseInt**(scanner.**nextLine**());

break;

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid stock quantity.");

}

}

*// Assign a new unique ID*

int newBookId = ++maxBookId;

**Book** newBook = new **Book**(newBookId, title, author, price, stockQuantity);

**addBook**(newBook);

**System**.out.**println**("Book added successfully with ID: " + newBookId);

}

*// Method to remove a book*

public void **removeBook**(**Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the book ID to remove: ");

int bookIdToRemove = **Integer**.**parseInt**(scanner.**nextLine**());

if (**removeBook**(bookIdToRemove)) {

**System**.out.**println**("Book removed successfully!");

} else {

**System**.out.**println**("No book found with ID: " + bookIdToRemove);

}

}

*// Method to add a book to the list*

public void **addBook**(**Book** book) {

books.**add**(book);

}

*// Method to remove a book from the list by its ID*

public boolean **removeBook**(int bookId) {

for (int i = 0; i < books.**size**(); i++) {

if (books.**get**(i).**getBookId**() == bookId) {

books.**remove**(i);

return true;

}

}

return false;

}

}

*// FILE: models/CartItem.java*

package **BooksApplication**.**models**;

public class **CartItem** implements **Comparable**<**CartItem**> {

private **Book** book; *// Reference to the Book object*

private int quantity; *// Quantity of the book in the cart*

*// Constructor to initialize the cart item object with a Book reference*

public **CartItem**(**Book** book, int quantity) {

this.book = book;

this.quantity = quantity;

}

*// Getter method for bookId*

public int **getBookId**() {

return book.**getBookId**();

}

*// Getter method for title*

public **String** **getTitle**() {

return book.**getTitle**();

}

*// Getter method for author*

public **String** **getAuthor**() {

return book.**getAuthor**();

}

*// Getter method for price*

public double **getPrice**() {

return book.**getPrice**();

}

*// Getter method for quantity*

public int **getQuantity**() {

return quantity;

}

*// Setter method for quantity*

public void **setQuantity**(int quantity) {

this.quantity = quantity;

}

*// Override compareTo method to compare cart items by title*

@**Override**

public int **compareTo**(**CartItem** other) {

return this.**getTitle**().**compareTo**(other.**getTitle**());

}

*// Override toString method to return a string representation of the cart item object*

@**Override**

public **String** **toString**() {

return **String**.**format**("Book ID: %d\tTitle: %s\tAuthor: %s\tPrice: %.2f\tQuantity: %d",

**getBookId**(), **getTitle**(), **getAuthor**(), **getPrice**(), quantity);

}

}

*// FILE: models/CartItemBuddy.java*

package **BooksApp**.**models**;

import **BooksApp**.**adt**.**ArrayListADT**;

import **BooksApp**.**algo**.**QuickSort**;

import **java**.**util**.**Scanner**;

public class **CartItemBuddy** {

private **ArrayListADT**<**CartItem**> cart; *// List to store cart items*

*// Constructor*

public **CartItemBuddy**() {

this.cart = new **ArrayListADT**<>();

}

*// Method to add a CartItem to the cart*

public void **addToCart**(**Book** book, int quantity) {

*// Check if the book is already in the cart*

for (int i = 0; i < cart.**size**(); i++) {

**CartItem** cartItem = cart.**get**(i);

if (cartItem.**getBookId**() == book.**getBookId**()) {

*// If the book is already in the cart, update the quantity*

cartItem.**setQuantity**(cartItem.**getQuantity**() + quantity);

return;

}

}

*// If the book is not in the cart, add a new CartItem with a reference to the*

*// book*

**CartItem** newCartItem = new **CartItem**(book, quantity);

cart.**add**(newCartItem);

}

*// Method to view the cart, sort and display the cart items*

public void **viewCart**() {

if (cart.**isEmpty**()) {

**System**.out.**println**("Your cart is empty.");

} else {

**QuickSort**<**CartItem**> sorter = new **QuickSort**<>();

sorter.**sort**(cart); *// Sort cart items by title before displaying*

**System**.out.**println**("Sorted Cart:");

for (int i = 0; i < cart.**size**(); i++) {

**CartItem** cartItem = cart.**get**(i);

**System**.out.**println**(cartItem);

}

}

}

*// Method to remove a CartItem from the cart by book ID*

public void **removeFromCart**(int bookId) {

**CartItem** itemToRemove = null;

*// Find the CartItem with the given book ID*

for (int i = 0; i < cart.**size**(); i++) {

**CartItem** cartItem = cart.**get**(i);

if (cartItem.**getBookId**() == bookId) {

itemToRemove = cartItem;

break;

}

}

*// If the CartItem is found, remove it from the cart*

if (itemToRemove != null) {

cart.**remove**(itemToRemove);

**System**.out.**println**("Book removed from cart!");

} else {

**System**.out.**println**("Book not found in cart.");

}

}

*// Method to remove a book from the cart with user input*

public void **removeBookFromCart**(**Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the book ID to remove from cart or 'n' to cancel: ");

**String** inputCart = scanner.**nextLine**();

if (!inputCart.**equalsIgnoreCase**("n")) {

try {

int bookIdToRemove = **Integer**.**parseInt**(inputCart);

**removeFromCart**(bookIdToRemove); *// Remove from cart*

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid book ID.");

}

}

}

*// Method to get a CartItem by book ID*

public **CartItem** **getCartItemById**(int bookId) {

*// Iterate through the cart to find the CartItem with the given book ID*

for (int i = 0; i < cart.**size**(); i++) {

**CartItem** cartItem = cart.**get**(i);

if (cartItem.**getBookId**() == bookId) {

return cartItem;

}

}

return null; *// If no CartItem found, return null*

}

*// Method to check if the cart is empty*

public boolean **isEmpty**() {

return cart.**isEmpty**();

}

*// Method to get all books in the cart*

public **ArrayListADT**<**CartItem**> **getBooksInCart**() {

return cart;

}

*// Method to get the quantity of a specific book in the cart*

public int **getQuantityInCart**(int bookId) {

*// Iterate through the cart to find the CartItem with the given book ID*

for (int i = 0; i < cart.**size**(); i++) {

**CartItem** cartItem = cart.**get**(i);

if (cartItem.**getBookId**() == bookId) {

return cartItem.**getQuantity**();

}

}

return 0; *// If no CartItem found, return 0*

}

*// Method to clear the cart*

public void **clearCart**() {

cart.**clear**();

}

}

*// FILE: models/Order.java*

package **BooksApp**.**models**;

import **java**.**util**.**Date**;

import **BooksApp**.**customer**.**Customer**;

import **BooksApp**.**adt**.**ArrayListADT**;

public class **Order** implements **Comparable**<**Order**> {

private int orderId; *// Unique identifier for the order*

private **Date** date; *// Date of the order*

private **Customer** customer; *// Customer who placed the order*

private **String** status; *// Status of the order (e.g., "Pending", "Shipped")*

private **ArrayListADT**<**CartItem**> items; *// List of items in the order*

*// Constructor to initialize the order object*

public **Order**(int orderId, **Date** date, **Customer** customer, **String** status, **ArrayListADT**<**CartItem**> items) {

this.orderId = orderId;

this.date = date;

this.customer = customer;

this.status = status;

this.items = items;

}

*// Getter method for orderId*

public int **getOrderId**() {

return orderId;

}

*// Getter method for date*

public **Date** **getDate**() {

return date;

}

*// Getter method for customer*

public **Customer** **getCustomer**() {

return customer;

}

*// Getter method for customer ID*

public int **getCustomerId**() {

return customer.**getUserId**();

}

*// Getter method for status*

public **String** **getStatus**() {

return status;

}

*// Setter method for status*

public void **setStatus**(**String** status) {

this.status = status;

}

*// Getter method for items*

public **ArrayListADT**<**CartItem**> **getItems**() {

return items;

}

*// Method to check if the order contains a book with a specific title*

public boolean **containsBookWithTitle**(**String** title) {

for (int i = 0; i < items.**size**(); i++) {

**CartItem** item = items.**get**(i);

if (item.**getTitle**().**equalsIgnoreCase**(title)) {

return true;

}

}

return false;

}

*// Override compareTo method to compare orders by orderId*

@**Override**

public int **compareTo**(**Order** other) {

return **Integer**.**compare**(this.orderId, other.orderId);

}

*// Override toString method to return a string representation of the order object*

@**Override**

public **String** **toString**() {

**StringBuilder** sb = new **StringBuilder**();

sb.**append**("Order Details:\n");

sb.**append**("---------------\n");

sb.**append**("Order ID: ").**append**(orderId).**append**("\n");

sb.**append**("Date: ").**append**(date).**append**("\n");

sb.**append**("Customer: ").**append**(customer).**append**("\n");

sb.**append**("Status: ").**append**(status).**append**("\n");

sb.**append**("Items:\n");

for (int i = 0; i < items.**size**(); i++) {

**CartItem** item = items.**get**(i);

sb.**append**("\t").**append**(item).**append**("\n");

}

return sb.**toString**();

}

}

*// FILE: models/OrderBuddy.java*

package **BooksApp**.**models**;

import **java**.**util**.**Date**;

import **java**.**util**.**Scanner**;

import **BooksApp**.**adt**.**ArrayListADT**;

import **BooksApp**.**customer**.**Customer**;

import **BooksApp**.**adt**.**QueueADT**;

import **BooksApp**.**algo**.**LinearSearch**;

import **BooksApp**.**algo**.**BinarySearch**;

import **BooksApp**.**algo**.**QuickSort**;

public class **OrderBuddy** {

private **ArrayListADT**<**Order**> orders; *// List to store order objects*

*// Constructor to initialize the orders list*

public **OrderBuddy**() {

orders = new **ArrayListADT**<>();

}

*// Method to create a new order*

public **Order** **createOrder**(**Date** date, **Customer** customer, **String** status, **ArrayListADT**<**CartItem**> items) {

int orderId = **generateOrderId**();

**Order** order = new **Order**(orderId, date, customer, status, items);

return order;

}

*// Method to add an order to the list*

public void **addOrder**(**Order** order) {

orders.**add**(order);

}

*// Method to get all orders as a queue*

public **QueueADT**<**Order**> **getOrders**() {

**QueueADT**<**Order**> orderQueue = new **QueueADT**<>();

for (int i = 0; i < orders.**size**(); i++) {

**Order** order = orders.**get**(i);

orderQueue.**offer**(order);

}

return orderQueue;

}

*// Method to generate a unique order ID*

private int **generateOrderId**() {

return orders.**size**() + 1;

}

*// Method to view the order history for a logged-in user*

public void **viewOrdersHistory**(**User** loggedInUser) {

**System**.out.**println**("----------------------");

**System**.out.**println**("Orders History:");

**QueueADT**<**Order**> orders = **getOrders**();

if (orders.**isEmpty**()) {

**System**.out.**println**("No orders found.");

} else {

boolean orderFound = false;

**QueueADT**<**Order**> tempQueue = new **QueueADT**<>();

while (!orders.**isEmpty**()) {

**Order** order = orders.**poll**();

if (order.**getCustomerId**() == loggedInUser.**getUserId**()) {

**System**.out.**println**(order);

orderFound = true;

}

tempQueue.**offer**(order);

}

*// Restore the original queue*

while (!tempQueue.**isEmpty**()) {

orders.**offer**(tempQueue.**poll**());

}

if (!orderFound) {

**System**.out.**println**("No orders found for your account.");

}

}

}

*// Method to search orders as a customer*

public void **searchOrdersAsCustomer**(**User** loggedInUser, **Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**println**("1. Search by Order ID");

**System**.out.**println**("2. Search by Book Title");

**System**.out.**print**("Enter your choice: ");

**String** searchChoice = scanner.**nextLine**();

if (searchChoice.**equals**("1")) {

**System**.out.**print**("Enter the order ID to search: ");

**String** searchOrderQuery = scanner.**nextLine**();

int orderId;

try {

orderId = **Integer**.**parseInt**(searchOrderQuery); *// Check if the input is a valid number*

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid order ID.");

return;

}

**System**.out.**println**("Searching for orders matching: " + searchOrderQuery);

**LinearSearch**<**Order**> linearSearch = new **LinearSearch**<>();

**QueueADT**<**Order**> ordersQueue = **getOrders**();

**QueueADT**<**Order**> tempQueue = new **QueueADT**<>();

*// Convert QueueADT to ArrayListADT for searching*

**ArrayListADT**<**Order**> orderList = new **ArrayListADT**<>();

while (!ordersQueue.**isEmpty**()) {

**Order** order = ordersQueue.**poll**();

orderList.**add**(order);

tempQueue.**offer**(order);

}

*// Restore the original queue*

while (!tempQueue.**isEmpty**()) {

ordersQueue.**offer**(tempQueue.**poll**());

}

*// Perform the search*

int index = linearSearch.**search**(orderList, null,

order -> order.**getCustomerId**() == loggedInUser.**getUserId**()

&& order.**getOrderId**() == orderId);

if (index != -1) {

**System**.out.**println**(orderList.**get**(index));

} else {

**System**.out.**println**("No orders found matching: " + searchOrderQuery);

}

} else if (searchChoice.**equals**("2")) {

**System**.out.**print**("Enter the book title to search: ");

**String** bookTitle = scanner.**nextLine**();

**System**.out.**println**("Searching for orders containing book title: " + bookTitle);

**QueueADT**<**Order**> ordersQueue = **getOrders**();

**QueueADT**<**Order**> tempQueue = new **QueueADT**<>();

*// Convert QueueADT to ArrayListADT for searching*

**ArrayListADT**<**Order**> orderList = new **ArrayListADT**<>();

while (!ordersQueue.**isEmpty**()) {

**Order** order = ordersQueue.**poll**();

orderList.**add**(order);

tempQueue.**offer**(order);

}

*// Restore the original queue*

while (!tempQueue.**isEmpty**()) {

ordersQueue.**offer**(tempQueue.**poll**());

}

*// Perform the search*

**ArrayListADT**<**Order**> matchingOrders = new **ArrayListADT**<>();

for (int i = 0; i < orderList.**size**(); i++) {

**Order** order = orderList.**get**(i);

if (order.**getCustomerId**() == loggedInUser.**getUserId**()

&& order.**containsBookWithTitle**(bookTitle)) {

matchingOrders.**add**(order);

}

}

if (!matchingOrders.**isEmpty**()) {

for (int i = 0; i < matchingOrders.**size**(); i++) {

**System**.out.**println**(matchingOrders.**get**(i));

}

} else {

**System**.out.**println**("No orders found containing book title: " + bookTitle);

}

}

}

*// Method to view all orders as an admin*

public void **viewAllOrdersAsAdmin**(**Scanner** scanner) {

**System**.out.**println**("----------------------");

**System**.out.**println**("All Orders:");

**QueueADT**<**Order**> ordersQueue = **getOrders**();

**QueueADT**<**Order**> tempQueue = new **QueueADT**<>();

if (ordersQueue.**isEmpty**()) {

**System**.out.**println**("No orders found.");

} else {

*// Convert QueueADT to ArrayListADT for displaying, sorting, and searching*

**ArrayListADT**<**Order**> orderList = new **ArrayListADT**<>();

while (!ordersQueue.**isEmpty**()) {

**Order** order = ordersQueue.**poll**();

orderList.**add**(order);

tempQueue.**offer**(order);

}

*// Restore the original queue*

while (!tempQueue.**isEmpty**()) {

ordersQueue.**offer**(tempQueue.**poll**());

}

*// Display all orders*

for (int i = 0; i < orderList.**size**(); i++) {

**System**.out.**println**(orderList.**get**(i));

}

*// Search Orders as admin*

**System**.out.**println**("----------------------");

**System**.out.**print**("Enter the order ID to search: ");

**String** searchOrderQuery = scanner.**nextLine**();

int orderId;

try {

orderId = **Integer**.**parseInt**(searchOrderQuery); *// Check if the input is a valid number*

} catch (**NumberFormatException** e) {

**System**.out.**println**("Invalid input. Please enter a valid order ID.");

return;

}

**System**.out.**println**("Searching for orders matching: " + searchOrderQuery);

*// Sort the orders by order ID using QuickSort*

**QuickSort**<**Order**> sorter = new **QuickSort**<>();

sorter.**sort**(orderList);

*// Perform the binary search*

**BinarySearch**<**Order**> binarySearch = new **BinarySearch**<>();

int index = binarySearch.**search**(orderList, new **Order**(orderId, null, null, null, null));

if (index != -1) {

**Order** foundOrder = orderList.**get**(index);

**System**.out.**println**(foundOrder);

*// Change order status*

**System**.out.**print**("Enter the new status for the order: ");

**String** newStatus = scanner.**nextLine**();

foundOrder.**setStatus**(newStatus);

**System**.out.**println**("Order status updated successfully!");

} else {

**System**.out.**println**("No orders found matching: " + searchOrderQuery);

}

}

}

}

### Run the app

Register as customer

A screen shot of a computer

Description automatically generated

Login

A screenshot of a computer screen

Description automatically generated

Display all books

A close-up of a computer screen

Description automatically generated

Add book to cart

A screenshot of a computer

Description automatically generated

Go to cart

A screenshot of a computer

Description automatically generated

Change book quantity and remove book in cart



Place order and order history

A screenshot of a computer

Description automatically generated

When order was placed, cart items were deleted and book quantity in stock was updated

A screen shot of a computer

Description automatically generated

Search order by order ID or book title in order

A close-up of a computer screen

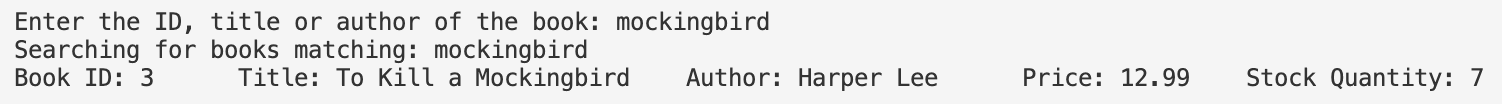
Description automatically generatedA screenshot of a computer

Description automatically generated

Search book

A screenshot of a computer

Description automatically generated



Search book history

A white paper with black text

Description automatically generated

Order many ordersA screenshot of a computer

Description automatically generated

Logout

A white paper with black text

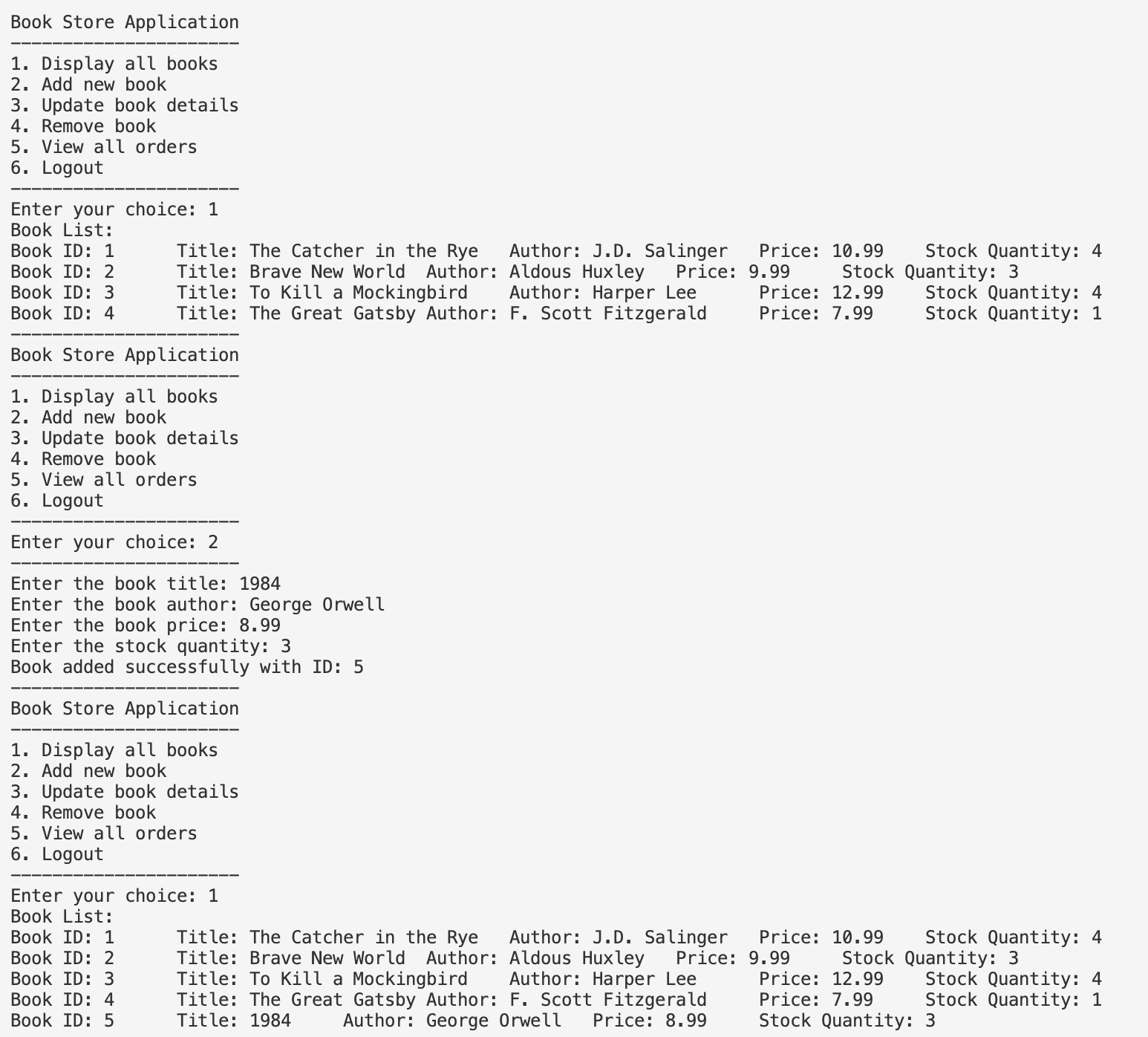
Description automatically generated

Register and login as admin

A screenshot of a computer screen

Description automatically generated

Add a new book with a new book ID



Update book details

A screenshot of a computer

Description automatically generated

Remove book

A screen shot of a computer

Description automatically generated

View all orders, search order by order ID, update new status for that order

A screenshot of a computer

Description automatically generated

Logout

A close up of words

Description automatically generated

Login as a customer to see updated order status

A screenshot of a computer

Description automatically generated

**Errors and exceptions handeling**

Invalid email or password

A screen shot of a computer

Description automatically generated

Invalid admin key. Registragtion as admin failed

A screenshot of a computer

Description automatically generated

Invalid choice

A screenshot of a computer

Description automatically generated

No book found

A white screen with black text

Description automatically generatedA white screen with black text

Description automatically generated

Invalid book quantity (order more books than available books in stock)

A screenshot of a computer screen

Description automatically generated

No order found

A close-up of a computer screen

Description automatically generated

A white background with black text

Description automatically generated

No search history found

A white paper with black text

Description automatically generated

Your cart is empty

A white paper with black text

Description automatically generated

Invalid input

A screenshot of a computer

Description automatically generated

## Task 3:

### Data Structures

1. **ArrayListADT**
   * **Usage**: This is used extensively to store collections of objects, such as books, customers, admins, and orders.
   * **Efficiency**: The ArrayListADT provides dynamic array capabilities, allowing for efficient random access and modification of elements. It supports operations like adding, removing, and accessing elements in constant or linear time, making it suitable for scenarios where the size of the collection can change frequently.
   * **Design Choice**: The choice of ArrayListADT over a simple array is due to its dynamic resizing capability, which ensures that the list can grow as needed without manual intervention.
2. **QueueADT**
   * **Usage**: This is used to manage orders in a first-in, first-out (FIFO) manner.
   * **Efficiency**: The QueueADT ensures that orders are processed in the order they are received, which is crucial for maintaining fairness and order in processing customer requests.
   * **Design Choice**: Using a queue for orders ensures that the oldest orders are processed first, which is a common requirement in order management systems.
3. **StackADT**
   * **Usage**: This is used to store search history.
   * **Efficiency**: The StackADT provides last-in, first-out (LIFO) access, which is ideal for undo operations or tracking the most recent searches.
   * **Design Choice**: Using a stack for search history allows users to easily backtrack through their recent searches, enhancing the user experience.

### Algorithms

1. **QuickSort**
   * **Usage**: This is used to sort collections of books and orders.
   * **Efficiency**: QuickSort is an efficient, comparison-based sorting algorithm with an average-case time complexity of O(n log n). It is well-suited for large datasets.
   * **Design Choice**: QuickSort is chosen for its efficiency and performance in sorting large collections, ensuring that operations like displaying sorted books or orders are fast.
2. **BinarySearch**
   * **Usage**: This is used to search for books and orders by their IDs.
   * **Efficiency**: BinarySearch has a time complexity of O(log n), making it very efficient for searching in sorted collections.
   * **Design Choice**: Using BinarySearch ensures that searches are performed quickly, which is crucial for providing a responsive user experience.
3. **LinearSearch**
   * **Usage**: This is used to search for books and orders based on attributes other than IDs, such as titles or authors.
   * **Efficiency**: LinearSearch has a time complexity of O(n), which is suitable for smaller datasets or when the collection is not sorted.
   * **Design Choice**: LinearSearch is used for its simplicity and flexibility in searching for items based on various attributes.

## Task 4:

Analyzing the Complexity of Methods:

### 1. searchOrdersAsAdmin

Complexity Analysis

1. Input Validation: O(1) - Constant time to check if the input is a valid number.
2. Sorting Orders: O(n log n) - QuickSort for sorting orders by ID.
3. Binary Search: O(log n) - Binary search on the sorted list.
4. Updating Order Status: O(1) - Constant time to update the order status.

Total Complexity: O(n log n)

Justification of Design Choices

1. QuickSort: Efficient sorting algorithm with O(n log n) complexity.
2. Binary Search: Fast retrieval with O(log n) complexity.
3. Input Validation: Ensures correct input with O(1) complexity.

Future Considerations

1. Scalability: QuickSort remains efficient for large datasets.
2. Data Structure Optimization: Consider balanced binary search trees for O(log n) operations.
3. Algorithm Optimization: Optimize QuickSort for specific dataset characteristics.

### 2. Search book history

Complexity Analysis

1. Checking if the Stack is Empty:
   * Complexity: O(1) - Constant time operation.
2. Popping Elements from searchHistory and Pushing to tempStack:
   * Complexity: O(n) - Iterates through all elements, each pop and push is O(1).
3. Restoring the Original Stack:
   * Complexity: O(n) - Iterates through all elements, each pop and push is O(1).

Total Complexity: O(n) - Linear, as it involves iterating through all elements twice.

Justification of Design Choices

1. Use of StackADT:
   * Reason: Provides LIFO access, suitable for search history.
   * Efficiency: Push and pop operations are O(1).
2. Temporary Stack for Restoration:
   * Reason: Maintains the order of elements.
   * Efficiency: Restoration process is O(n), efficient for moderate-sized stacks.

Future Considerations

1. Scalability:
   * Efficient for moderate-sized stacks. O(n) remains efficient as queries grow.
2. Data Structure Optimization:
   * For very large stacks, consider advanced data structures or optimization techniques.

### 3. Method to get all orders as a queue

Complexity Analysis

1. Iterating Through Orders:
   * Complexity: O(n) - Iterates through all elements in the orders list.
2. Adding Orders to Queue:
   * Complexity: O(1) per operation - Each offer operation is constant time.
   * Total Complexity: O(n) - Adding all orders to the queue.

Total Complexity: O(n) - Linear, as it involves iterating through all elements once.

Justification of Design Choices

1. Use of QueueADT:
   * Reason: Provides FIFO access, suitable for processing orders.
   * Efficiency: offer operation is O(1), making it efficient for adding elements.

### 4. searchOrdersAsCustomer

I chose linear search for the searchOrdersAsCustomer method because each customer has different orders, and using binary search would require sorting all orders of the customer before searching. This makes linear search more efficient in this context.

Complexity Analysis

1. Input Validation:
   * Complexity: O(1) - Checking if the input is a valid number is a constant time operation.
2. Queue to ArrayList Conversion:
   * Complexity: O(n) - Converting the queue to an array list involves iterating through all orders.
   * Details: Each poll and offer operation is O(1), but since we iterate through all elements, the total complexity is O(n).
3. Linear Search by Order ID:
   * Complexity: O(n) - Linear search through the array list to find the order by ID.
   * Details: The search involves iterating through the list and checking each order's ID.
4. Linear Search by Book Title:
   * Complexity: O(n) - Linear search through the array list to find orders containing the book title.
   * Details: The search involves iterating through the list and checking each order's book titles.

Total Complexity: O(n) - The overall complexity is linear, dominated by the conversion and search operations.

Justification of Design Choices

1. Use of QueueADT:
   * Reason: QueueADT is used to manage orders in a FIFO manner, ensuring that orders are processed in the order they were placed.
   * Efficiency: While converting the queue to an array list has a linear time complexity, it allows for efficient searching using linear search.
2. Linear Search:
   * Reason: Linear search is simple and effective for the current scale of the application.
   * Efficiency: Linear search has a time complexity of O(n), which is acceptable for moderate-sized lists. However, as the number of orders grows, this may become a bottleneck.
3. Queue Restoration:
   * Reason: Restoring the original queue ensures that the order of elements is maintained after the search operation.
   * Efficiency: The restoration process has a linear time complexity (O(n)), which is necessary to maintain the integrity of the queue.

Future Considerations

1. Scalability:
   * As the number of orders grows, the linear search operations (O(n)) may become a bottleneck. To improve scalability, we could:
     + Use a hash map to store orders by their IDs, reducing the search time to O(1) on average.
     + Implement binary search on a sorted list of orders, reducing the search time to O(log n).
2. Data Structure Optimization:
   * Consider using more advanced data structures like balanced binary search trees (e.g., AVL trees, Red-Black trees) or hash maps to improve the efficiency of search operations.
3. Algorithm Optimization:
   * For sorting operations, ensure that efficient algorithms like QuickSort (O(n log n) average time complexity) are used.
   * For frequent search operations, consider using binary search (O(log n) time complexity) on sorted lists.

### 5. addNewBook

Complexity Analysis

1. Time Complexity:
   * Input Reading: O(1)
   * ID Assignment: O(1)
   * Book Creation: O(1)
   * Adding Book to List: O(1) (average), O(n) (worst-case for resizing)
2. Space Complexity: O(1) for method, O(n) for ArrayListADT.

Justification of Design Choices

1. ArrayListADT:
   * Pros: Fast access and addition (average O(1)).
   * Cons: Costly resizing (O(n)).
2. Scanner for Input:
   * Pros: Easy to use.
   * Cons: Blocking I/O operations.

Scalability Considerations

1. Data Storage: Use LinkedList or HashMap for better scalability.
2. Input Handling: Implement asynchronous I/O or web-based interface.
3. Concurrency: Use synchronization mechanisms or concurrent data structures.

### 6. Changing Quantity in Cart

Complexity Analysis

1. Retrieving the CartItem: O(n) - Linear search in ArrayListADT<CartItem>.
2. Retrieving the Book: O(n) - Linear search in ArrayListADT<Book>.
3. Validating and Updating Quantity: O(1) - Constant time for validation and update.

Justification of Design Choices

1. ArrayListADT: Efficient access and modification, dynamic resizing.
2. Linear Search: Simple and effective for moderate-sized lists.
3. Input Validation: Ensures inventory integrity with O(1) complexity.

Future Considerations

1. Data Structure Optimization: Use balanced binary search trees or hash maps for efficient search operations.