XIAMEN UNIVERSITY MALAYSIA

Course Code	:	CST207
Course Name	:	Design and Analysis of Algorithms
Lecturer	:	Dr. Mohammed N. M. Ali
Academic Session	:	2024/09
Assessment Title	:	Project (Group)
Submission Due Date	:	30th December

Prepared by :

Student ID	Student Name
CYS2309202	Sun Zhengwu
CYS2309205	Zheng Yucheng
CYS2309680	Lim Jun Ming
CYS2309275	Mosaddiqur Rahman
CYS2309193	Chen Junfeng

Date Received :

Feedback from Lecturer:

Mark:

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

郑煜程 陈俊锋

Date:2024.12.27

1. Comparison and challenges of implemented search algorithm

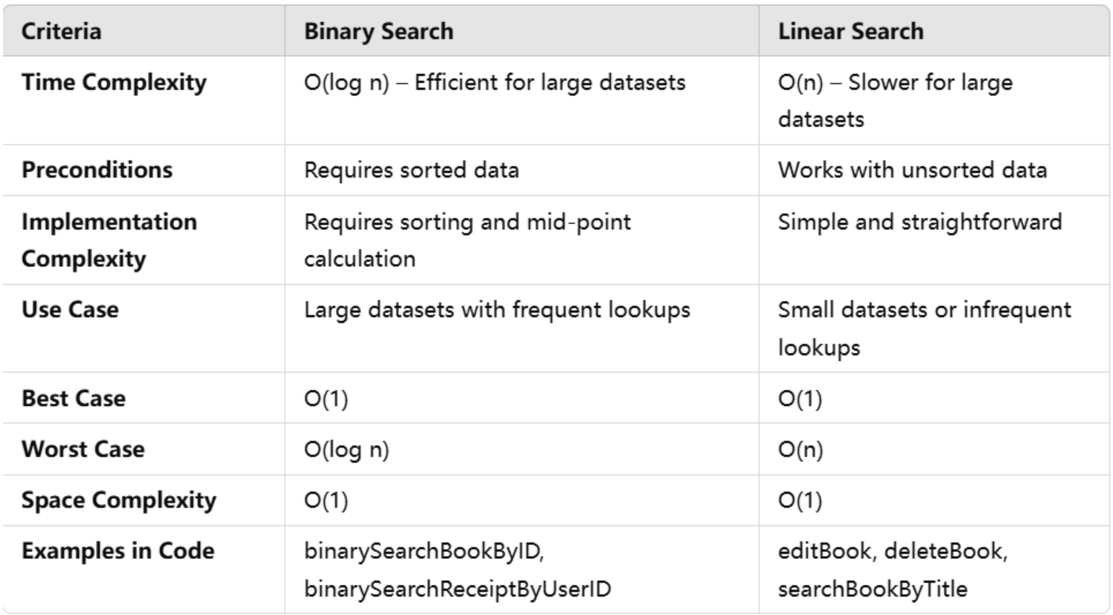
The main search algorithms used in the project are binary search and linear search. Binary search is a search method that continuously divides the search range into two to find the target element. At the same time, the limitation is that the data must be ordered. In the project, we used binary search functions named binarySearchBookByID, binarySearchBookByID and binarySearchReceiptByUserID. The first one is to find books by book ID, the second one is to find books by book title substring, and the last one is to find transaction records by user ID.

Another linear search algorithm is a search method that searches for elements one by one until the target is found or the entire data set is traversed. It is used in the editBook function, deleteBook and some functions in the project. Find_if function is used to find and edit books by book ID and to find and delete books by book ID. It is also used in the "viewTransactionHistoryByUserID_LinearSearch" function to find borrowing records by user ID.

Finally, "searchBookByTitle" searches for books based on the substring of the book title. For the comparison of the two-time complexities, the best case of both is O (1). Binary search corresponds to the case when the target element is in the middle, and linear search corresponds to the case when the target element is in the first position. In the average and worst cases, the two are completely different. Binary search can halve the search range in each iteration, so its complexity is O (log _n), while linear search needs to traverse the entire data set, resulting in its time complexity of O(n). In terms of space complexity, both are O (1). Binary search only needs a few variables to save indexes and intermediate results, and linear search only requires constant space.

Next, considering the characteristics of the two, binary search requires ordered data, while linear search does not, which reflects the limitations of binary search. However, for large data sets, binary search is more efficient, which means that for string search in the corresponding project, binary search is obviously more suitable. In terms of implementation difficulty, linear search logic is intuitive and easier to implement and understand.

The following is a comparison table:



Criteria	Binary Search	Linear Search
Time Complexity	O(log n) – Efficient for large datasets	O(n) – Slower for large datasets
Preconditions	Requires sorted data	Works with unsorted data
Implementation Complexity	Requires sorting and mid-point calculation	Simple and straightforward
Use Case	Large datasets with frequent lookups	Small datasets or infrequent lookups
Best Case	O(1)	O(1)
Worst Case	O(log n)	O(n)
Space Complexity	O(1)	O(1)
Examples in Code	binarySearchBookByID, binarySearchReceiptByUserID	editBook, deleteBook, searchBookByTitle

In the implementation of binary search, due to its limitation, data must be ordered, so we have to consider how to keep the data in order after it is updated. In this project, this is reflected in the need to re-sort after each insertion or deletion of a book. After team discussion and reference, we decided to use quick sort to sort the book IDs and bubble sort to sort the book titles. This method ensures that the binarySearchBookByID binary search function can be executed smoothly and efficiently to find books.

When linear search is used in multiple places, the code may be redundant and difficult to maintain. We encapsulate linear search into independent functions, such as viewTransactionHistoryByUserID_LinearSearch, to improve code reusability and readability.

2. Easiest search algorithm

Linear search is the easiest search algorithm as it is a much straightforward option. Linear search operates by traversing each element of a dataset sequentially until the targeted element is found or the list ends.

In our example, we used linear search on the function for viewing transaction history. Our function iterates through the receipts vectors and check each element against the targeted element When a match is found, it prints out the transaction receipt. Else, it prints an error message.

The reason why linear search is a much easier

Linear search has minimal prerequisites. Linear search requires no preprocessing data, it is applicable for both sorted and unsorted datasets. Compared to binary search, it is mandatory for the dataset to be sorted. Linear search's biggest advantage is its robustness against implementation errors. Linear search does not involve indices, pivot and recursive calls that could result in possible results such as out of bounds and stack overflow. Due to the linear search algorithm conducted in a linear way, testing and debugging is much easier and direct.

3. Comparison of sorting algorithms and any difficulties encountered.

To fulfil the need for orderly management of book data and flow records, the assignment required the use of four classic sorting algorithms: Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. These algorithms exhibit certain complexities during implementation and debugging, with potential for coding errors, making it necessary to conduct practical operations to examine their performance. This paper makes a detailed comparison of the usage of these four algorithms and discusses some difficulties during their implementation.

Merge Sort was used for sorting the bibliography list in ascending order according to book IDs. It has a time complexity of O (nlog n) in both average and worst cases, making it suitable for large datasets. Merge Sort is a divide-and-conquer algorithm that recursively divides the data into smaller pieces and then merges them into a sorted sequence. In real implementation, the most important problems were the exact calculation of indices and the use of temporary arrays; any mistake could lead to out-of-bounds or misaligned data. By calculating the index carefully, testing thoroughly, and validating the boundary conditions of recursion, the correctness of Merge Sort was finally ensured.

The Bubble Sort was used for sorting book titles. Because it is simple and intuitive, Bubble Sort repeatedly compares adjacent elements and swaps them if necessary, causing larger elements to "bubble" toward the end of the sequence. Its time complexity is $O(n^2)$, but due to the infrequency of sorting book titles and relatively small dataset size, performance is adequate for practical purposes. The major concern was the fact that if the volume of data suddenly increased, the sorting time could also increase substantially. To further improve the performance, an early termination mechanism was implemented that would terminate the algorithm after a traversal if no swaps were performed.

Selection Sort was used to sort the transaction records. Selection Sort is an implementation that repeatedly finds the smallest element in the unsorted portion of the list and swaps it with the current position. It is straightforward to implement and particularly efficient for sorting small lists quickly. The only challenge was accurately identifying and updating the index of the minimum value: any mistake in initialization or updates would lead to wrong sorting. These were resolved by rigorously verifying the initial conditions of each iteration and the correctness of element swaps.

Quick Sort was used to generate user-requested lists sorted by book IDs. It is suitable for dynamic and real-time sorting tasks due to its high average efficiency and in-place sorting capability. The most significant problems were the choice of a suitable pivot and the manipulation of the partitioning phase. A poorly chosen set of pivots could yield $O(n^2)$ complexity. No such performance problems are experienced during debugging, yet there are such possible scenarios. To prevent this scenario, a randomized strategy is implemented for the selection of pivots. This randomization prevents bias and yields good reliability and efficiency from the algorithm.

Every one of these above sorting algorithms has their positives and negatives. The appropriate applications include using Merge Sort for large-scale data, Bubble Sort for small-scale and infrequent sorting tasks, Selection Sort for the quick development of a program when the data sets are small, and Quick Sort for its efficiency and flexibility in medium-scale and dynamic sorting tasks. With different optimization strategies adopted for different sorting algorithms according to their characteristics, the reliability and efficiency of the sorting functions could be ensured.

4. The Easier Sorting Algorithm to Implement

While developing the university library's LMS (Library Management System), a variety of sorting algorithms were employed, such as Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. Although every algorithm possesses unique strengths and challenges, Bubble Sort was notable for being the simplest to implement. Its straightforwardness, clarity, and low logic demands render it perfect for projects that need fundamental sorting capabilities.

Clarity and Ease of Comprehension

Bubble Sort follows a straightforward concept: examine two neighbouring elements and interchange them if they are not in the correct sequence. This procedure continues until the entire list is sorted. Its gradual approach simplifies understanding for newcomers, as it simulates a natural method of organizing objects, similar to sorting playing cards.

In comparison to other sorting algorithms, Bubble Sort is significantly easier to understand since it only necessitates a fundamental grasp of loops and conditional expressions. Conversely, Merge Sort and Quick Sort encompass more complex ideas, including recursion and divide-and-conquer strategies. Merge Sort, for instance, splits the array into smaller subarrays, sorts them recursively, and then merges them back into one. Likewise, Quick Sort chooses a pivot, divides the array, and sorts the partitions recursively. These steps necessitate a greater comprehension of programming logic, which makes their execution more difficult.

No Requirement for Complex Logic or Recursion

A key benefit of Bubble Sort is its lack of dependence on recursion or intricate logic. Recursion, employed in algorithms such as Merge Sort and Quick Sort, can be challenging for newcomers to comprehend since it demands an understanding of stack memory and termination criteria. In contrast, Bubble Sort employs an iterative method, which simplifies both understanding and execution. This simplicity makes Bubble Sort especially appropriate for tasks that don't demand high efficiency

Simple Troubleshooting and Visualization

Bubble Sort's consistent and orderly nature makes it one of the simplest sorting algorithms to analyse and understand visually. Through gradual comparison and exchange of elements, developers can effectively monitor the algorithm's advancement and spot possible mistakes. Visualization tools frequently utilize Bubble Sort as a demonstration due to its straightforward operation and ease of understanding.

Easier Code and Quicker Execution

In comparison to more intricate algorithms such as Merge Sort and Quick Sort, Bubble Sort demands significantly less code and effort to execute. More complex algorithms necessitate extra procedures, like dividing arrays, handling pivots, or combining subarrays. These actions extend the code's length and intricacy. Bubble Sort, due to its simple method, can be executed quickly with just a few lines of code.

Constraints and Appropriateness

Although Bubble Sort is simple to implement, it has notable drawbacks. The time complexity is O(n^2), rendering it impractical for large data sets. Algorithms such as Merge Sort (O (nlog_n)) and Quick Sort (average-case O (nlog_n)) are significantly more effective for sorting larger data sets. Nevertheless, for small datasets where efficiency is not essential, Bubble Sort is a viable option.

In the LMS project, Bubble Sort was employed to arrange book titles, a task that dealt with a fairly small dataset. Its straightforwardness guaranteed swift and precise execution without needing complex reasoning. For larger sorting assignments, such as organizing books by ID, more effective algorithms like Merge Sort and Quick Sort were employed to enhance performance.

Conclusion

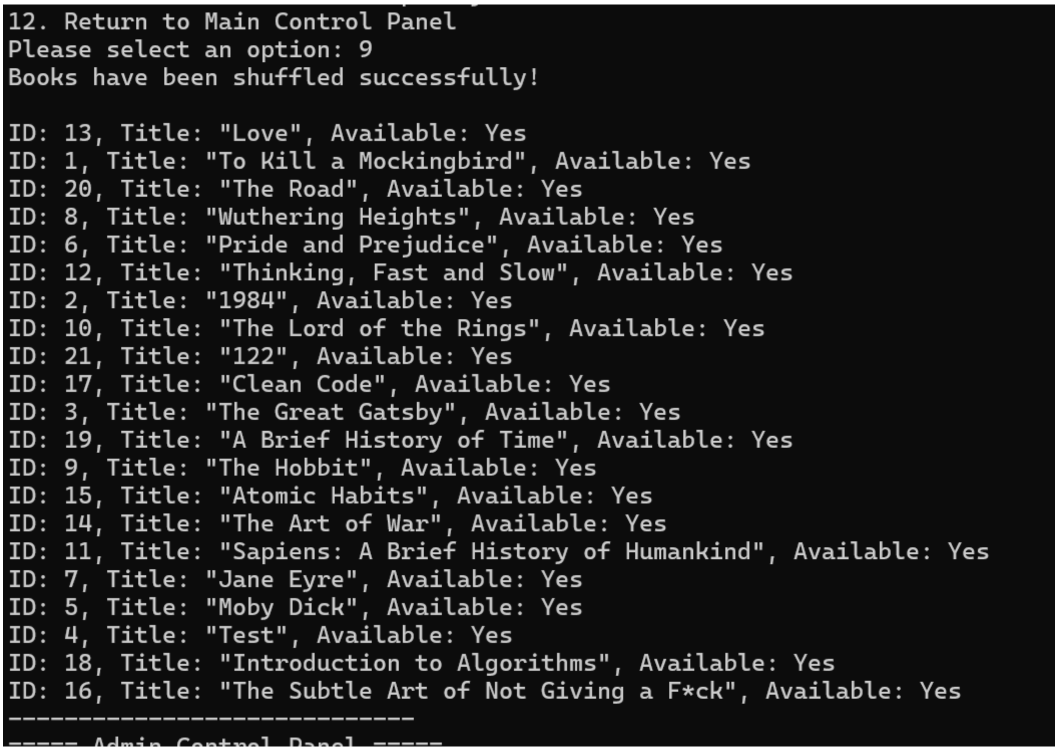
Bubble Sort is known for being the easiest sorting algorithm to use because it has a simple design, easy-to-understand logic, and is straightforward to debug. While it’s not suitable for large datasets due to its inefficiency, its simplicity makes it a handy tool for smaller projects.

5. Brief Explanation of Algorithms Used

The library management system integrates several fundamental algorithms to optimize data handling and enhance user experience. For instance, **Bubble Sort** is employed to organize books alphabetically by their titles, ensuring that administrators can easily navigate through the collection. **(**Figure 1 "Before Bubble sort"& Figure 2"After Bubble sort"**)**

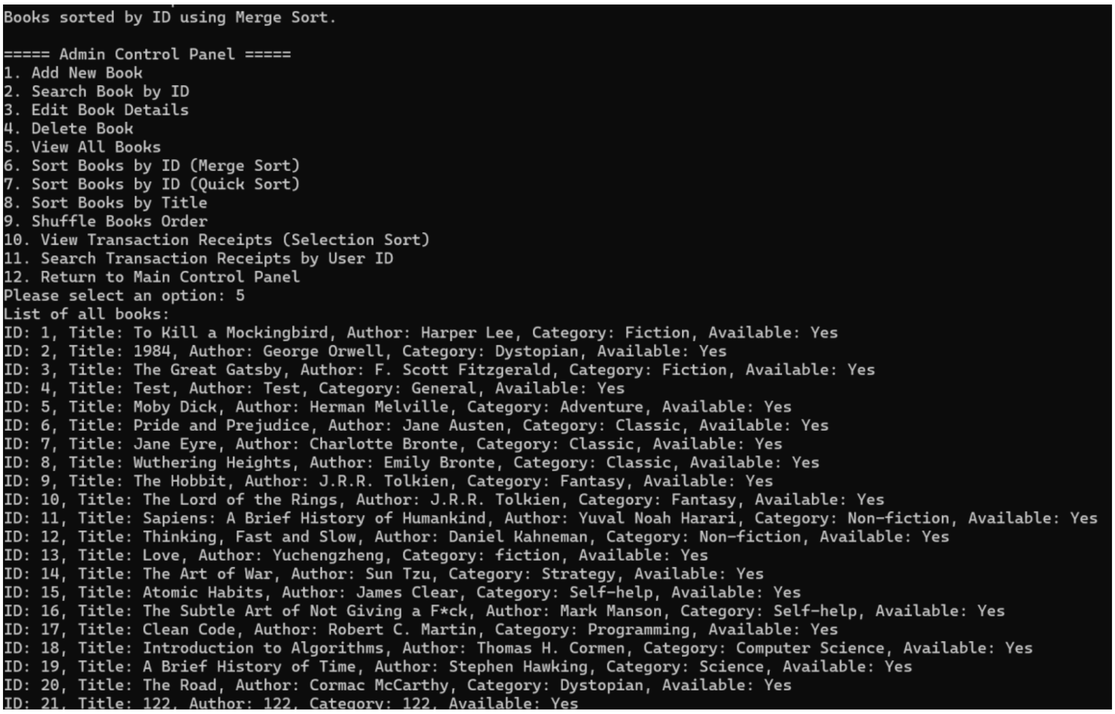
Additionally, the system utilizes both **Merge Sort** and **Quick Sort** to efficiently sort books numerically by their IDs, which facilitates rapid retrieval and management of book records.

(



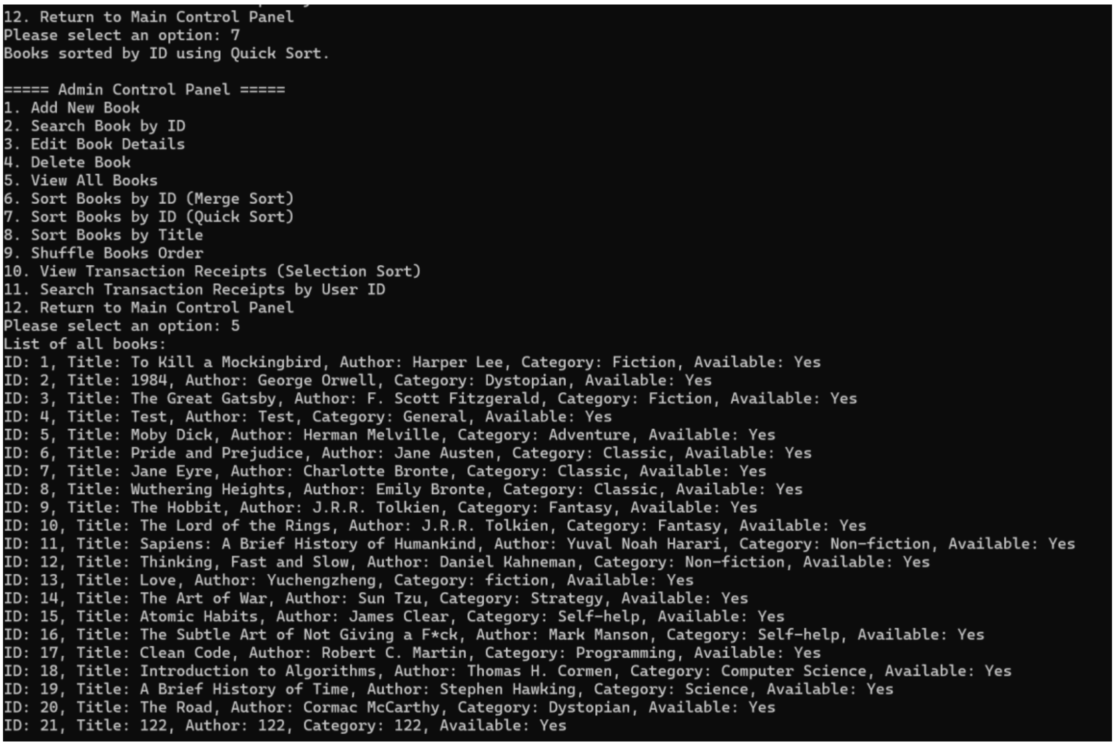
```
12. Return to Main Control Panel
Please select an option: 9
Books have been shuffled successfully!

ID: 13, Title: "Love", Available: Yes
ID: 1, Title: "To Kill a Mockingbird", Available: Yes
ID: 20, Title: "The Road", Available: Yes
ID: 8, Title: "Wuthering Heights", Available: Yes
ID: 6, Title: "Pride and Prejudice", Available: Yes
ID: 12, Title: "Thinking, Fast and Slow", Available: Yes
ID: 2, Title: "1984", Available: Yes
ID: 10, Title: "The Lord of the Rings", Available: Yes
ID: 21, Title: "122", Available: Yes
ID: 17, Title: "Clean Code", Available: Yes
ID: 3, Title: "The Great Gatsby", Available: Yes
ID: 19, Title: "A Brief History of Time", Available: Yes
ID: 9, Title: "The Hobbit", Available: Yes
ID: 15, Title: "Atomic Habits", Available: Yes
ID: 14, Title: "The Art of War", Available: Yes
ID: 11, Title: "Sapiens: A Brief History of Humankind", Available: Yes
ID: 7, Title: "Jane Eyre", Available: Yes
ID: 5, Title: "Moby Dick", Available: Yes
ID: 4, Title: "Test", Available: Yes
ID: 18, Title: "Introduction to Algorithms", Available: Yes
ID: 16, Title: "The Subtle Art of Not Giving a F*ck", Available: Yes
-----------------------------
----- Admin Control Panel -----
```

Figure 3 "Before sorting",



```
Books sorted by ID using Merge Sort.

===== Admin Control Panel =====
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 5
List of all books:
ID: 1, Title: To Kill a Mockingbird, Author: Harper Lee, Category: Fiction, Available: Yes
ID: 2, Title: 1984, Author: George Orwell, Category: Dystopian, Available: Yes
ID: 3, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Category: Fiction, Available: Yes
ID: 4, Title: Test, Author: Test, Category: General, Available: Yes
ID: 5, Title: Moby Dick, Author: Herman Melville, Category: Adventure, Available: Yes
ID: 6, Title: Pride and Prejudice, Author: Jane Austen, Category: Classic, Available: Yes
ID: 7, Title: Jane Eyre, Author: Charlotte Bronte, Category: Classic, Available: Yes
ID: 8, Title: Wuthering Heights, Author: Emily Bronte, Category: Classic, Available: Yes
ID: 9, Title: The Hobbit, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 10, Title: The Lord of the Rings, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 11, Title: Sapiens: A Brief History of Humankind, Author: Yuval Noah Harari, Category: Non-fiction, Available: Yes
ID: 12, Title: Thinking, Fast and Slow, Author: Daniel Kahneman, Category: Non-fiction, Available: Yes
ID: 13, Title: Love, Author: Yuchengzheng, Category: fiction, Available: Yes
ID: 14, Title: The Art of War, Author: Sun Tzu, Category: Strategy, Available: Yes
ID: 15, Title: Atomic Habits, Author: James Clear, Category: Self-help, Available: Yes
ID: 16, Title: The Subtle Art of Not Giving a F*ck, Author: Mark Manson, Category: Self-help, Available: Yes
ID: 17, Title: Clean Code, Author: Robert C. Martin, Category: Programming, Available: Yes
ID: 18, Title: Introduction to Algorithms, Author: Thomas H. Cormen, Category: Computer Science, Available: Yes
ID: 19, Title: A Brief History of Time, Author: Stephen Hawking, Category: Science, Available: Yes
ID: 20, Title: The Road, Author: Cormac McCarthy, Category: Dystopian, Available: Yes
ID: 21, Title: 122, Author: 122, Category: 122, Available: Yes
```

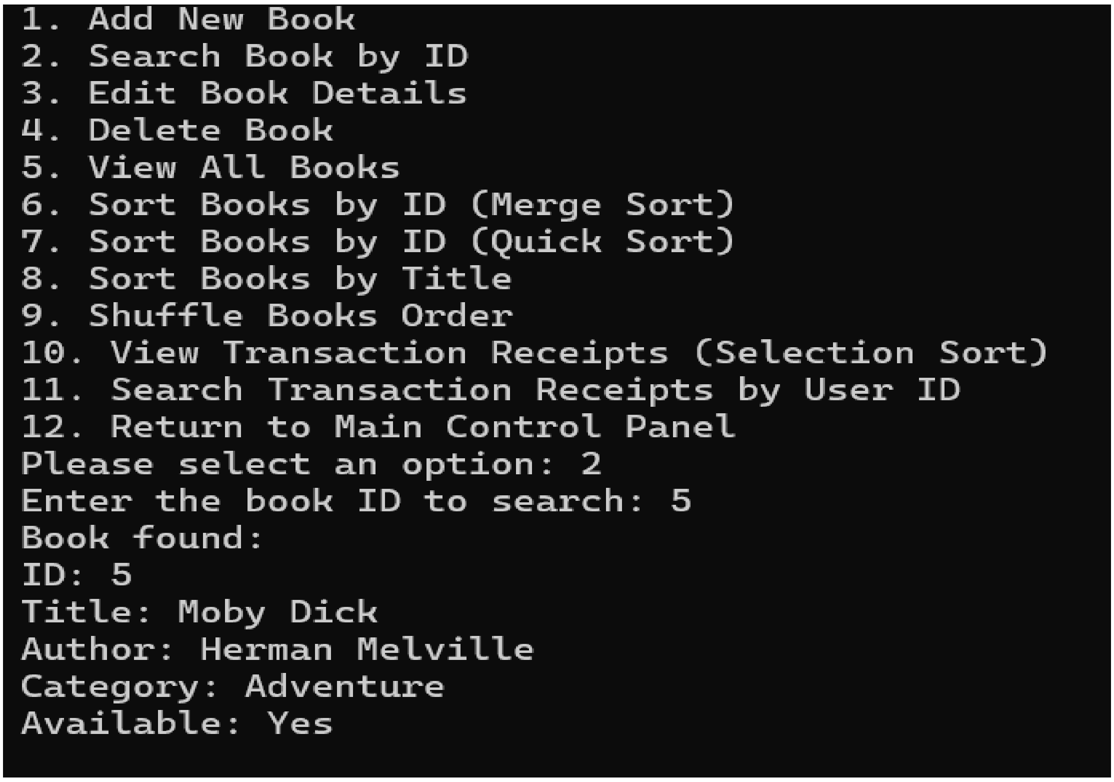
FIGURE 4 "AFTER MERGE sort",



```
12. Return to Main Control Panel
Please select an option: 7
Books sorted by ID using Quick Sort.

===== Admin Control Panel =====
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 5
List of all books:
ID: 1, Title: To Kill a Mockingbird, Author: Harper Lee, Category: Fiction, Available: Yes
ID: 2, Title: 1984, Author: George Orwell, Category: Dystopian, Available: Yes
ID: 3, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Category: Fiction, Available: Yes
ID: 4, Title: Test, Author: Test, Category: General, Available: Yes
ID: 5, Title: Moby Dick, Author: Herman Melville, Category: Adventure, Available: Yes
ID: 6, Title: Pride and Prejudice, Author: Jane Austen, Category: Classic, Available: Yes
ID: 7, Title: Jane Eyre, Author: Charlotte Bronte, Category: Classic, Available: Yes
ID: 8, Title: Wuthering Heights, Author: Emily Bronte, Category: Classic, Available: Yes
ID: 9, Title: The Hobbit, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 10, Title: The Lord of the Rings, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 11, Title: Sapiens: A Brief History of Humankind, Author: Yuval Noah Harari, Category: Non-fiction, Available: Yes
ID: 12, Title: Thinking, Fast and Slow, Author: Daniel Kahneman, Category: Non-fiction, Available: Yes
ID: 13, Title: Love, Author: Yuchengzheng, Category: fiction, Available: Yes
ID: 14, Title: The Art of War, Author: Sun Tzu, Category: Strategy, Available: Yes
ID: 15, Title: Atomic Habits, Author: James Clear, Category: Self-help, Available: Yes
ID: 16, Title: The Subtle Art of Not Giving a F*ck, Author: Mark Manson, Category: Self-help, Available: Yes
ID: 17, Title: Clean Code, Author: Robert C. Martin, Category: Programming, Available: Yes
ID: 18, Title: Introduction to Algorithms, Author: Thomas H. Cormen, Category: Computer Science, Available: Yes
ID: 19, Title: A Brief History of Time, Author: Stephen Hawking, Category: Science, Available: Yes
ID: 20, Title: The Road, Author: Cormac McCarthy, Category: Dystopian, Available: Yes
ID: 21, Title: 122, Author: 122, Category: 122, Available: Yes
```

FIGURE 5 "AFTER QUICK sort")

When users search for a specific book by its ID, the system leverages **Binary Search** to swiftly locate the desired entry within the sorted list, which significantly reduces search time.

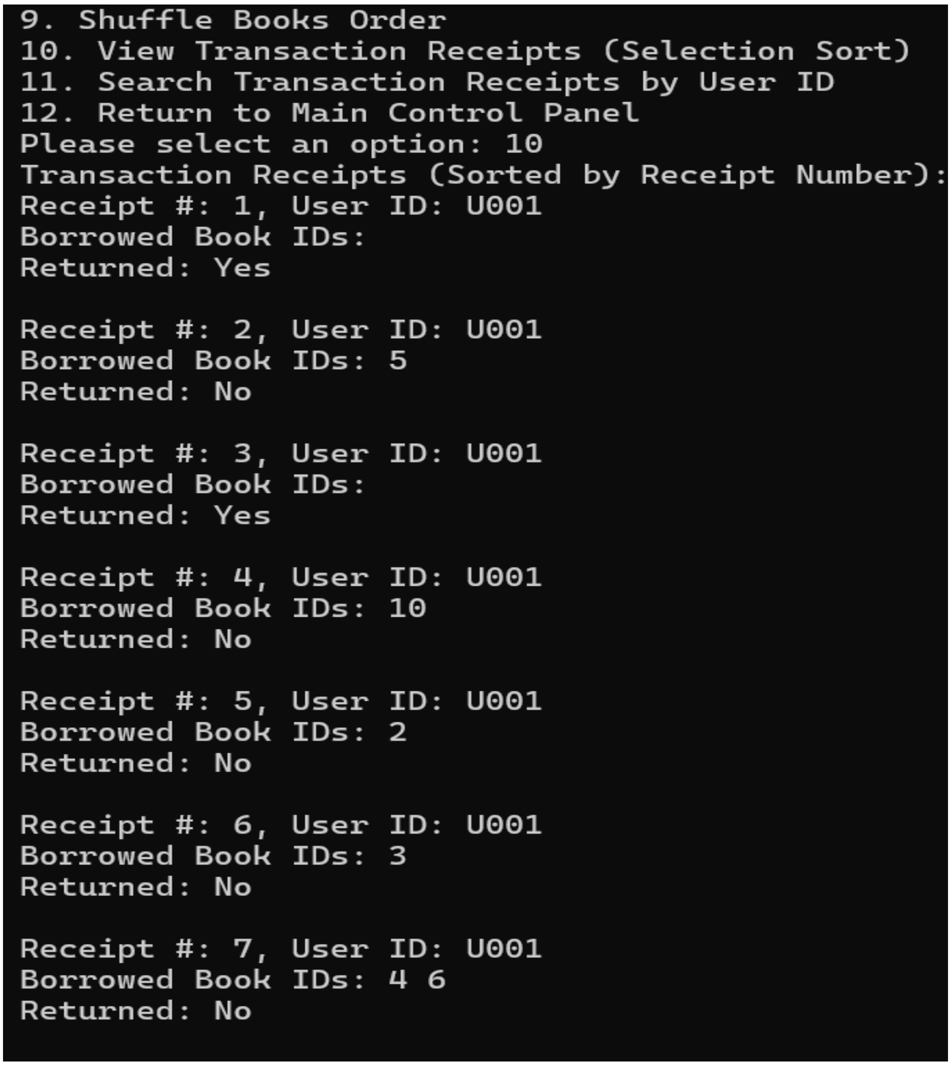


```
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 2
Enter the book ID to search: 5
Book found:
ID: 5
Title: Moby Dick
Author: Herman Melville
Category: Adventure
Available: Yes
```

(

FIGURE 6 "BINARY SEARCH")

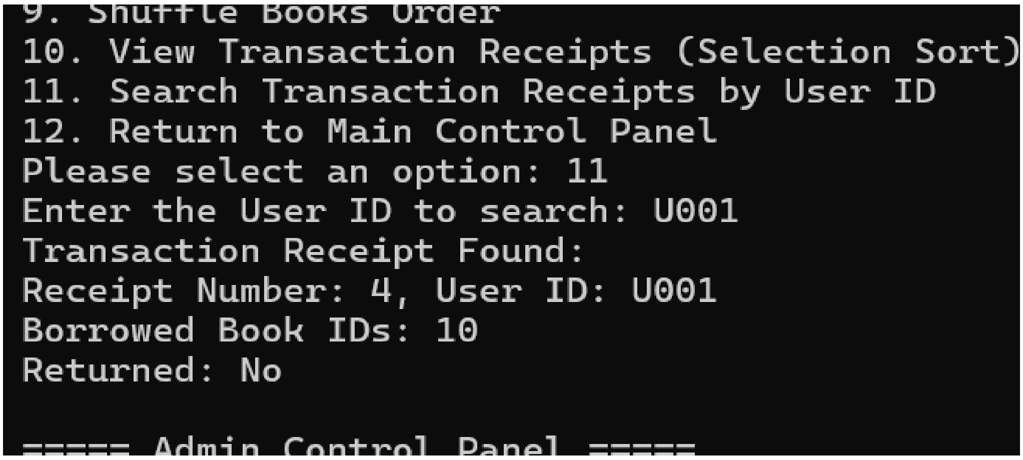
Similarly, transaction receipts are organized using **Selection Sort** based on receipt numbers, allowing for orderly record-keeping and easy access to transaction histories.



```
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 10
Transaction Receipts (Sorted by Receipt Number):
Receipt #: 1, User ID: U001
Borrowed Book IDs:
Returned: Yes

Receipt #: 2, User ID: U001
Borrowed Book IDs: 5
Returned: No

Receipt #: 3, User ID: U001
Borrowed Book IDs:
Returned: Yes

Receipt #: 4, User ID: U001
Borrowed Book IDs: 10
Returned: No

Receipt #: 5, User ID: U001
Borrowed Book IDs: 2
Returned: No

Receipt #: 6, User ID: U001
Borrowed Book IDs: 3
Returned: No

Receipt #: 7, User ID: U001
Borrowed Book IDs: 4 6
Returned: No
```

(FIGURE 7 "SELECTION SORT")

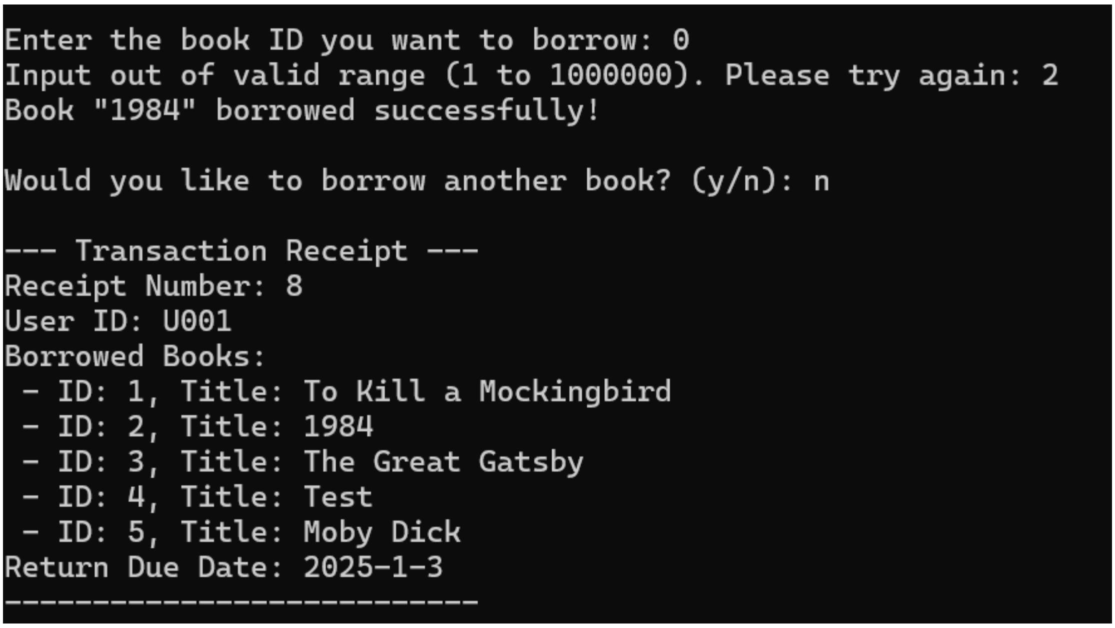
Furthermore, to display a user's borrowing history, the system implements **Linear Search**, which sequentially scans through transaction records to compile all relevant entries for the user.



```
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 11
Enter the User ID to search: U001
Transaction Receipt Found:
Receipt Number: 4, User ID: U001
Borrowed Book IDs: 10
Returned: No

===== Admin Control Panel =====
```

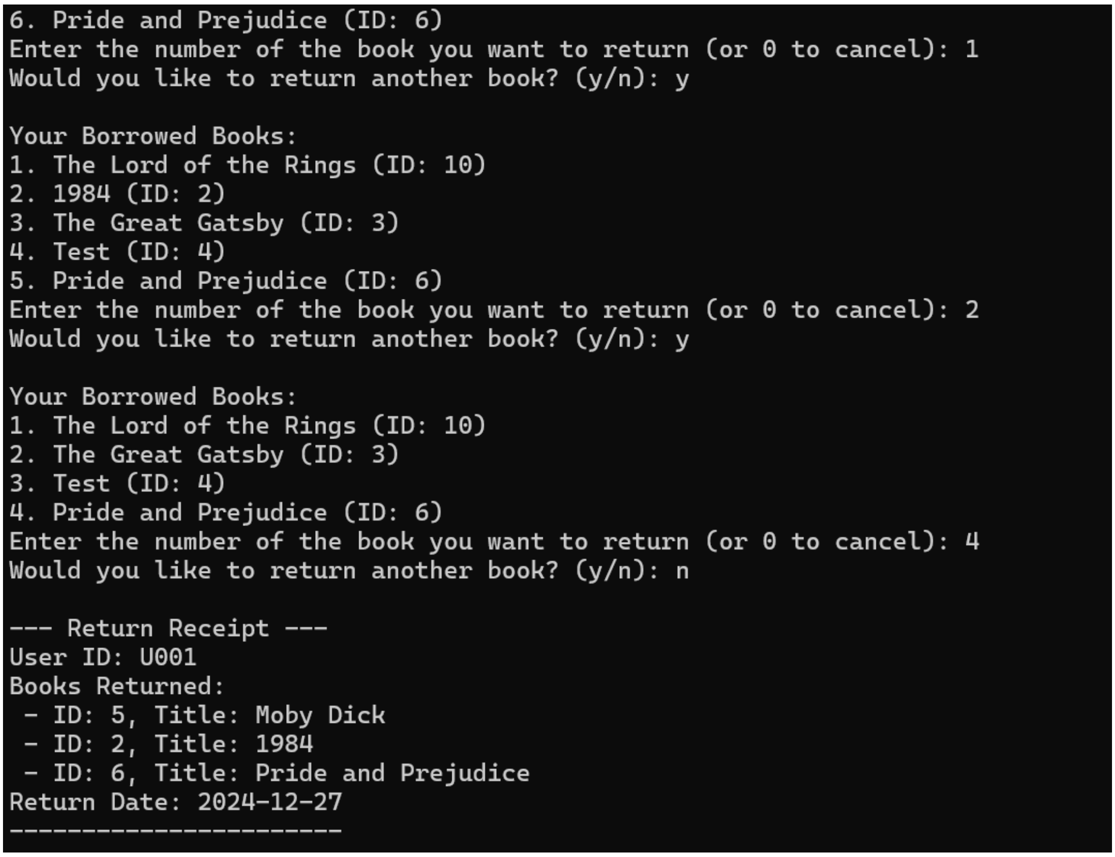
(FIGURE 8 "LINEAR SEARCH")

we also applied quick sort of borrowed book IDs, in order to finally show receipts for borrowed books or returned books.(



```
Enter the book ID you want to borrow: 0
Input out of valid range (1 to 1000000). Please try again: 2
Book "1984" borrowed successfully!

Would you like to borrow another book? (y/n): n

--- Transaction Receipt ---
Receipt Number: 8
User ID: U001
Borrowed Books:
 - ID: 1, Title: To Kill a Mockingbird
 - ID: 2, Title: 1984
 - ID: 3, Title: The Great Gatsby
 - ID: 4, Title: Test
 - ID: 5, Title: Moby Dick
Return Due Date: 2025-1-3
---------------------------
```

FIGURE 10 "QUICK SORT FOR BORROW RECEIPT",



```
6. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 1
Would you like to return another book? (y/n): y

Your Borrowed Books:
1. The Lord of the Rings (ID: 10)
2. 1984 (ID: 2)
3. The Great Gatsby (ID: 3)
4. Test (ID: 4)
5. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 2
Would you like to return another book? (y/n): y

Your Borrowed Books:
1. The Lord of the Rings (ID: 10)
2. The Great Gatsby (ID: 3)
3. Test (ID: 4)
4. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 4
Would you like to return another book? (y/n): n

--- Return Receipt ---
User ID: U001
Books Returned:
 - ID: 5, Title: Moby Dick
 - ID: 2, Title: 1984
 - ID: 6, Title: Pride and Prejudice
Return Date: 2024-12-27
-----------------------
```

FIGURE 9 "QUICK SORT FOR RETURN RECEIPT")

These algorithmic implementations collectively and ensures that the system operates efficiently, providing quick access to information and maintaining organized data structures for both administrators and users.

6. Error Handling Approach and Challenges

6.1 Error Handling Approach

File Operations:

The system verifies that files (e.g., books.txt, receipts.txt) are accessible before performing any read or write operations. If a file cannot be opened due to issues like missing files or insufficient permissions, a detailed error message will be displayed on the console and recorded in error_log.txt.

A retry mechanism is implemented for saving files. This allows the user to fix the issue like provide the correct file path or permissions and attempt the operation again before the system gives up loading.

User Input Validation:

We also use input validation to ensure the accuracy of user data. For example, the input format of user IDs must be some strings like “U001”, and book IDs must be valid integers within the specified range. If invalid input is detected, an error message will be displayed, reminding the user to re-enter the data until it meets the required input format.

Logical Operation Validation:

We designed logic check when a user borrowing or returning books. For instance, the system ensures users cannot borrow books that are unavailable or return books they have not borrowed.

A rollback mechanism is implemented to maintain data consistency. For example, if an error occurs during a book borrowing transaction, the system reverts all changes (e.g., resetting book availability and removing incomplete receipts).

Logging:

A logging system records critical errors in an error_log.txt and error.txt file. Each entry includes a timestamp and a detailed error description, in that case developers can diagnose issues efficiently.

6.2 Challenges Faced

Limited Understanding of Error Handling:

As a beginning developer, we initially overlooked many error scenarios, such as file absence or invalid user inputs. This resulted in many compiling errors during early testing.

The use of C++ features like try-catch blocks and error codes was challenging for us, so we need to take extra time to learn the relevant methods by ourselves.

Handling Complex Scenarios:

Implementing rollback logic was quite challenging. For example, when something goes wrong, we need to immediately undo the book’s borrowing status and delete the related receipt records. This requires different parts of the system to work closely together to prevent errors. Additionally, adding retry options for file operations means we have to repeatedly check how functions interact and depend on each other, which makes the process even more complicated.

Debugging Difficulties:

Error handling logic was scattered across multiple .cpp files, which makes it hard to track the flow of errors in the system initially. And some of us cannot use Git to manage the version of our codes, so without a centralized error-reporting and recording mechanism, debugging became time-consuming.

7. Appendix

7.1 Reference:

[1] Sedgewick, R. (2002). *Algorithms in C, Part 5: Graph algorithms*. Addison-Wesley. Retrieved from https://www.pearson.com/us/higher-education/program/Sedgewick-Algorithms-in-C-Parts-1-4-3rd-Edition/PGM199497.html

[2] Weiss, M. A. (2013). *Data structures and algorithm analysis in C++* (4th ed.). Pearson. Retrieved from https://www.pearson.com/us/higher-education/program/Weiss-Data-Structures-and-Algorithm-Analysis-in-C-4th-Edition/PGM1070418.html

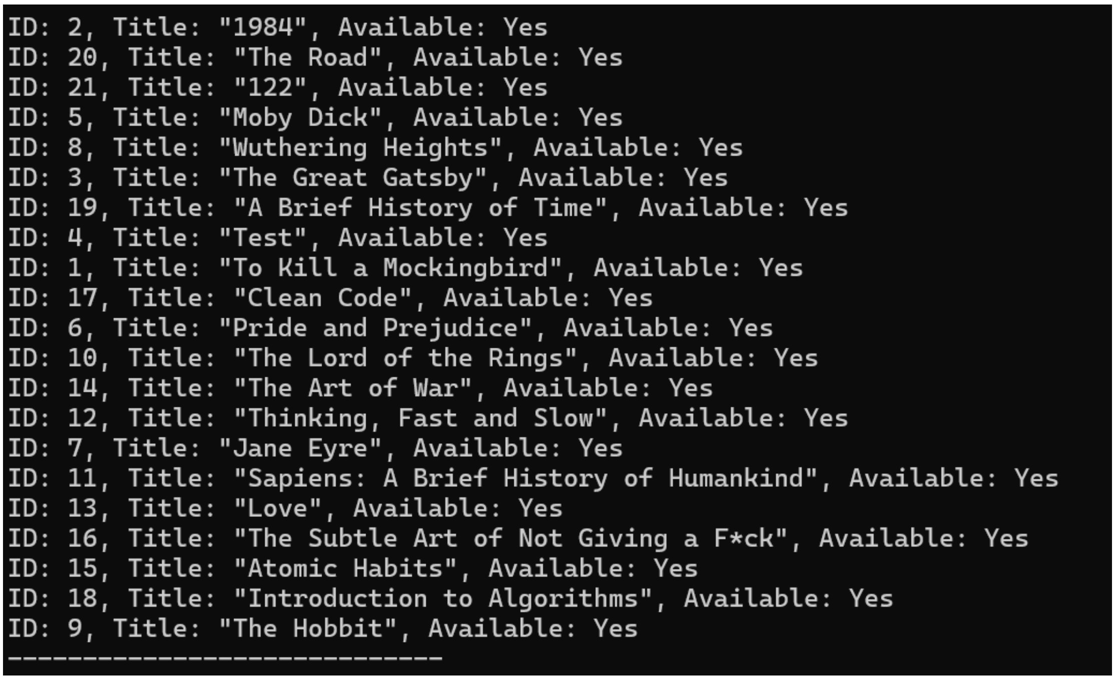
[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press. Retrieved from https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/

[4] Govindswamy, S. (n.d.). *Library Management System* [Computer software]. GitHub. Retrieved from https://github.com/govindswamy26/Library-Management-System

[5] GeeksforGeeks. (2024). *Bubble sort: How it works and implementation in C++*. Retrieved from https://www.geeksforgeeks.org/bubble-sort/

[6] Patel, J. (2023). Simplified sorting algorithms for beginners. *Journal of Computer Science Education, 12*(3), 45–50.

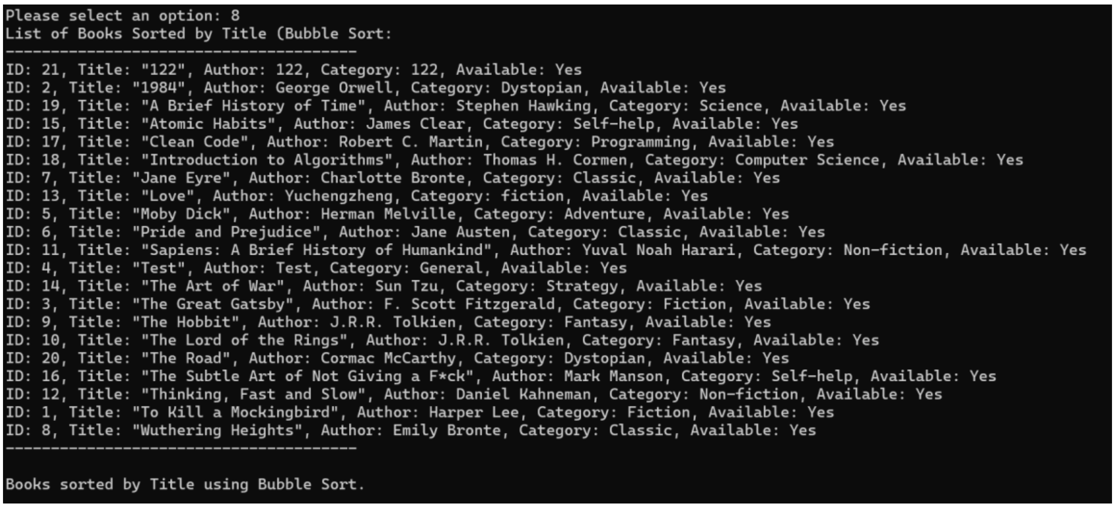
[7] Wirth, N. (1976). *Algorithms + data structures = programs*. Prentice Hall.



```
ID: 2, Title: "1984", Available: Yes
ID: 20, Title: "The Road", Available: Yes
ID: 21, Title: "122", Available: Yes
ID: 5, Title: "Moby Dick", Available: Yes
ID: 8, Title: "Wuthering Heights", Available: Yes
ID: 3, Title: "The Great Gatsby", Available: Yes
ID: 19, Title: "A Brief History of Time", Available: Yes
ID: 4, Title: "Test", Available: Yes
ID: 1, Title: "To Kill a Mockingbird", Available: Yes
ID: 17, Title: "Clean Code", Available: Yes
ID: 6, Title: "Pride and Prejudice", Available: Yes
ID: 10, Title: "The Lord of the Rings", Available: Yes
ID: 14, Title: "The Art of War", Available: Yes
ID: 12, Title: "Thinking, Fast and Slow", Available: Yes
ID: 7, Title: "Jane Eyre", Available: Yes
ID: 11, Title: "Sapiens: A Brief History of Humankind", Available: Yes
ID: 13, Title: "Love", Available: Yes
ID: 16, Title: "The Subtle Art of Not Giving a F*ck", Available: Yes
ID: 15, Title: "Atomic Habits", Available: Yes
ID: 18, Title: "Introduction to Algorithms", Available: Yes
ID: 9, Title: "The Hobbit", Available: Yes
-----------------------------
```

[8]

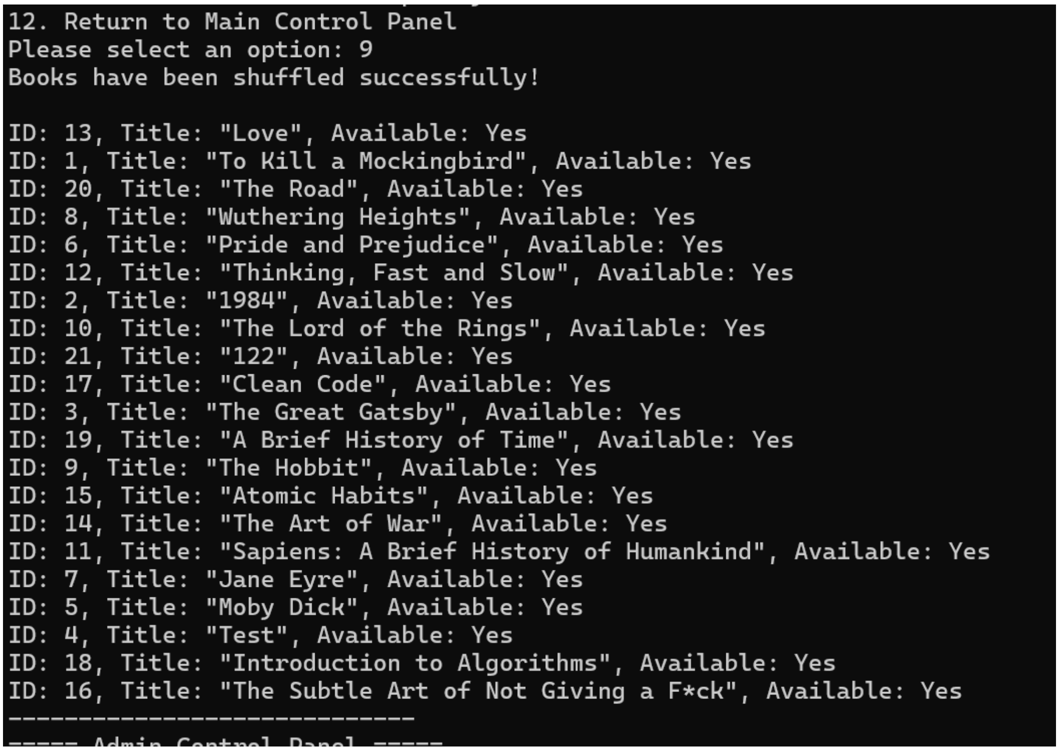
Figure 1 "Before Bubble sort"



```
Please select an option: 8
List of Books Sorted by Title (Bubble Sort:
---------------------------------------
ID: 21, Title: "122", Author: 122, Category: 122, Available: Yes
ID: 2, Title: "1984", Author: George Orwell, Category: Dystopian, Available: Yes
ID: 19, Title: "A Brief History of Time", Author: Stephen Hawking, Category: Science, Available: Yes
ID: 15, Title: "Atomic Habits", Author: James Clear, Category: Self-help, Available: Yes
ID: 17, Title: "Clean Code", Author: Robert C. Martin, Category: Programming, Available: Yes
ID: 18, Title: "Introduction to Algorithms", Author: Thomas H. Cormen, Category: Computer Science, Available: Yes
ID: 7, Title: "Jane Eyre", Author: Charlotte Bronte, Category: Classic, Available: Yes
ID: 13, Title: "Love", Author: Yuchengzheng, Category: fiction, Available: Yes
ID: 5, Title: "Moby Dick", Author: Herman Melville, Category: Adventure, Available: Yes
ID: 6, Title: "Pride and Prejudice", Author: Jane Austen, Category: Classic, Available: Yes
ID: 11, Title: "Sapiens: A Brief History of Humankind", Author: Yuval Noah Harari, Category: Non-fiction, Available: Yes
ID: 4, Title: "Test", Author: Test, Category: General, Available: Yes
ID: 14, Title: "The Art of War", Author: Sun Tzu, Category: Strategy, Available: Yes
ID: 3, Title: "The Great Gatsby", Author: F. Scott Fitzgerald, Category: Fiction, Available: Yes
ID: 9, Title: "The Hobbit", Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 10, Title: "The Lord of the Rings", Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 20, Title: "The Road", Author: Cormac McCarthy, Category: Dystopian, Available: Yes
ID: 16, Title: "The Subtle Art of Not Giving a F*ck", Author: Mark Manson, Category: Self-help, Available: Yes
ID: 12, Title: "Thinking, Fast and Slow", Author: Daniel Kahneman, Category: Non-fiction, Available: Yes
ID: 1, Title: "To Kill a Mockingbird", Author: Harper Lee, Category: Fiction, Available: Yes
ID: 8, Title: "Wuthering Heights", Author: Emily Bronte, Category: Classic, Available: Yes
---------------------------------------

Books sorted by Title using Bubble Sort.
```

[9]

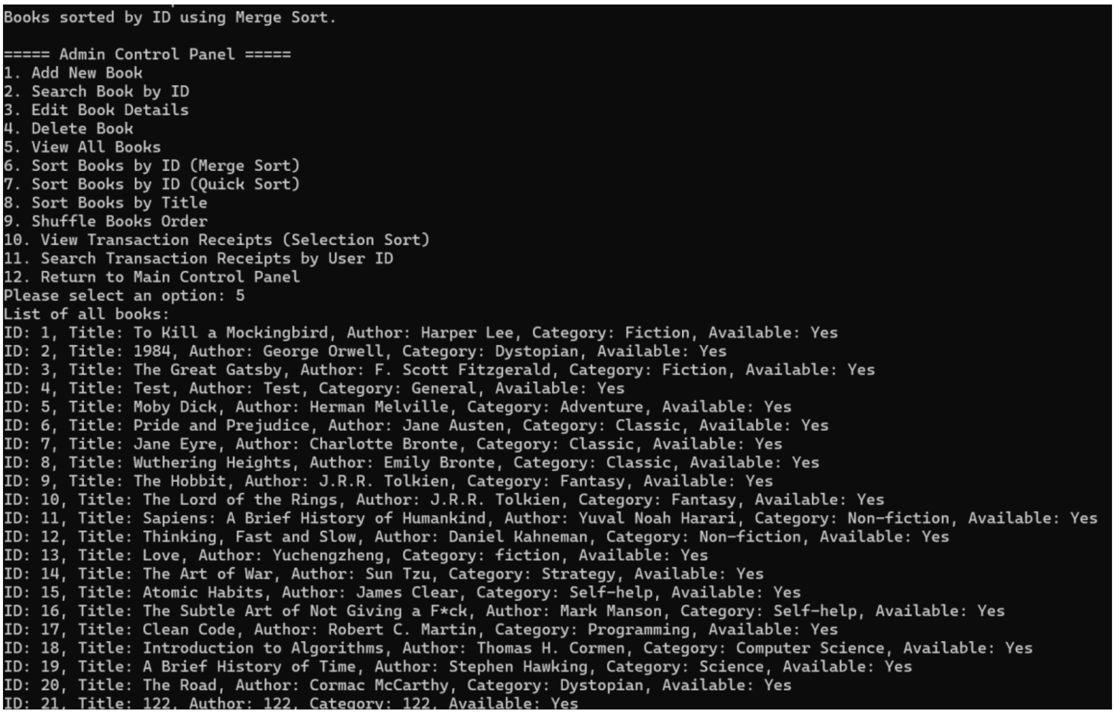
Figure 2"After Bubble sort"



```
12. Return to Main Control Panel
Please select an option: 9
Books have been shuffled successfully!

ID: 13, Title: "Love", Available: Yes
ID: 1, Title: "To Kill a Mockingbird", Available: Yes
ID: 20, Title: "The Road", Available: Yes
ID: 8, Title: "Wuthering Heights", Available: Yes
ID: 6, Title: "Pride and Prejudice", Available: Yes
ID: 12, Title: "Thinking, Fast and Slow", Available: Yes
ID: 2, Title: "1984", Available: Yes
ID: 10, Title: "The Lord of the Rings", Available: Yes
ID: 21, Title: "122", Available: Yes
ID: 17, Title: "Clean Code", Available: Yes
ID: 3, Title: "The Great Gatsby", Available: Yes
ID: 19, Title: "A Brief History of Time", Available: Yes
ID: 9, Title: "The Hobbit", Available: Yes
ID: 15, Title: "Atomic Habits", Available: Yes
ID: 14, Title: "The Art of War", Available: Yes
ID: 11, Title: "Sapiens: A Brief History of Humankind", Available: Yes
ID: 7, Title: "Jane Eyre", Available: Yes
ID: 5, Title: "Moby Dick", Available: Yes
ID: 4, Title: "Test", Available: Yes
ID: 18, Title: "Introduction to Algorithms", Available: Yes
ID: 16, Title: "The Subtle Art of Not Giving a F*ck", Available: Yes
-----------------------------
===== Admin Control Panel =====
```

[10]

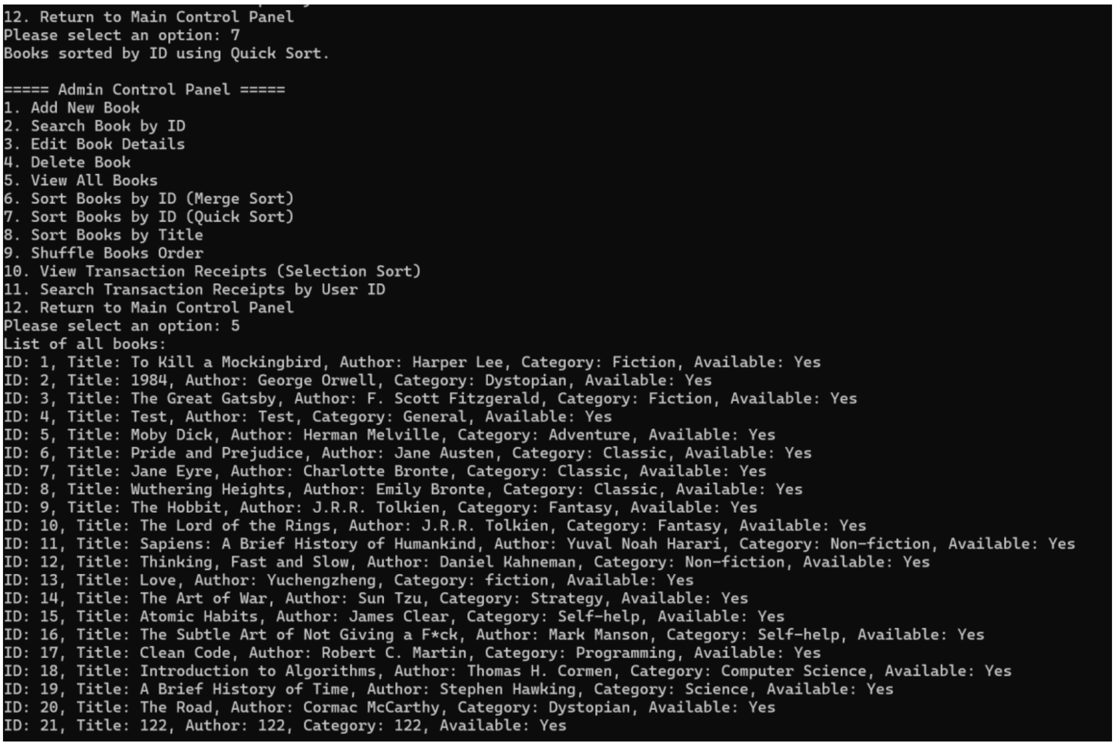
Figure 3 "Before sorting"



```
Books sorted by ID using Merge Sort.

===== Admin Control Panel =====
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 5
List of all books:
ID: 1, Title: To Kill a Mockingbird, Author: Harper Lee, Category: Fiction, Available: Yes
ID: 2, Title: 1984, Author: George Orwell, Category: Dystopian, Available: Yes
ID: 3, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Category: Fiction, Available: Yes
ID: 4, Title: Test, Author: Test, Category: General, Available: Yes
ID: 5, Title: Moby Dick, Author: Herman Melville, Category: Adventure, Available: Yes
ID: 6, Title: Pride and Prejudice, Author: Jane Austen, Category: Classic, Available: Yes
ID: 7, Title: Jane Eyre, Author: Charlotte Bronte, Category: Classic, Available: Yes
ID: 8, Title: Wuthering Heights, Author: Emily Bronte, Category: Classic, Available: Yes
ID: 9, Title: The Hobbit, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 10, Title: The Lord of the Rings, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 11, Title: Sapiens: A Brief History of Humankind, Author: Yuval Noah Harari, Category: Non-fiction, Available: Yes
ID: 12, Title: Thinking, Fast and Slow, Author: Daniel Kahneman, Category: Non-fiction, Available: Yes
ID: 13, Title: Love, Author: Yuchengzheng, Category: fiction, Available: Yes
ID: 14, Title: The Art of War, Author: Sun Tzu, Category: Strategy, Available: Yes
ID: 15, Title: Atomic Habits, Author: James Clear, Category: Self-help, Available: Yes
ID: 16, Title: The Subtle Art of Not Giving a F*ck, Author: Mark Manson, Category: Self-help, Available: Yes
ID: 17, Title: Clean Code, Author: Robert C. Martin, Category: Programming, Available: Yes
ID: 18, Title: Introduction to Algorithms, Author: Thomas H. Cormen, Category: Computer Science, Available: Yes
ID: 19, Title: A Brief History of Time, Author: Stephen Hawking, Category: Science, Available: Yes
ID: 20, Title: The Road, Author: Cormac McCarthy, Category: Dystopian, Available: Yes
ID: 21, Title: 122, Author: 122, Category: 122, Available: Yes
```

[11]

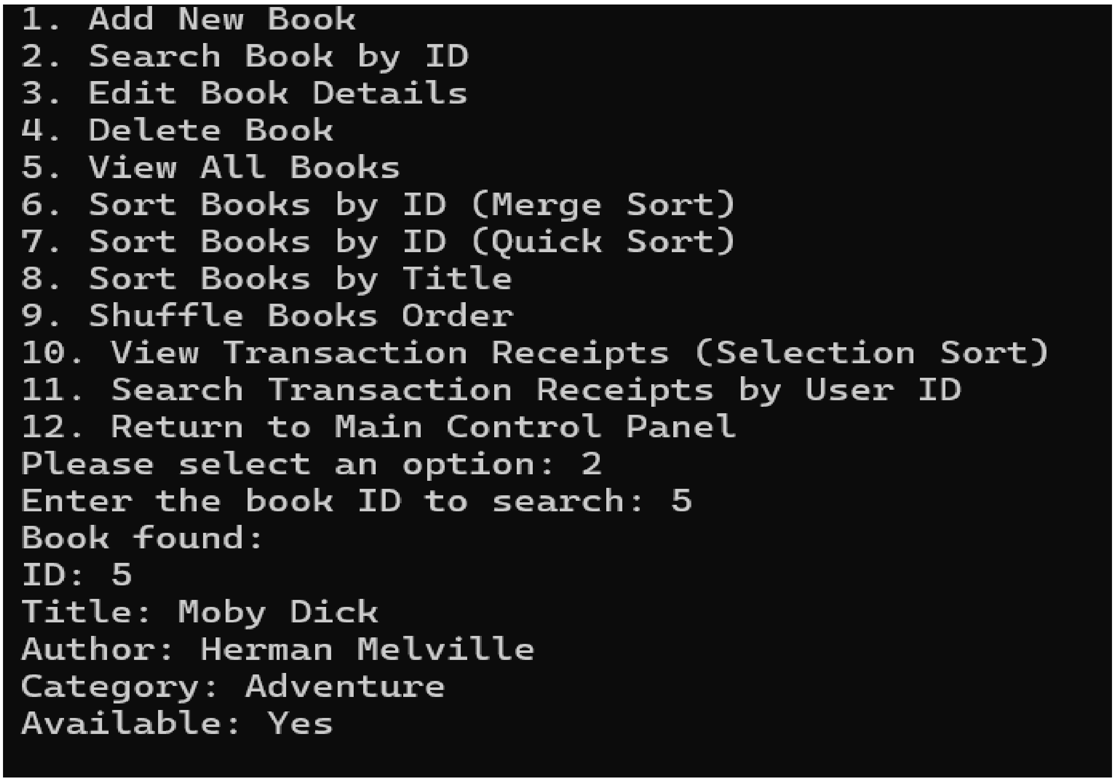
Figure 4 "After merge sort"



```
12. Return to Main Control Panel
Please select an option: 7
Books sorted by ID using Quick Sort.

===== Admin Control Panel =====
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 5
List of all books:
ID: 1, Title: To Kill a Mockingbird, Author: Harper Lee, Category: Fiction, Available: Yes
ID: 2, Title: 1984, Author: George Orwell, Category: Dystopian, Available: Yes
ID: 3, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Category: Fiction, Available: Yes
ID: 4, Title: Test, Author: Test, Category: General, Available: Yes
ID: 5, Title: Moby Dick, Author: Herman Melville, Category: Adventure, Available: Yes
ID: 6, Title: Pride and Prejudice, Author: Jane Austen, Category: Classic, Available: Yes
ID: 7, Title: Jane Eyre, Author: Charlotte Bronte, Category: Classic, Available: Yes
ID: 8, Title: Wuthering Heights, Author: Emily Bronte, Category: Classic, Available: Yes
ID: 9, Title: The Hobbit, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 10, Title: The Lord of the Rings, Author: J.R.R. Tolkien, Category: Fantasy, Available: Yes
ID: 11, Title: Sapiens: A Brief History of Humankind, Author: Yuval Noah Harari, Category: Non-fiction, Available: Yes
ID: 12, Title: Thinking, Fast and Slow, Author: Daniel Kahneman, Category: Non-fiction, Available: Yes
ID: 13, Title: Love, Author: Yuchengzheng, Category: fiction, Available: Yes
ID: 14, Title: The Art of War, Author: Sun Tzu, Category: Strategy, Available: Yes
ID: 15, Title: Atomic Habits, Author: James Clear, Category: Self-help, Available: Yes
ID: 16, Title: The Subtle Art of Not Giving a F*ck, Author: Mark Manson, Category: Self-help, Available: Yes
ID: 17, Title: Clean Code, Author: Robert C. Martin, Category: Programming, Available: Yes
ID: 18, Title: Introduction to Algorithms, Author: Thomas H. Cormen, Category: Computer Science, Available: Yes
ID: 19, Title: A Brief History of Time, Author: Stephen Hawking, Category: Science, Available: Yes
ID: 20, Title: The Road, Author: Cormac McCarthy, Category: Dystopian, Available: Yes
ID: 21, Title: 122, Author: 122, Category: 122, Available: Yes
```

[12]

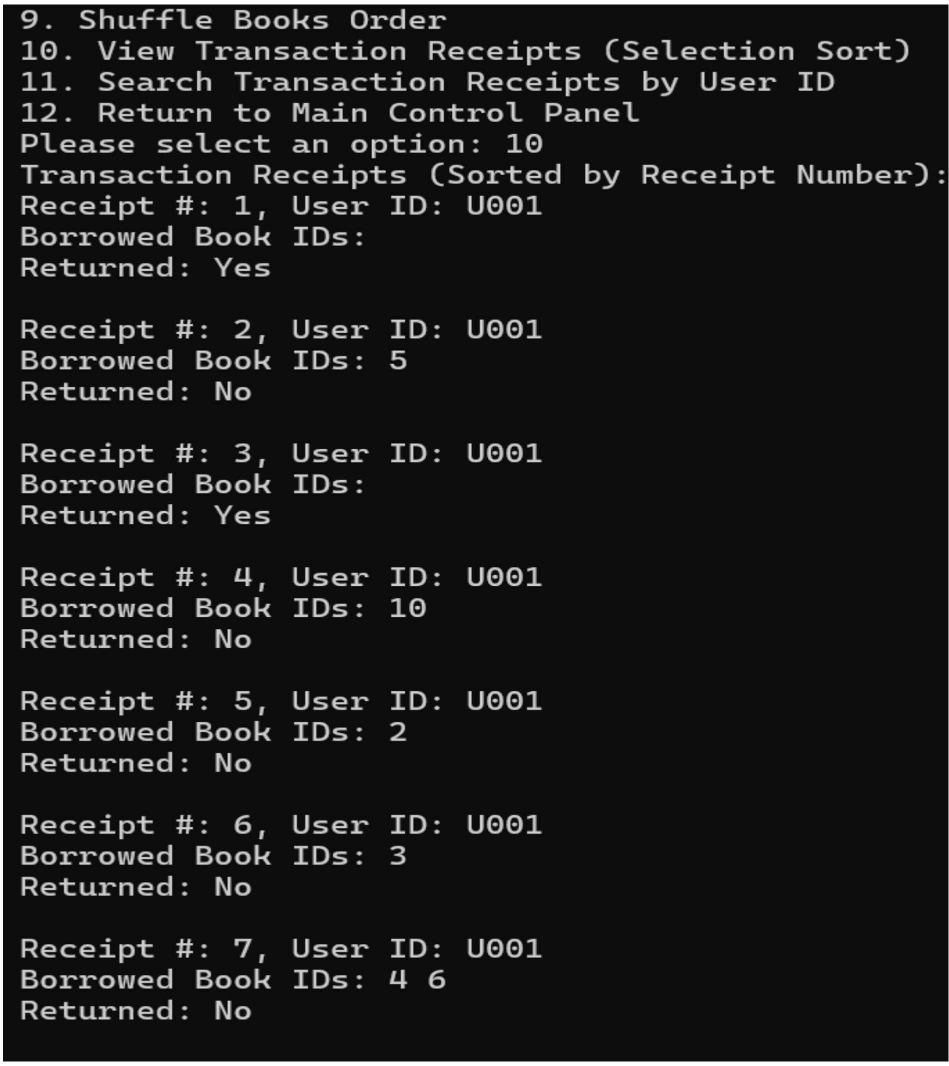
Figure 5 "After quick sort"



```
1. Add New Book
2. Search Book by ID
3. Edit Book Details
4. Delete Book
5. View All Books
6. Sort Books by ID (Merge Sort)
7. Sort Books by ID (Quick Sort)
8. Sort Books by Title
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 2
Enter the book ID to search: 5
Book found:
ID: 5
Title: Moby Dick
Author: Herman Melville
Category: Adventure
Available: Yes
```

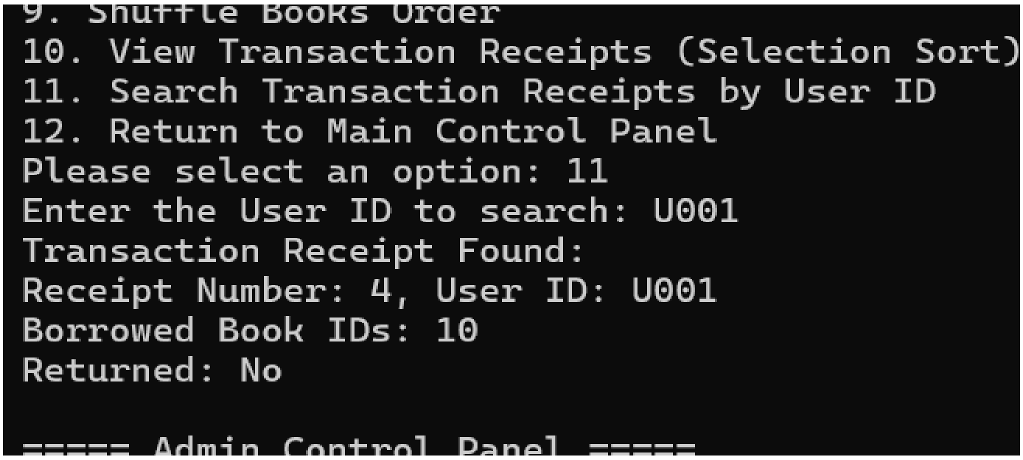
[13]

Figure 6 "Binary search"

[14] 

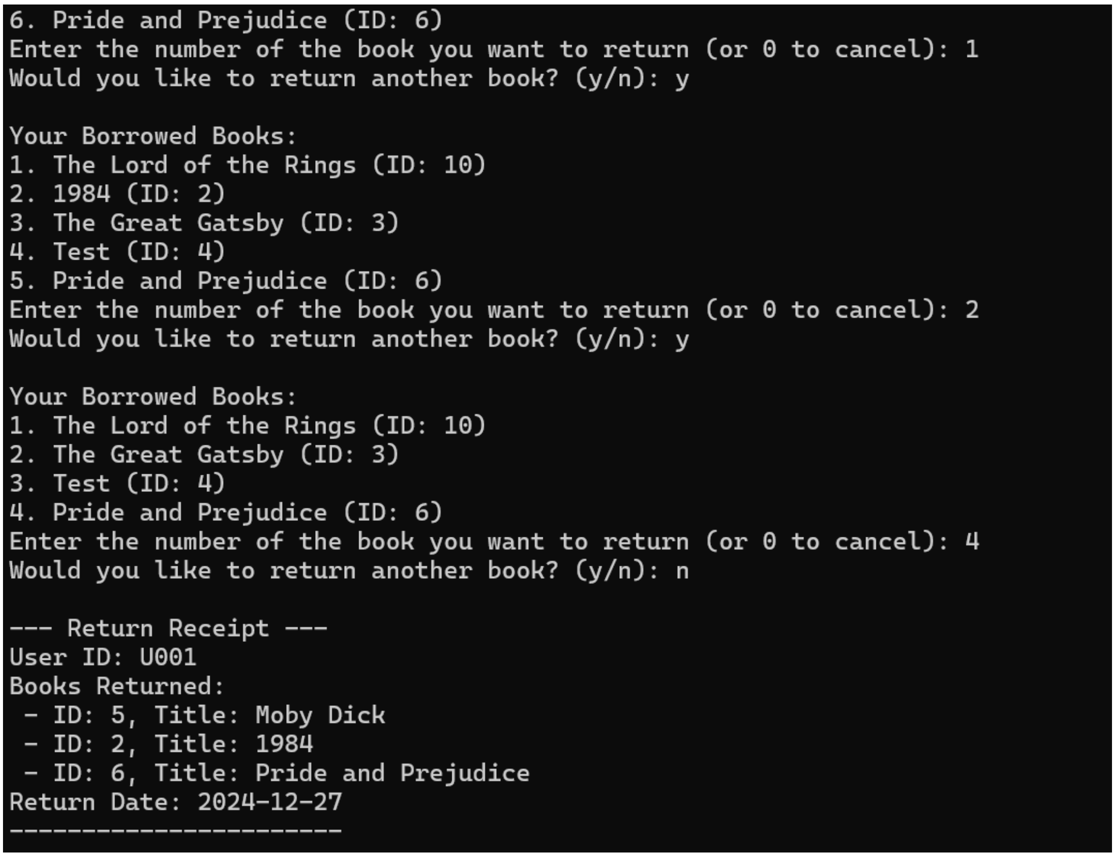
```
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 10
Transaction Receipts (Sorted by Receipt Number):
Receipt #: 1, User ID: U001
Borrowed Book IDs:
Returned: Yes

Receipt #: 2, User ID: U001
Borrowed Book IDs: 5
Returned: No

Receipt #: 3, User ID: U001
Borrowed Book IDs:
Returned: Yes

Receipt #: 4, User ID: U001
Borrowed Book IDs: 10
Returned: No

Receipt #: 5, User ID: U001
Borrowed Book IDs: 2
Returned: No

Receipt #: 6, User ID: U001
Borrowed Book IDs: 3
Returned: No

Receipt #: 7, User ID: U001
Borrowed Book IDs: 4 6
Returned: No
```

Figure 7 "Selection Sort"

[15] 

```
9. Shuffle Books Order
10. View Transaction Receipts (Selection Sort)
11. Search Transaction Receipts by User ID
12. Return to Main Control Panel
Please select an option: 11
Enter the User ID to search: U001
Transaction Receipt Found:
Receipt Number: 4, User ID: U001
Borrowed Book IDs: 10
Returned: No

===== Admin Control Panel =====
```

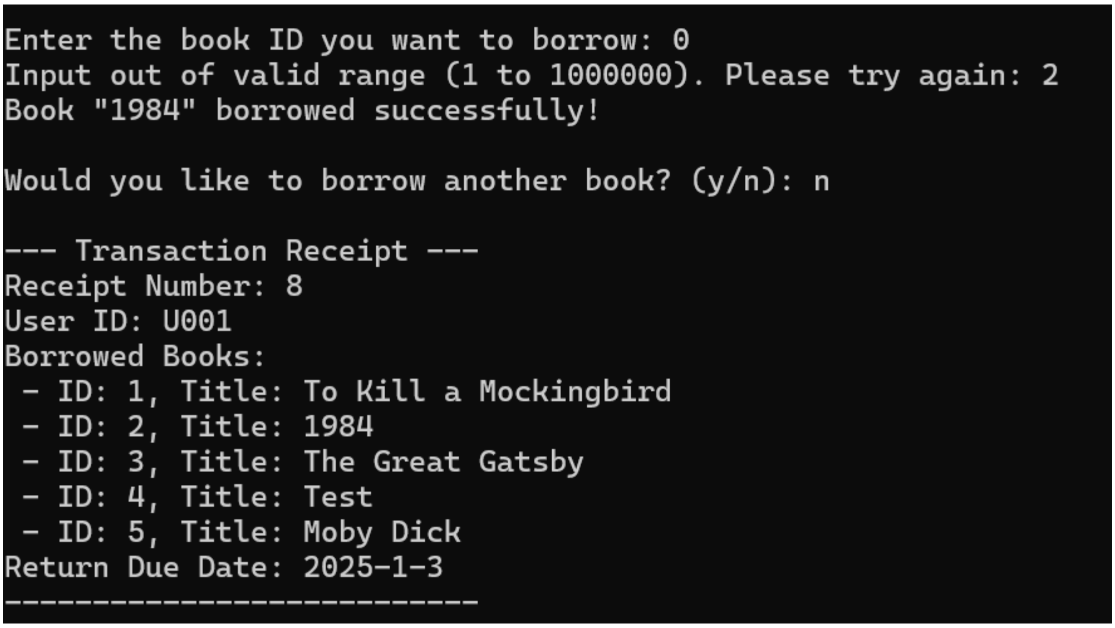
Figure 8 "Linear search"



```
6. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 1
Would you like to return another book? (y/n): y

Your Borrowed Books:
1. The Lord of the Rings (ID: 10)
2. 1984 (ID: 2)
3. The Great Gatsby (ID: 3)
4. Test (ID: 4)
5. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 2
Would you like to return another book? (y/n): y

Your Borrowed Books:
1. The Lord of the Rings (ID: 10)
2. The Great Gatsby (ID: 3)
3. Test (ID: 4)
4. Pride and Prejudice (ID: 6)
Enter the number of the book you want to return (or 0 to cancel): 4
Would you like to return another book? (y/n): n

--- Return Receipt ---
User ID: U001
Books Returned:
 - ID: 5, Title: Moby Dick
 - ID: 2, Title: 1984
 - ID: 6, Title: Pride and Prejudice
Return Date: 2024-12-27
----------------------
```

[16]

Figure 9 "Quick sort for return receipt"

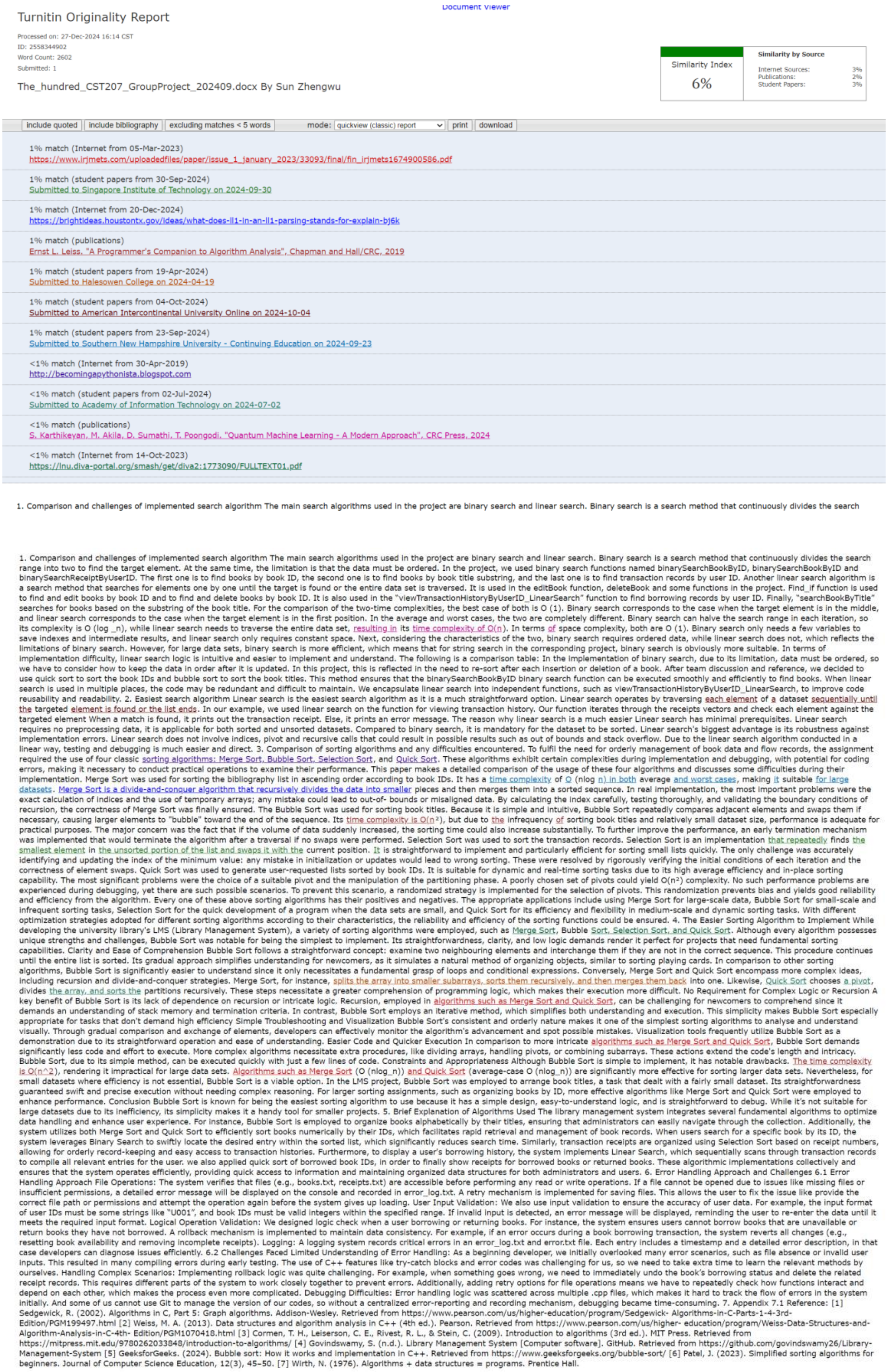


```
Enter the book ID you want to borrow: 0
Input out of valid range (1 to 1000000). Please try again: 2
Book "1984" borrowed successfully!

Would you like to borrow another book? (y/n): n

--- Transaction Receipt ---
Receipt Number: 8
User ID: U001
Borrowed Books:
 - ID: 1, Title: To Kill a Mockingbird
 - ID: 2, Title: 1984
 - ID: 3, Title: The Great Gatsby
 - ID: 4, Title: Test
 - ID: 5, Title: Moby Dick
Return Due Date: 2025-1-3
---------------------------
```

[17]

Figure 10 "Quick sort for borrow receipt"

7.2 Turnitin Report



Document Viewer

Turnitin Originality Report

Processed on: 27-Dec-2024 16:14 CST
ID: 2558344902
Word Count: 2602
Submitted: 1

The_hundred_CST207_GroupProject_202409.docx By Sun Zhengwu

Similarity Index	Similarity by Source	
6%	Internet Sources:	3%
	Publications:	2%
	Student Papers:	3%

1% match (Internet from 05-Mar-2023)
https://www.irjmets.com/uploadedfiles/paper/issue_1_january_2023/33093/final/fin_irjmets1674900586.pdf

1% match (student papers from 30-Sep-2024)
Submitted to Singapore Institute of Technology on 2024-09-30

1% match (Internet from 20-Dec-2024)
https://brightideas.houstontx.gov/ideas/what-does-ll1-in-an-ll1-parsing-stands-for-explain-bj6k

1% match (publications)
Ernst L. Leiss. "A Programmer's Companion to Algorithm Analysis", Chapman and Hall/CRC, 2019

1% match (student papers from 19-Apr-2024)
Submitted to Halesowen College on 2024-04-19

1% match (student papers from 04-Oct-2024)
Submitted to American Intercontinental University Online on 2024-10-04

1% match (student papers from 23-Sep-2024)
Submitted to Southern New Hampshire University - Continuing Education on 2024-09-23

<1% match (Internet from 30-Apr-2019)
http://becomingapythonista.blogspot.com

<1% match (student papers from 02-Jul-2024)
Submitted to Academy of Information Technology on 2024-07-02

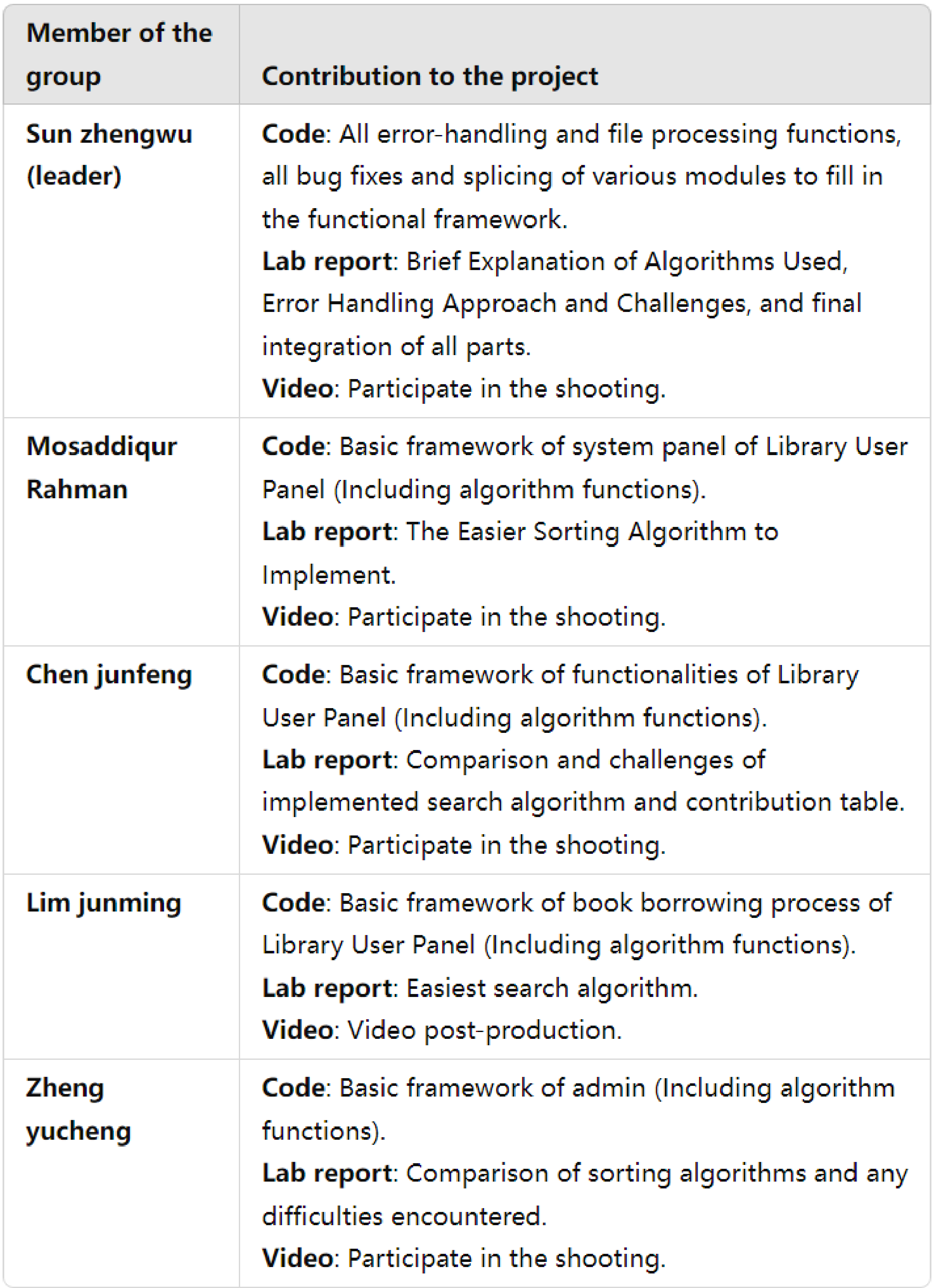
<1% match (publications)
S. Karthikeyan, M. Akila, D. Sumathi, T. Poongodi. "Quantum Machine Learning - A Modern Approach", CRC Press, 2024

<1% match (Internet from 14-Oct-2023)
https://lnu.diva-portal.org/smash/get/diva2:1773090/FULLTEXT01.pdf

1. Comparison and challenges of implemented search algorithm The main search algorithms used in the project are binary search and linear search. Binary search is a search method that continuously divides the search

1. Comparison and challenges of implemented search algorithm The main search algorithms used in the project are binary search and linear search. Binary search is a search method that continuously divides the search range into two to find the target element. At the same time, the limitation is that the data must be ordered. In the project, we used binary search functions named binarySearchBookByID, binarySearchBookByID and binarySearchReceiptByUserID. The first one is to find books by book ID, the second one is to find books by book title substring, and the last one is to find transaction records by user ID. Another linear search algorithm is a search method that searches for elements one by one until the target is found or the entire data set is traversed. It is used in the editBook function, deleteBook and some functions in the project. Find_if function is used to find and edit books by book ID and to find and delete books by book ID. It is also used in the "viewTransactionHistoryByUserID_LinearSearch" function to find borrowing records by user ID. Finally, "searchBookByTitle" searches for books based on the substring of the book title. For the comparison of the two-time complexities, the best case of both is O (1). Binary search corresponds to the case when the target element is in the middle, and linear search corresponds to the case when the target element is in the first position. In the average and worst cases, the two are completely different. Binary search can halve the search range in each iteration, so its complexity is O (log _n), while linear search needs to traverse the entire data set, resulting in its time complexity of O(n). In terms of space complexity, both are O (1). Binary search only needs a few variables to save indexes and intermediate results, and linear search only requires constant space. Next, considering the characteristics of the two, binary search requires ordered data, while linear search does not, which reflects the limitations of binary search. However, for large data sets, binary search is more efficient, which means that for string search in the corresponding project, binary search is obviously more suitable. In terms of implementation difficulty, linear search logic is intuitive and easier to implement and understand. The following is a comparison table: In the implementation of binary search, due to its limitation, data must be ordered, so we have to consider how to keep the data in order after it is updated. In this project, this is reflected in the need to re-sort after each insertion or deletion of a book. After team discussion and reference, we decided to use quick sort to sort the book IDs and bubble sort to sort the book titles. This method ensures that the binarySearchBookByID binary search function can be executed smoothly and efficiently to find books. When linear search is used in multiple places, the code may be redundant and difficult to maintain. We encapsulate linear search into independent functions, such as viewTransactionHistoryByUserID_LinearSearch, to improve code reusability and readability. 2. Easiest search algorithm Linear search is the easiest search algorithm as it is a much straightforward option. Linear search operates by traversing each element of a dataset sequentially until the targeted element is found or the list ends. In our example, we used linear search on the function for viewing transaction history. Our function iterates through the receipts vectors and check each element against the targeted element When a match is found, it prints out the transaction receipt. Else, it prints an error message. The reason why linear search is a much easier Linear search has minimal prerequisites. Linear search requires no preprocessing data, it is applicable for both sorted and unsorted datasets. Compared to binary search, it is mandatory for the dataset to be sorted. Linear search's biggest advantage is its robustness against implementation errors. Linear search does not involve indices, pivot and recursive calls that could result in possible results such as out of bounds and stack overflow. Due to the linear search algorithm conducted in a linear way, testing and debugging is much easier and direct. 3. Comparison of sorting algorithms and any difficulties encountered. To fulfil the need for orderly management of book data and flow records, the assignment required the use of four classic sorting algorithms: Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. These algorithms exhibit certain complexities during implementation and debugging, with potential for coding errors, making it necessary to conduct practical operations to examine their performance. This paper makes a detailed comparison of the usage of these four algorithms and discusses some difficulties during their implementation. Merge Sort was used for sorting the bibliography list in ascending order according to book IDs. It has a time complexity of O (nlog n) in both average and worst cases, making it suitable for large datasets. Merge Sort is a divide-and-conquer algorithm that recursively divides the data into smaller pieces and then merges them into a sorted sequence. In real implementation, the most important problems were the exact calculation of indices and the use of temporary arrays; any mistake could lead to out-of- bounds or misaligned data. By calculating the index carefully, testing thoroughly, and validating the boundary conditions of recursion, the correctness of Merge Sort was finally ensured. The Bubble Sort was used for sorting book titles. Because it is simple and intuitive, Bubble Sort repeatedly compares adjacent elements and swaps them if necessary, causing larger elements to "bubble" toward the end of the sequence. Its time complexity is $O(n^2)$, but due to the infrequency of sorting book titles and relatively small dataset size, performance is adequate for practical purposes. The major concern was the fact that if the volume of data suddenly increased, the sorting time could also increase substantially. To further improve the performance, an early termination mechanism was implemented that would terminate the algorithm after a traversal if no swaps were performed. Selection Sort was used to sort the transaction records. Selection Sort is an implementation that repeatedly finds the smallest element in the unsorted portion of the list and swaps it with the current position. It is straightforward to implement and particularly efficient for sorting small lists quickly. The only challenge was accurately identifying and updating the index of the minimum value: any mistake in initialization or updates would lead to wrong sorting. These were resolved by rigorously verifying the initial conditions of each iteration and the correctness of element swaps. Quick Sort was used to generate user-requested lists sorted by book IDs. It is suitable for dynamic and real-time sorting tasks due to its high average efficiency and in-place sorting capability. The most significant problems were the choice of a suitable pivot and the manipulation of the partitioning phase. A poorly chosen set of pivots could yield $O(n^2)$ complexity. No such performance problems are experienced during debugging, yet there are such possible scenarios. To prevent this scenario, a randomized strategy is implemented for the selection of pivots. This randomization prevents bias and yields good reliability and efficiency from the algorithm. Every one of these above sorting algorithms has their positives and negatives. The appropriate applications include using Merge Sort for large-scale data, Bubble Sort for small-scale and infrequent sorting tasks, Selection Sort for the quick development of a program when the data sets are small, and Quick Sort for its efficiency and flexibility in medium-scale and dynamic sorting tasks. With different optimization strategies adopted for different sorting algorithms according to their characteristics, the reliability and efficiency of the sorting functions could be ensured. 4. The Easier Sorting Algorithm to Implement While developing the university library's LMS (Library Management System), a variety of sorting algorithms were employed, such as Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. Although every algorithm possesses unique strengths and challenges, Bubble Sort was notable for being the simplest to implement. Its straightforwardness, clarity, and low logic demands render it perfect for projects that need fundamental sorting capabilities. Clarity and Ease of Comprehension Bubble Sort follows a straightforward concept: examine two neighbouring elements and interchange them if they are not in the correct sequence. This procedure continues until the entire list is sorted. Its gradual approach simplifies understanding for newcomers, as it simulates a natural method of organizing objects, similar to sorting playing cards. In comparison to other sorting algorithms, Bubble Sort is significantly easier to understand since it only necessitates a fundamental grasp of loops and conditional expressions. Conversely, Merge Sort and Quick Sort encompass more complex ideas, including recursion and divide-and-conquer strategies. Merge Sort, for instance, splits the array into smaller subarrays, sorts them recursively, and then merges them back into one. Likewise, Quick Sort chooses a pivot, divides the array, and sorts the partitions recursively. These steps necessitate a greater comprehension of programming logic, which makes their execution more difficult. No Requirement for Complex Logic or Recursion A key benefit of Bubble Sort is its lack of dependence on recursion or intricate logic. Recursion, employed in algorithms such as Merge Sort and Quick Sort, can be challenging for newcomers to comprehend since it demands an understanding of stack memory and termination criteria. In contrast, Bubble Sort employs an iterative method, which simplifies both understanding and execution. This simplicity makes Bubble Sort especially appropriate for tasks that don't demand high efficiency Simple Troubleshooting and Visualization Bubble Sort's consistent and orderly nature makes it one of the simplest sorting algorithms to analyse and understand visually. Through gradual comparison and exchange of elements, developers can effectively monitor the algorithm's advancement and spot possible mistakes. Visualization tools frequently utilize Bubble Sort as a demonstration due to its straightforward operation and ease of understanding. Easier Code and Quicker Execution In comparison to more intricate algorithms such as Merge Sort and Quick Sort, Bubble Sort demands significantly less code and effort to execute. More complex algorithms necessitate extra procedures, like dividing arrays, handling pivots, or combining subarrays. These actions extend the code's length and intricacy. Bubble Sort, due to its simple method, can be executed quickly with just a few lines of code. Constraints and Appropriateness Although Bubble Sort is simple to implement, it has notable drawbacks. The time complexity is O(n^2), rendering it impractical for large data sets. Algorithms such as Merge Sort (O (nlog_n)) and Quick Sort (average-case O (nlog_n)) are significantly more effective for sorting larger data sets. Nevertheless, for small datasets where efficiency is not essential, Bubble Sort is a viable option. In the LMS project, Bubble Sort was employed to arrange book titles, a task that dealt with a fairly small dataset. Its straightforwardness guaranteed swift and precise execution without needing complex reasoning. For larger sorting assignments, such as organizing books by ID, more effective algorithms like Merge Sort and Quick Sort were employed to enhance performance. Conclusion Bubble Sort is known for being the easiest sorting algorithm to use because it has a simple design, easy-to-understand logic, and is straightforward to debug. While it's not suitable for large datasets due to its inefficiency, its simplicity makes it a handy tool for smaller projects. 5. Brief Explanation of Algorithms Used The library management system integrates several fundamental algorithms to optimize data handling and enhance user experience. For instance, Bubble Sort is employed to organize books alphabetically by their titles, ensuring that administrators can easily navigate through the collection. Additionally, the system utilizes both Merge Sort and Quick Sort to efficiently sort books numerically by their IDs, which facilitates rapid retrieval and management of book records. When users search for a specific book by its ID, the system leverages Binary Search to swiftly locate the desired entry within the sorted list, which significantly reduces search time. Similarly, transaction receipts are organized using Selection Sort based on receipt numbers, allowing for orderly record-keeping and easy access to transaction histories. Furthermore, to display a user's borrowing history, the system implements Linear Search, which sequentially scans through transaction records to compile all relevant entries for the user. we also applied quick sort of borrowed book IDs, in order to finally show receipts for borrowed books or returned books. These algorithmic implementations collectively and ensures that the system operates efficiently, providing quick access to information and maintaining organized data structures for both administrators and users. 6. Error Handling Approach and Challenges 6.1 Error Handling Approach File Operations: The system verifies that files (e.g., books.txt, receipts.txt) are accessible before performing any read or write operations. If a file cannot be opened due to issues like missing files or insufficient permissions, a detailed error message will be displayed on the console and recorded in error_log.txt. A retry mechanism is implemented for saving files. This allows the user to fix the issue like provide the correct file path or permissions and attempt the operation again before the system gives up loading. User Input Validation: We also use input validation to ensure the accuracy of user data. For example, the input format of user IDs must be some strings like "U001", and book IDs must be valid integers within the specified range. If invalid input is detected, an error message will be displayed, reminding the user to re-enter the data until it meets the required input format. Logical Operation Validation: We designed logic check when a user borrowing or returning books. For instance, the system ensures users cannot borrow books that are unavailable or return books they have not borrowed. A rollback mechanism is implemented to maintain data consistency. For example, if an error occurs during a book borrowing transaction, the system reverts all changes (e.g., resetting book availability and removing incomplete receipts). Logging: A logging system records critical errors in an error_log.txt and error.txt file. Each entry includes a timestamp and a detailed error description, in that case developers can diagnose issues efficiently. 6.2 Challenges Faced Limited Understanding of Error Handling: As a beginning developer, we initially overlooked many error scenarios, such as file absence or invalid user inputs. This resulted in many compiling errors during early testing. The use of C++ features like try-catch blocks and error codes was challenging for us, so we need to take extra time to learn the relevant methods by ourselves. Handling Complex Scenarios: Implementing rollback logic was quite challenging. For example, when something goes wrong, we need to immediately undo the book's borrowing status and delete the related receipt records. This requires different parts of the system to work closely together to prevent errors. Additionally, adding retry options for file operations means we have to repeatedly check how functions interact and depend on each other, which makes the process even more complicated. Debugging Difficulties: Error handling logic was scattered across multiple .cpp files, which makes it hard to track the flow of errors in the system initially. And some of us cannot use Git to manage the version of our codes, so without a centralized error-reporting and recording mechanism, debugging became time-consuming. 7. Appendix 7.1 Reference: [1] Sedgewick, R. (2002). Algorithms in C, Part 5: Graph algorithms. Addison-Wesley. Retrieved from https://www.pearson.com/us/higher-education/program/Sedgewick- Algorithms-in-C-Parts-1-4-3rd-Edition/PGM199497.html [2] Weiss, M. A. (2013). Data structures and algorithm analysis in C++ (4th ed.). Pearson. Retrieved from https://www.pearson.com/us/higher- education/program/Weiss-Data-Structures-and-Algorithm-Analysis-in-C-4th- Edition/PGM1070418.html [3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press. Retrieved from https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/ [4] Govindswamy, S. (n.d.). Library Management System [Computer software]. GitHub. Retrieved from https://github.com/govindswamy26/Library-Management-System [5] GeeksforGeeks. (2024). Bubble sort: How it works and implementation in C++. Retrieved from https://www.geeksforgeeks.org/bubble-sort/ [6] Patel, J. (2023). Simplified sorting algorithms for beginners. Journal of Computer Science Education, 12(3), 45–50. [7] Wirth, N. (1976). Algorithms + data structures = programs. Prentice Hall.

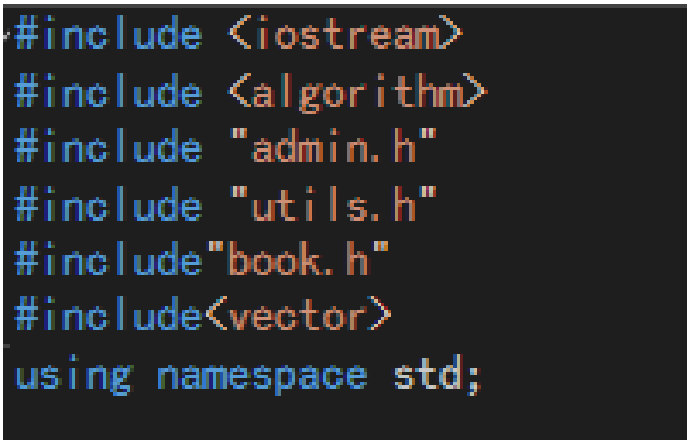
8. Contribution Table



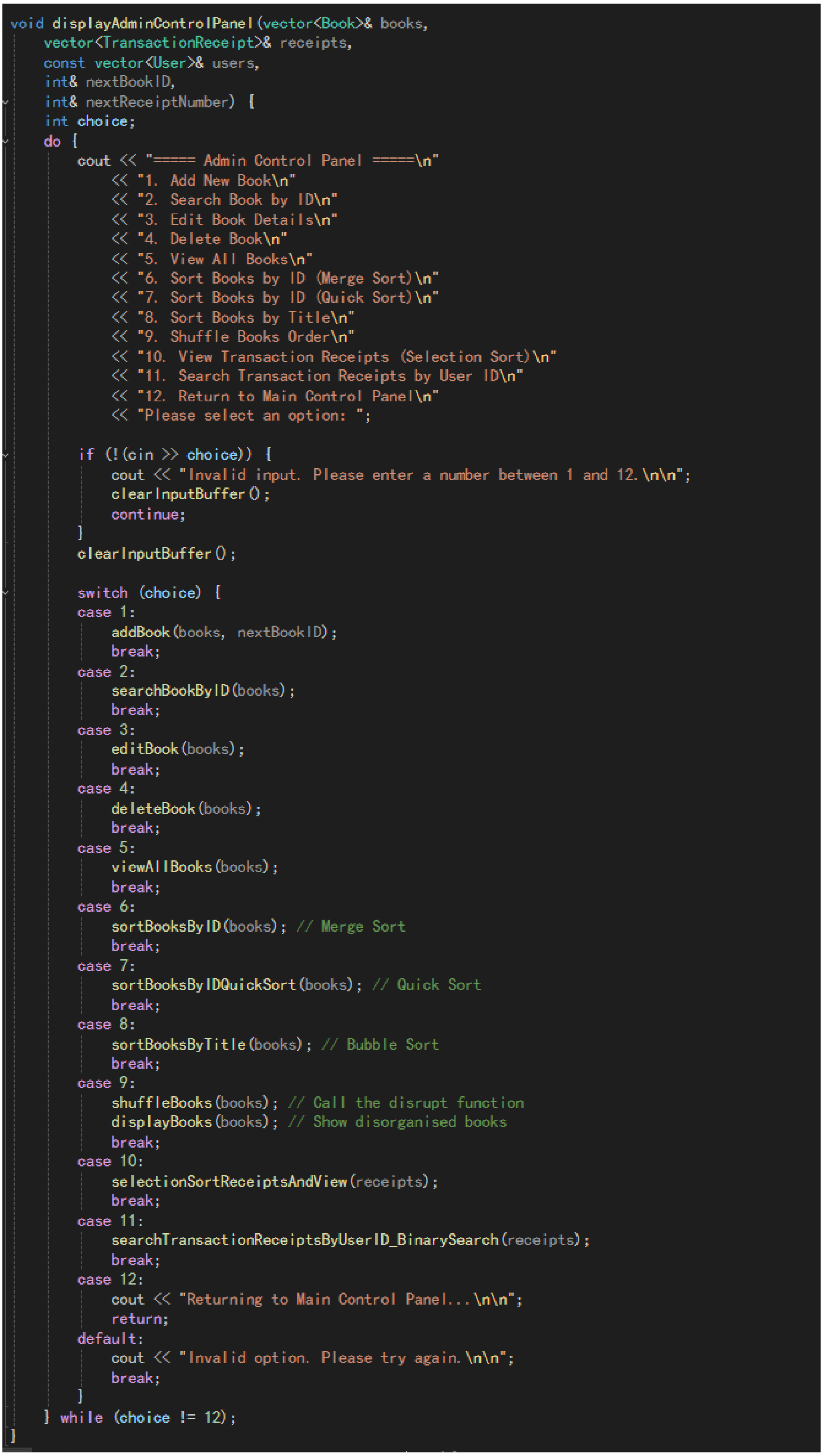
Member of the group	**Contribution to the project**
Sun zhengwu (leader)	**Code**: All error-handling and file processing functions, all bug fixes and splicing of various modules to fill in the functional framework. **Lab report**: Brief Explanation of Algorithms Used, Error Handling Approach and Challenges, and final integration of all parts. **Video**: Participate in the shooting.
Mosaddiqur Rahman	**Code**: Basic framework of system panel of Library User Panel (Including algorithm functions). **Lab report**: The Easier Sorting Algorithm to Implement. **Video**: Participate in the shooting.
Chen junfeng	**Code**: Basic framework of functionalities of Library User Panel (Including algorithm functions). **Lab report**: Comparison and challenges of implemented search algorithm and contribution table. **Video**: Participate in the shooting.
Lim junming	**Code**: Basic framework of book borrowing process of Library User Panel (Including algorithm functions). **Lab report**: Easiest search algorithm. **Video**: Video post-production.
Zheng yucheng	**Code**: Basic framework of admin (Including algorithm functions). **Lab report**: Comparison of sorting algorithms and any difficulties encountered. **Video**: Participate in the shooting.

9. Appendix:Source Code

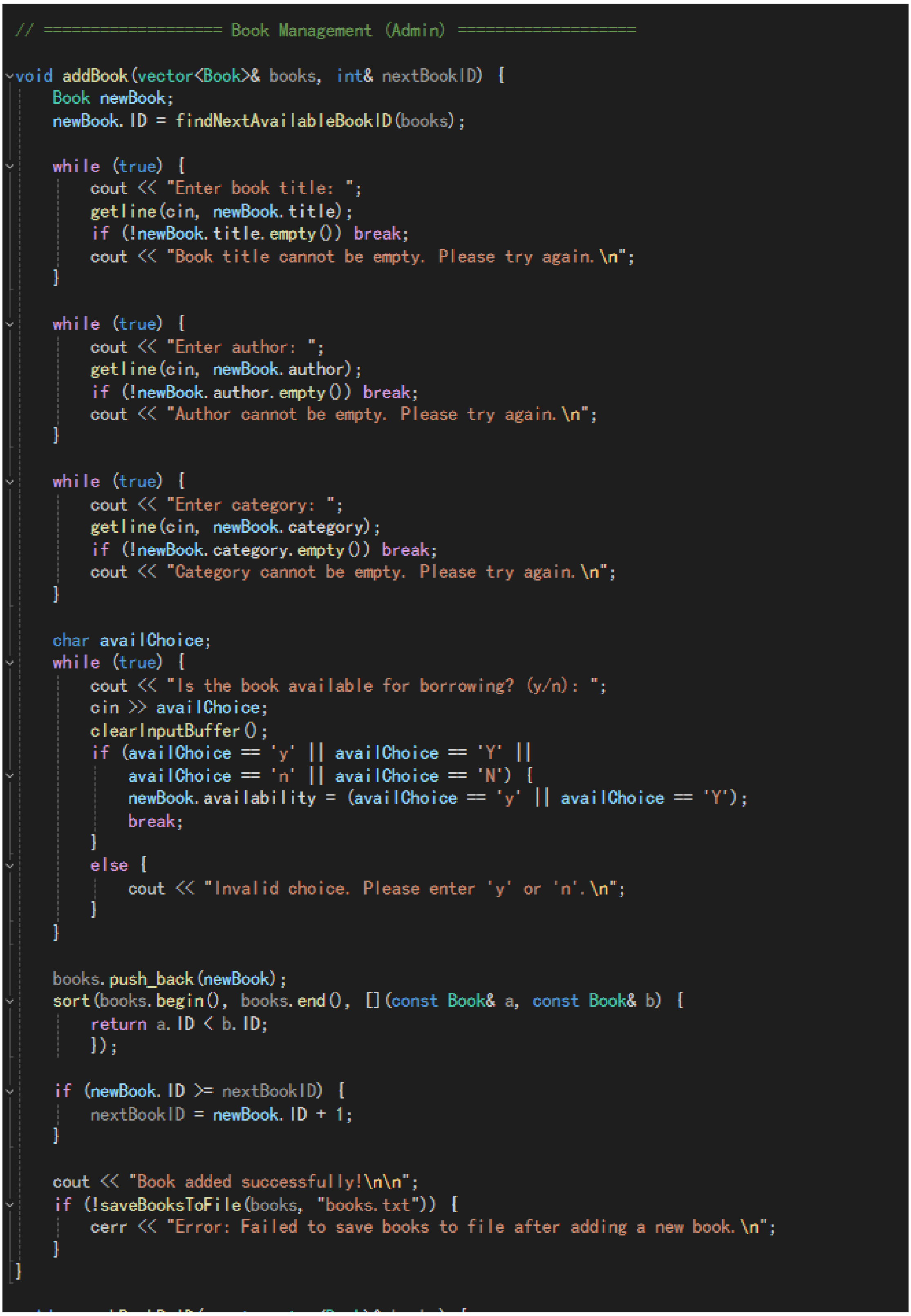
Admin.cpp



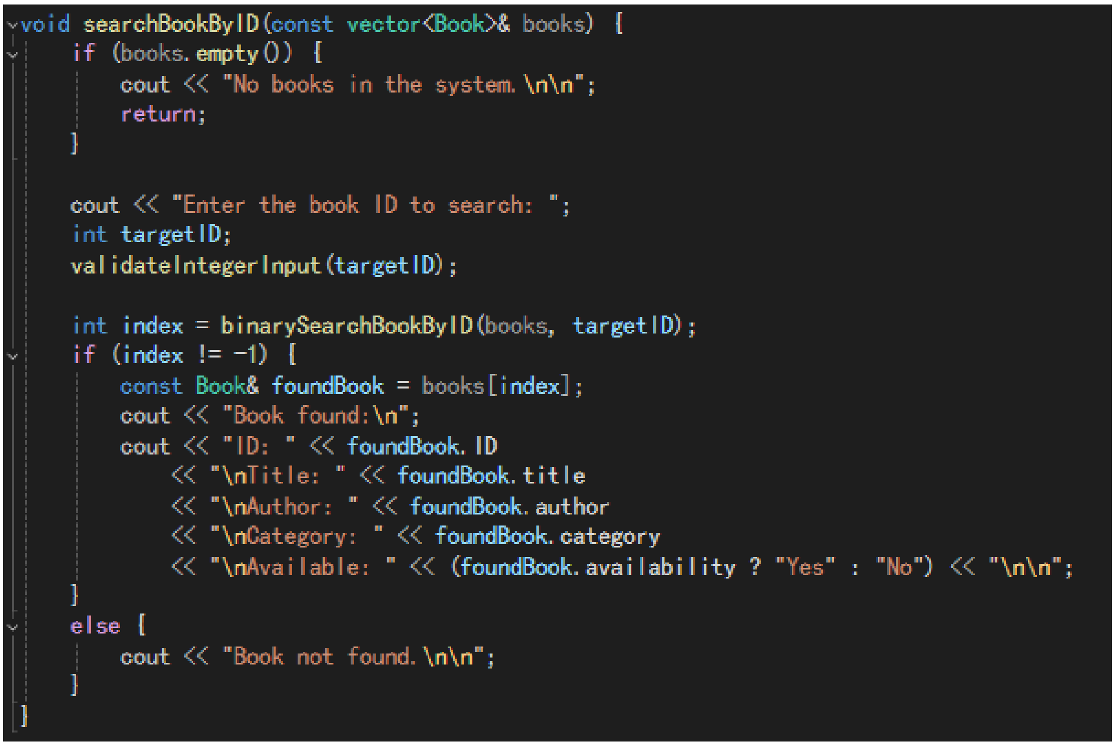
```
#include <iostream>
#include <algorithm>
#include "admin.h"
#include "utils.h"
#include"book.h"
#include<vector>
using namespace std;
```



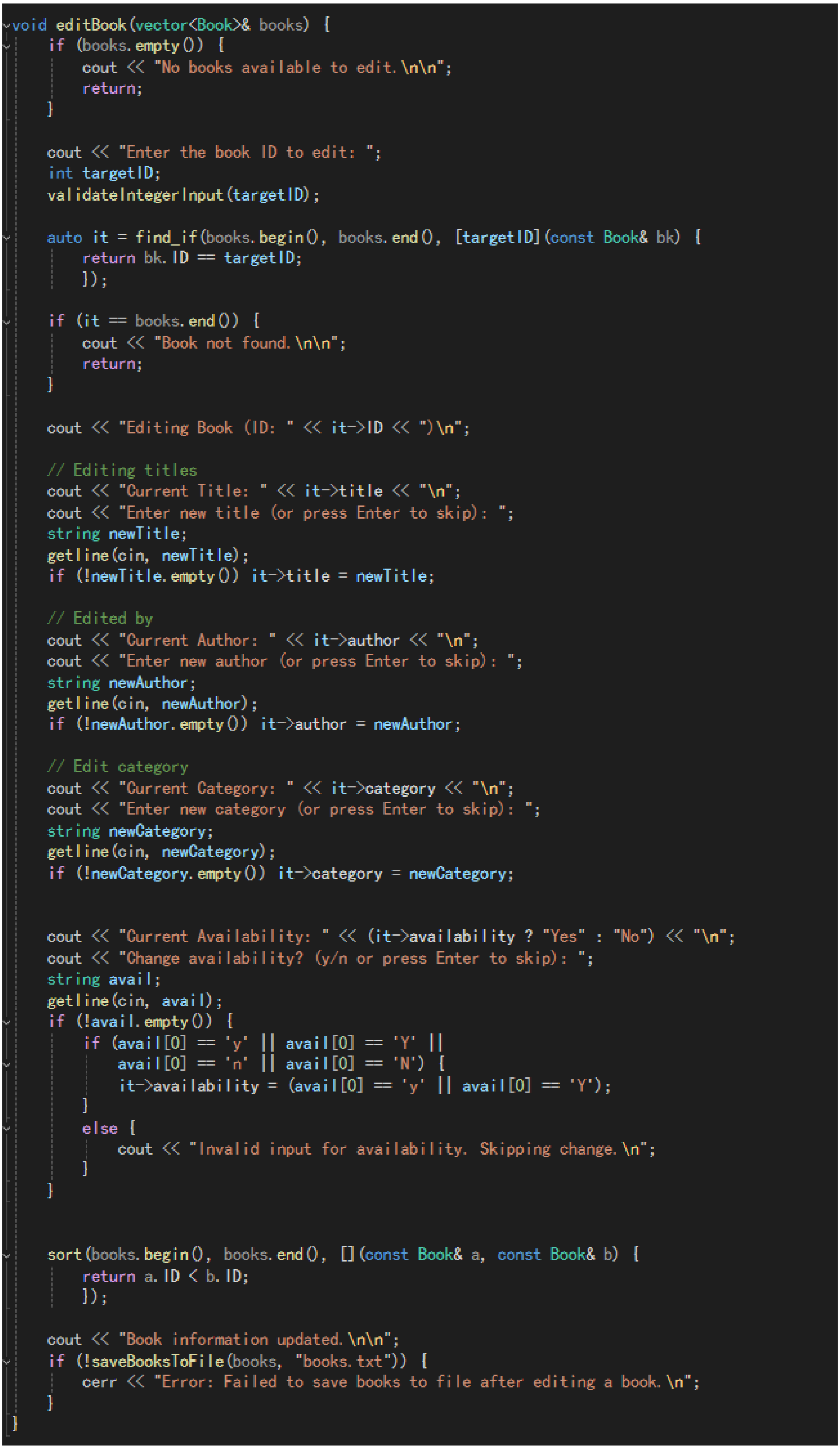
```
void displayAdminControlPanel(vector<Book>& books,
    vector<TransactionReceipt>& receipts,
    const vector<User>& users,
    int& nextBookID,
    int& nextReceiptNumber) {
    int choice;
    do {
        cout << "===== Admin Control Panel =====\n"
            << "1. Add New Book\n"
            << "2. Search Book by ID\n"
            << "3. Edit Book Details\n"
            << "4. Delete Book\n"
            << "5. View All Books\n"
            << "6. Sort Books by ID (Merge Sort)\n"
            << "7. Sort Books by ID (Quick Sort)\n"
            << "8. Sort Books by Title\n"
            << "9. Shuffle Books Order\n"
            << "10. View Transaction Receipts (Selection Sort)\n"
            << "11. Search Transaction Receipts by User ID\n"
            << "12. Return to Main Control Panel\n"
            << "Please select an option: ";

        if (!(cin >> choice)) {
            cout << "Invalid input. Please enter a number between 1 and 12.\n\n";
            clearInputBuffer();
            continue;
        }
        clearInputBuffer();

        switch (choice) {
        case 1:
            addBook(books, nextBookID);
            break;
        case 2:
            searchBookByID(books);
            break;
        case 3:
            editBook(books);
            break;
        case 4:
            deleteBook(books);
            break;
        case 5:
            viewAllBooks(books);
            break;
        case 6:
            sortBooksByID(books); // Merge Sort
            break;
        case 7:
            sortBooksByIDQuickSort(books); // Quick Sort
            break;
        case 8:
            sortBooksByTitle(books); // Bubble Sort
            break;
        case 9:
            shuffleBooks(books); // Call the disrupt function
            displayBooks(books); // Show disorganised books
            break;
        case 10:
            selectionSortReceiptsAndView(receipts);
            break;
        case 11:
            searchTransactionReceiptsByUserID_BinarySearch(receipts);
            break;
        case 12:
            cout << "Returning to Main Control Panel...\n\n";
            return;
        default:
            cout << "Invalid option. Please try again.\n\n";
            break;
        }
    } while (choice != 12);
}
```



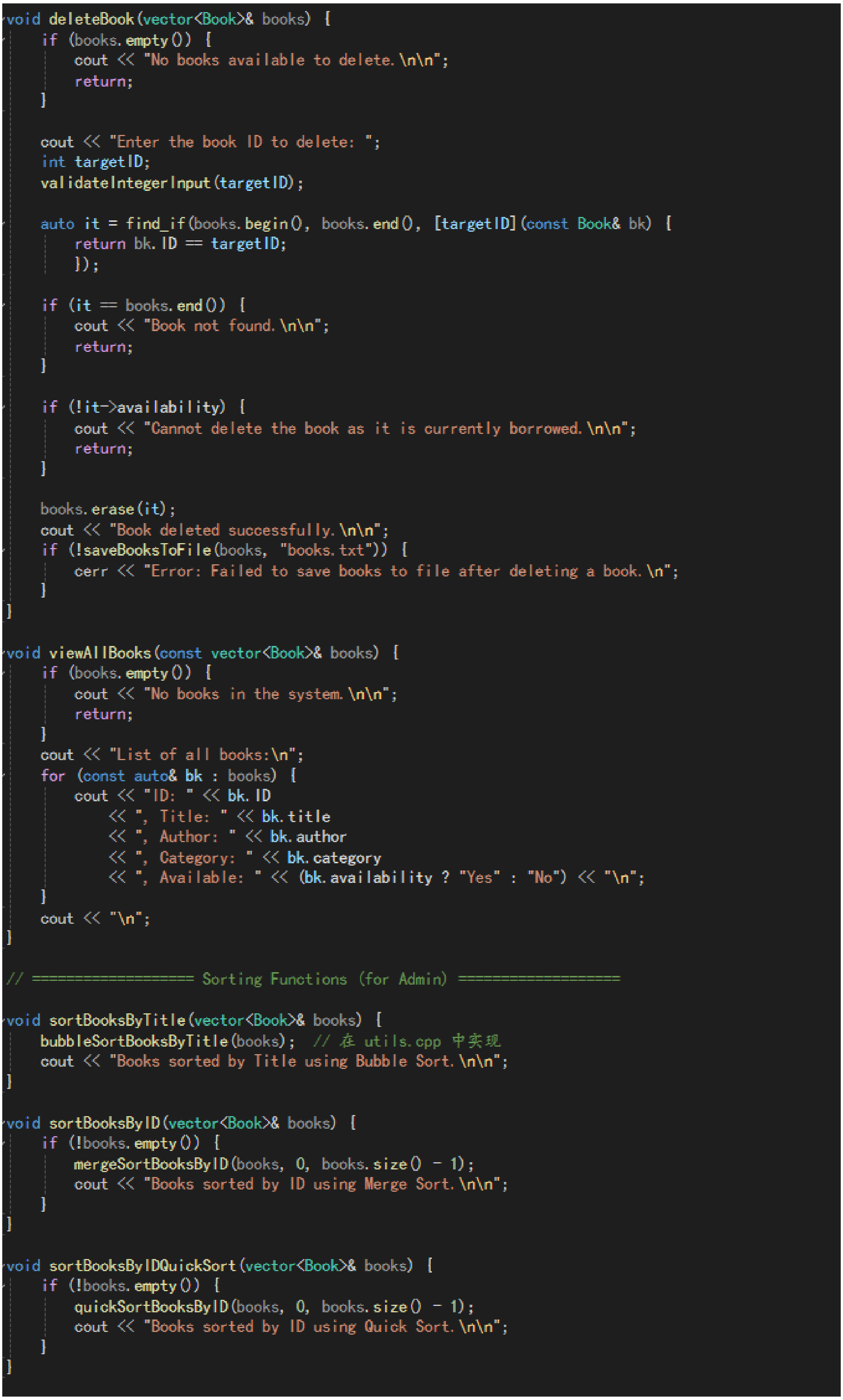
```
// ================== Book Management (Admin) ==================

void addBook(vector<Book>& books, int& nextBookID) {
    Book newBook;
    newBook.ID = findNextAvailableBookID(books);

    while (true) {
        cout << "Enter book title: ";
        getline(cin, newBook.title);
        if (!newBook.title.empty()) break;
        cout << "Book title cannot be empty. Please try again.\n";
    }

    while (true) {
        cout << "Enter author: ";
        getline(cin, newBook.author);
        if (!newBook.author.empty()) break;
        cout << "Author cannot be empty. Please try again.\n";
    }

    while (true) {
        cout << "Enter category: ";
        getline(cin, newBook.category);
        if (!newBook.category.empty()) break;
        cout << "Category cannot be empty. Please try again.\n";
    }

    char availChoice;
    while (true) {
        cout << "Is the book available for borrowing? (y/n): ";
        cin >> availChoice;
        clearInputBuffer();
        if (availChoice == 'y' || availChoice == 'Y' ||
            availChoice == 'n' || availChoice == 'N') {
            newBook.availability = (availChoice == 'y' || availChoice == 'Y');
            break;
        }
        else {
            cout << "Invalid choice. Please enter 'y' or 'n'.\n";
        }
    }

    books.push_back(newBook);
    sort(books.begin(), books.end(), [](const Book& a, const Book& b) {
        return a.ID < b.ID;
        });

    if (newBook.ID >= nextBookID) {
        nextBookID = newBook.ID + 1;
    }

    cout << "Book added successfully!\n\n";
    if (!saveBooksToFile(books, "books.txt")) {
        cerr << "Error: Failed to save books to file after adding a new book.\n";
    }
}
```



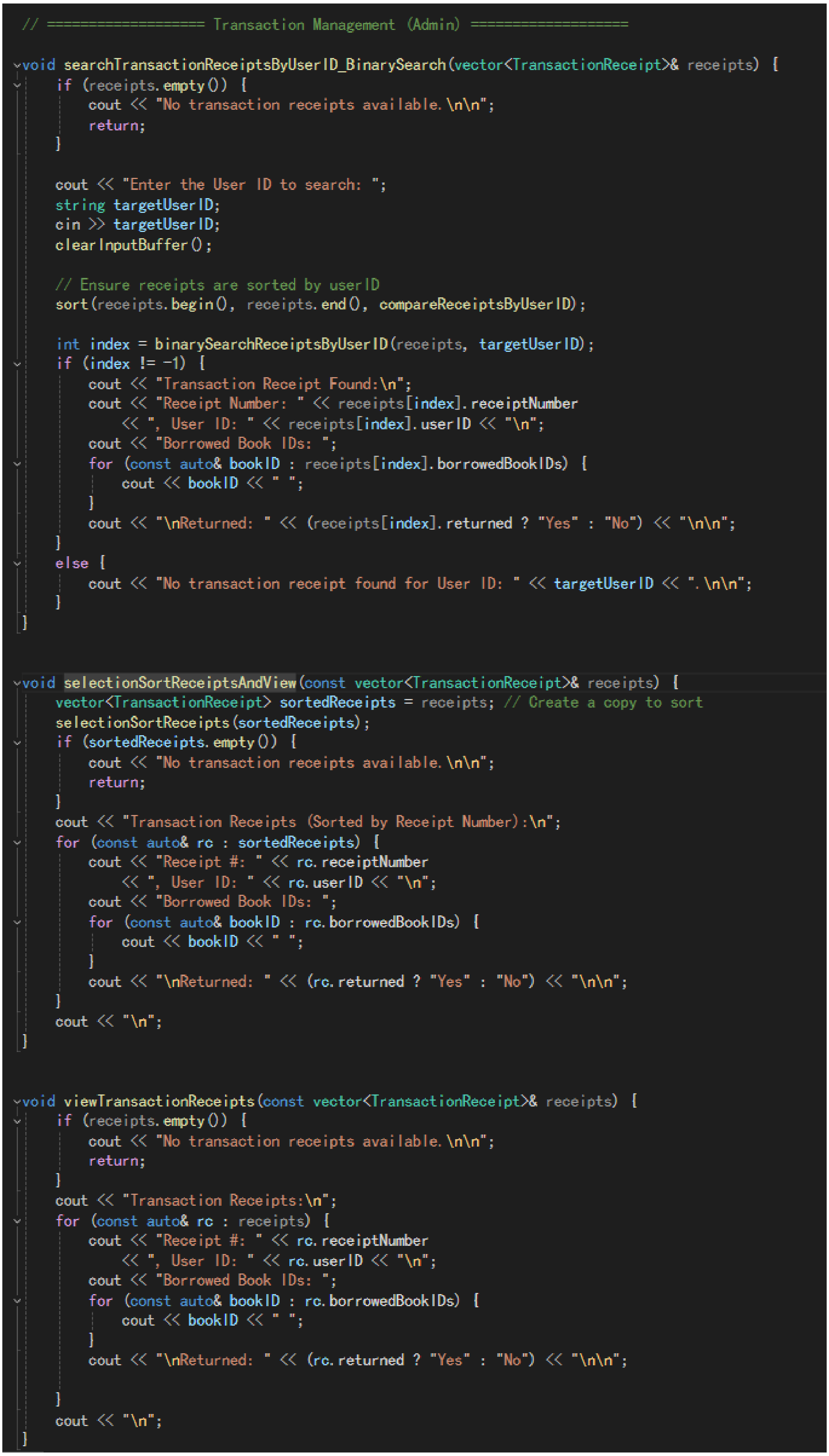
```
void searchBookByID(const vector<Book>& books) {
    if (books.empty()) {
        cout << "No books in the system.\n\n";
        return;
    }

    cout << "Enter the book ID to search: ";
    int targetID;
    validateIntegerInput(targetID);

    int index = binarySearchBookByID(books, targetID);
    if (index != -1) {
        const Book& foundBook = books[index];
        cout << "Book found:\n";
        cout << "ID: " << foundBook.ID
            << "\nTitle: " << foundBook.title
            << "\nAuthor: " << foundBook.author
            << "\nCategory: " << foundBook.category
            << "\nAvailable: " << (foundBook.availability ? "Yes" : "No") << "\n\n";
    }
    else {
        cout << "Book not found.\n\n";
    }
}
```



```
void editBook(vector<Book>& books) {
    if (books.empty()) {
        cout << "No books available to edit.\n\n";
        return;
    }

    cout << "Enter the book ID to edit: ";
    int targetID;
    validateIntegerInput(targetID);

    auto it = find_if(books.begin(), books.end(), [targetID](const Book& bk) {
        return bk.ID == targetID;
        });

    if (it == books.end()) {
        cout << "Book not found.\n\n";
        return;
    }

    cout << "Editing Book (ID: " << it->ID << ")\n";

    // Editing titles
    cout << "Current Title: " << it->title << "\n";
    cout << "Enter new title (or press Enter to skip): ";
    string newTitle;
    getline(cin, newTitle);
    if (!newTitle.empty()) it->title = newTitle;

    // Edited by
    cout << "Current Author: " << it->author << "\n";
    cout << "Enter new author (or press Enter to skip): ";
    string newAuthor;
    getline(cin, newAuthor);
    if (!newAuthor.empty()) it->author = newAuthor;

    // Edit category
    cout << "Current Category: " << it->category << "\n";
    cout << "Enter new category (or press Enter to skip): ";
    string newCategory;
    getline(cin, newCategory);
    if (!newCategory.empty()) it->category = newCategory;


    cout << "Current Availability: " << (it->availability ? "Yes" : "No") << "\n";
    cout << "Change availability? (y/n or press Enter to skip): ";
    string avail;
    getline(cin, avail);
    if (!avail.empty()) {
        if (avail[0] == 'y' || avail[0] == 'Y' ||
            avail[0] == 'n' || avail[0] == 'N') {
            it->availability = (avail[0] == 'y' || avail[0] == 'Y');
        }
        else {
            cout << "Invalid input for availability. Skipping change.\n";
        }
    }


    sort(books.begin(), books.end(), [](const Book& a, const Book& b) {
        return a.ID < b.ID;
        });

    cout << "Book information updated.\n\n";
    if (!saveBooksToFile(books, "books.txt")) {
        cerr << "Error: Failed to save books to file after editing a book.\n";
    }
}
```

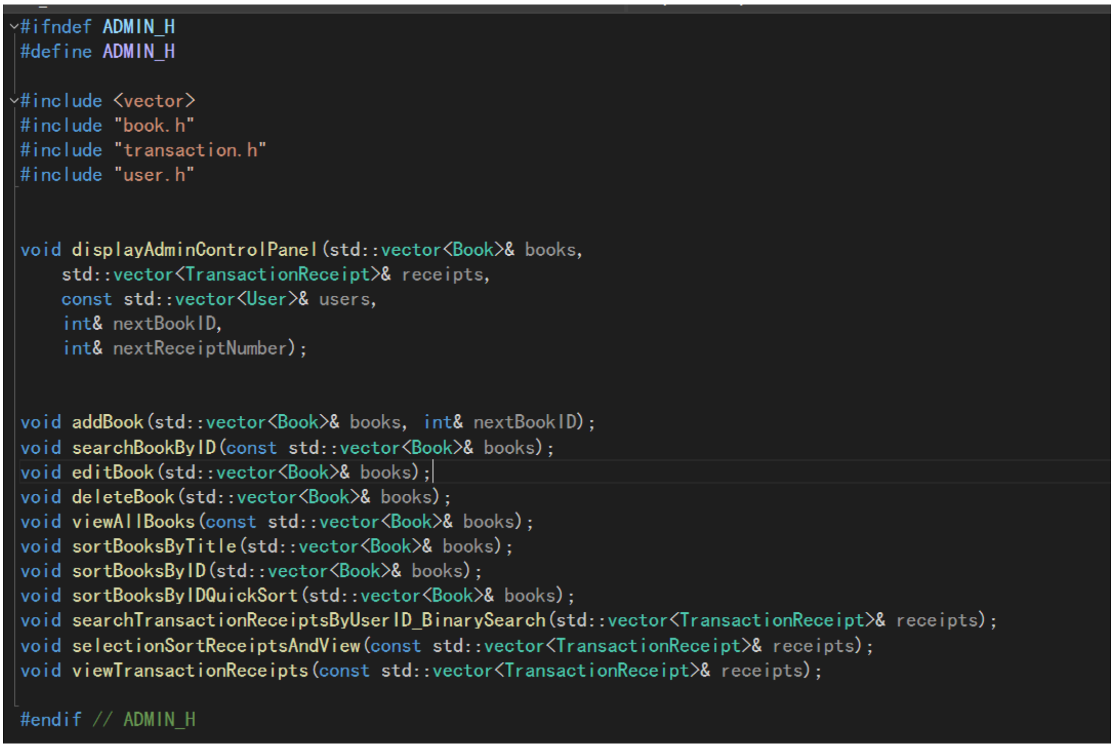


```cpp
void deleteBook(vector<Book>& books) {
    if (books.empty()) {
        cout << "No books available to delete.\n\n";
        return;
    }

    cout << "Enter the book ID to delete: ";
    int targetID;
    validateIntegerInput(targetID);

    auto it = find_if(books.begin(), books.end(), [targetID](const Book& bk) {
        return bk.ID == targetID;
        });

    if (it == books.end()) {
        cout << "Book not found.\n\n";
        return;
    }

    if (!it->availability) {
        cout << "Cannot delete the book as it is currently borrowed.\n\n";
        return;
    }

    books.erase(it);
    cout << "Book deleted successfully.\n\n";
    if (!saveBooksToFile(books, "books.txt")) {
        cerr << "Error: Failed to save books to file after deleting a book.\n";
    }
}

void viewAllBooks(const vector<Book>& books) {
    if (books.empty()) {
        cout << "No books in the system.\n\n";
        return;
    }
    cout << "List of all books:\n";
    for (const auto& bk : books) {
        cout << "ID: " << bk.ID
            << ", Title: " << bk.title
            << ", Author: " << bk.author
            << ", Category: " << bk.category
            << ", Available: " << (bk.availability ? "Yes" : "No") << "\n";
    }
    cout << "\n";
}

// ================== Sorting Functions (for Admin) ==================

void sortBooksByTitle(vector<Book>& books) {
    bubbleSortBooksByTitle(books);  // 在 utils.cpp 中实现
    cout << "Books sorted by Title using Bubble Sort.\n\n";
}

void sortBooksByID(vector<Book>& books) {
    if (!books.empty()) {
        mergeSortBooksByID(books, 0, books.size() - 1);
        cout << "Books sorted by ID using Merge Sort.\n\n";
    }
}

void sortBooksByIDQuickSort(vector<Book>& books) {
    if (!books.empty()) {
        quickSortBooksByID(books, 0, books.size() - 1);
        cout << "Books sorted by ID using Quick Sort.\n\n";
    }
}
```



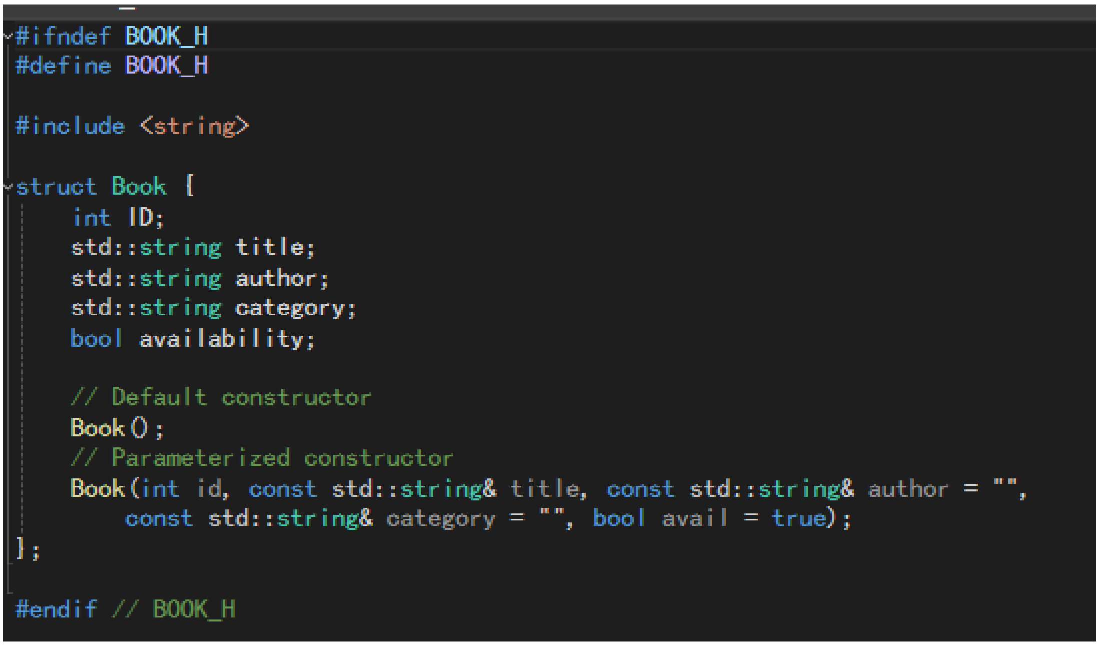
```
// ================== Transaction Management (Admin) ==================

void searchTransactionReceiptsByUserID_BinarySearch(vector<TransactionReceipt>& receipts) {
    if (receipts.empty()) {
        cout << "No transaction receipts available.\n\n";
        return;
    }

    cout << "Enter the User ID to search: ";
    string targetUserID;
    cin >> targetUserID;
    clearInputBuffer();

    // Ensure receipts are sorted by userID
    sort(receipts.begin(), receipts.end(), compareReceiptsByUserID);

    int index = binarySearchReceiptsByUserID(receipts, targetUserID);
    if (index != -1) {
        cout << "Transaction Receipt Found:\n";
        cout << "Receipt Number: " << receipts[index].receiptNumber
            << ", User ID: " << receipts[index].userID << "\n";
        cout << "Borrowed Book IDs: ";
        for (const auto& bookID : receipts[index].borrowedBookIDs) {
            cout << bookID << " ";
        }
        cout << "\nReturned: " << (receipts[index].returned ? "Yes" : "No") << "\n\n";
    }
    else {
        cout << "No transaction receipt found for User ID: " << targetUserID << ".\n\n";
    }
}


void selectionSortReceiptsAndView(const vector<TransactionReceipt>& receipts) {
    vector<TransactionReceipt> sortedReceipts = receipts; // Create a copy to sort
    selectionSortReceipts(sortedReceipts);
    if (sortedReceipts.empty()) {
        cout << "No transaction receipts available.\n\n";
        return;
    }
    cout << "Transaction Receipts (Sorted by Receipt Number):\n";
    for (const auto& rc : sortedReceipts) {
        cout << "Receipt #: " << rc.receiptNumber
            << ", User ID: " << rc.userID << "\n";
        cout << "Borrowed Book IDs: ";
        for (const auto& bookID : rc.borrowedBookIDs) {
            cout << bookID << " ";
        }
        cout << "\nReturned: " << (rc.returned ? "Yes" : "No") << "\n\n";
    }
    cout << "\n";
}


void viewTransactionReceipts(const vector<TransactionReceipt>& receipts) {
    if (receipts.empty()) {
        cout << "No transaction receipts available.\n\n";
        return;
    }
    cout << "Transaction Receipts:\n";
    for (const auto& rc : receipts) {
        cout << "Receipt #: " << rc.receiptNumber
            << ", User ID: " << rc.userID << "\n";
        cout << "Borrowed Book IDs: ";
        for (const auto& bookID : rc.borrowedBookIDs) {
            cout << bookID << " ";
        }
        cout << "\nReturned: " << (rc.returned ? "Yes" : "No") << "\n\n";

    }
    cout << "\n";
}
```

Admin.h



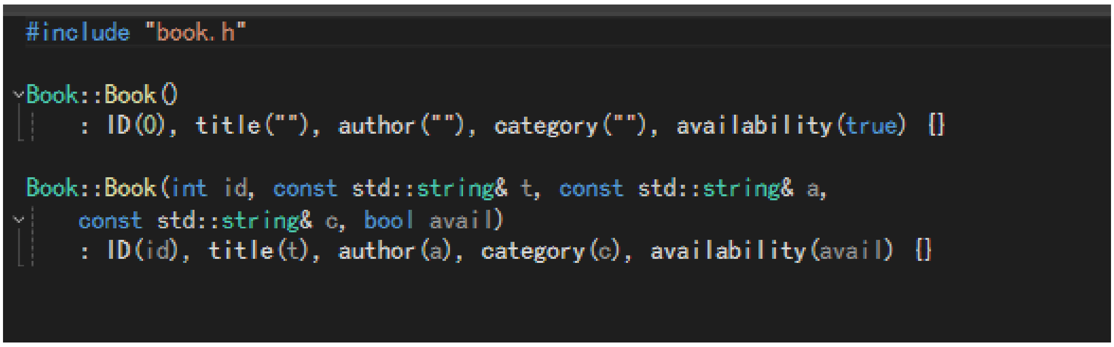
```
#ifndef ADMIN_H
#define ADMIN_H

#include <vector>
#include "book.h"
#include "transaction.h"
#include "user.h"


void displayAdminControlPanel(std::vector<Book>& books,
    std::vector<TransactionReceipt>& receipts,
    const std::vector<User>& users,
    int& nextBookID,
    int& nextReceiptNumber);


void addBook(std::vector<Book>& books, int& nextBookID);
void searchBookByID(const std::vector<Book>& books);
void editBook(std::vector<Book>& books);
void deleteBook(std::vector<Book>& books);
void viewAllBooks(const std::vector<Book>& books);
void sortBooksByTitle(std::vector<Book>& books);
void sortBooksByID(std::vector<Book>& books);
void sortBooksByIDQuickSort(std::vector<Book>& books);
void searchTransactionReceiptsByUserID_BinarySearch(std::vector<TransactionReceipt>& receipts);
void selectionSortReceiptsAndView(const std::vector<TransactionReceipt>& receipts);
void viewTransactionReceipts(const std::vector<TransactionReceipt>& receipts);

#endif // ADMIN_H
```

Book.h



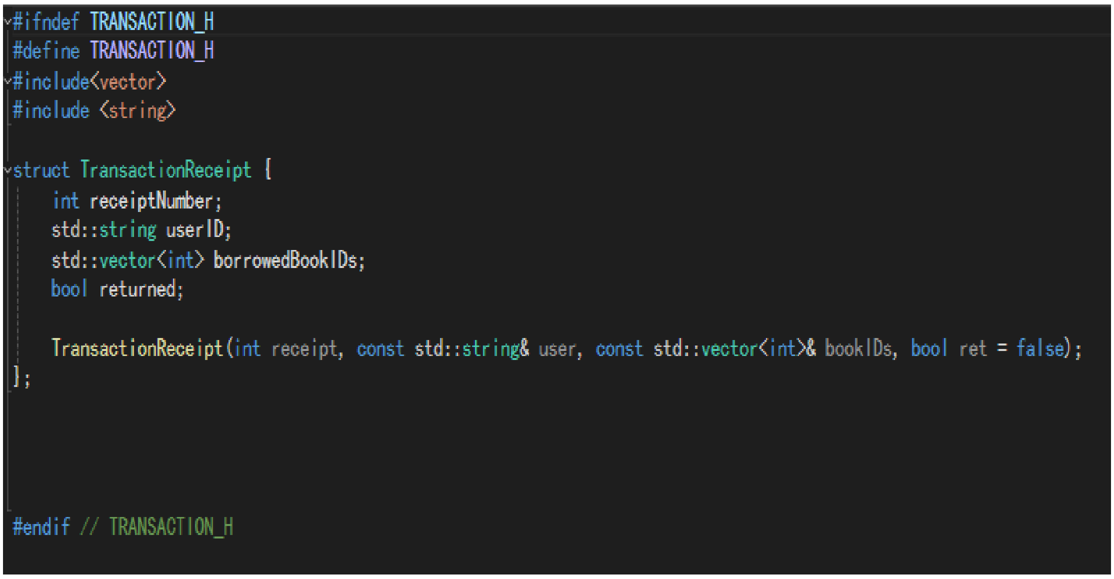
```
#ifndef BOOK_H
#define BOOK_H

#include <string>

struct Book {
    int ID;
    std::string title;
    std::string author;
    std::string category;
    bool availability;

    // Default constructor
    Book();
    // Parameterized constructor
    Book(int id, const std::string& title, const std::string& author = "",
        const std::string& category = "", bool avail = true);
};

#endif // BOOK_H
```

Book.cpp



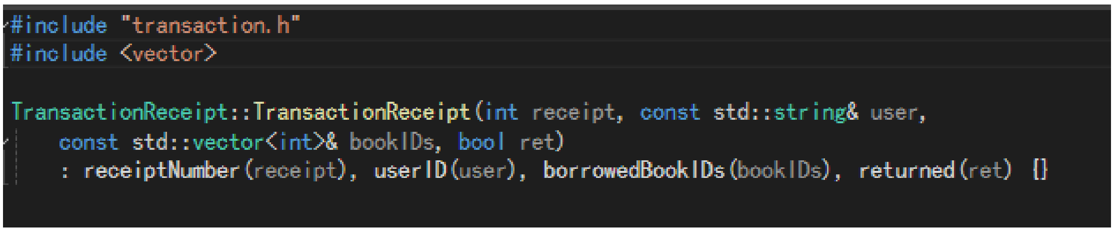
```cpp
#include "book.h"

Book::Book()
    : ID(0), title(""), author(""), category(""), availability(true) {}

Book::Book(int id, const std::string& t, const std::string& a,
    const std::string& c, bool avail)
    : ID(id), title(t), author(a), category(c), availability(avail) {}
```

Transaction.h



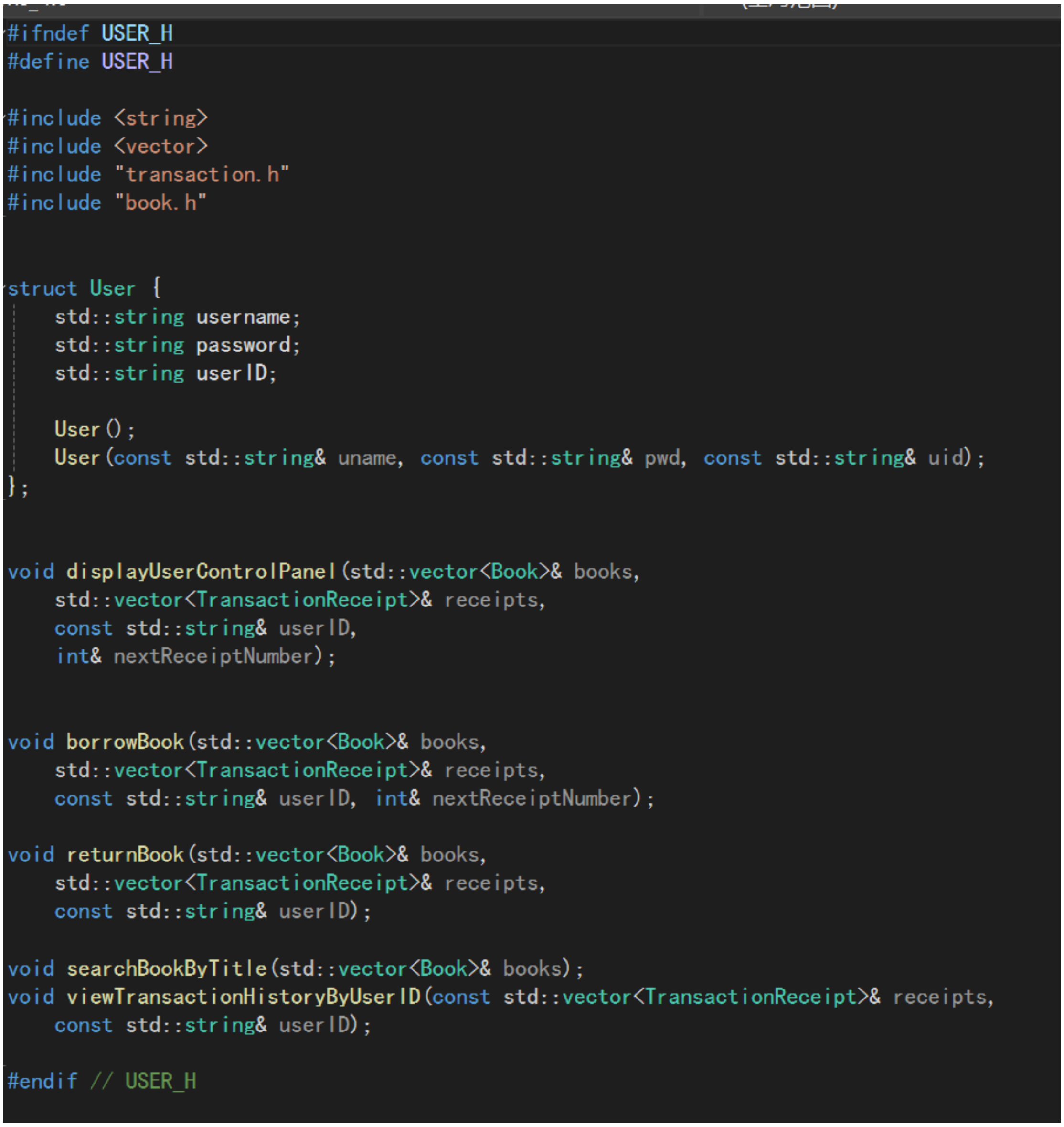
```cpp
#ifndef TRANSACTION_H
#define TRANSACTION_H
#include<vector>
#include <string>

struct TransactionReceipt {
    int receiptNumber;
    std::string userID;
    std::vector<int> borrowedBookIDs;
    bool returned;

    TransactionReceipt(int receipt, const std::string& user, const std::vector<int>& bookIDs, bool ret = false);
};




#endif // TRANSACTION_H
```

Transaction.cpp



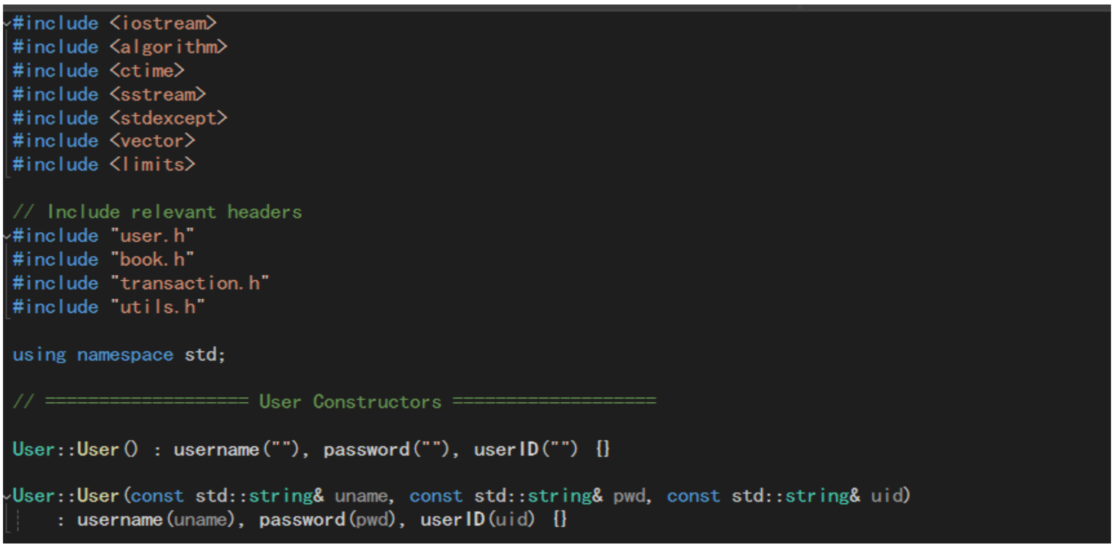
```cpp
#include "transaction.h"
#include <vector>

TransactionReceipt::TransactionReceipt(int receipt, const std::string& user,
    const std::vector<int>& bookIDs, bool ret)
    : receiptNumber(receipt), userID(user), borrowedBookIDs(bookIDs), returned(ret) {}
```

User.h

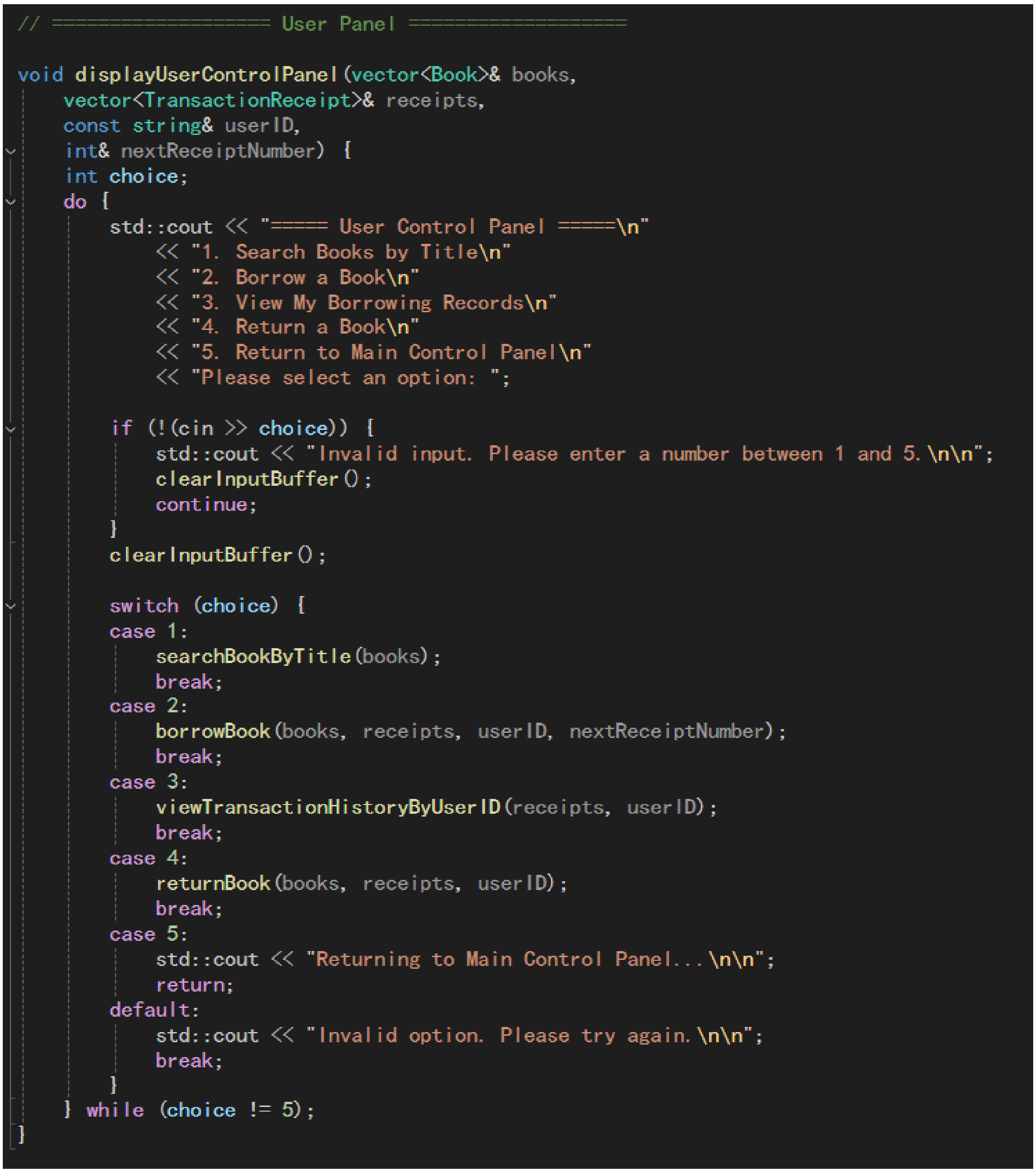


```
#ifndef USER_H
#define USER_H

#include <string>
#include <vector>
#include "transaction.h"
#include "book.h"


struct User {
    std::string username;
    std::string password;
    std::string userID;

    User();
    User(const std::string& uname, const std::string& pwd, const std::string& uid);
};


void displayUserControlPanel(std::vector<Book>& books,
    std::vector<TransactionReceipt>& receipts,
    const std::string& userID,
    int& nextReceiptNumber);


void borrowBook(std::vector<Book>& books,
    std::vector<TransactionReceipt>& receipts,
    const std::string& userID, int& nextReceiptNumber);

void returnBook(std::vector<Book>& books,
    std::vector<TransactionReceipt>& receipts,
    const std::string& userID);

void searchBookByTitle(std::vector<Book>& books);
void viewTransactionHistoryByUserID(const std::vector<TransactionReceipt>& receipts,
    const std::string& userID);

#endif // USER_H
```

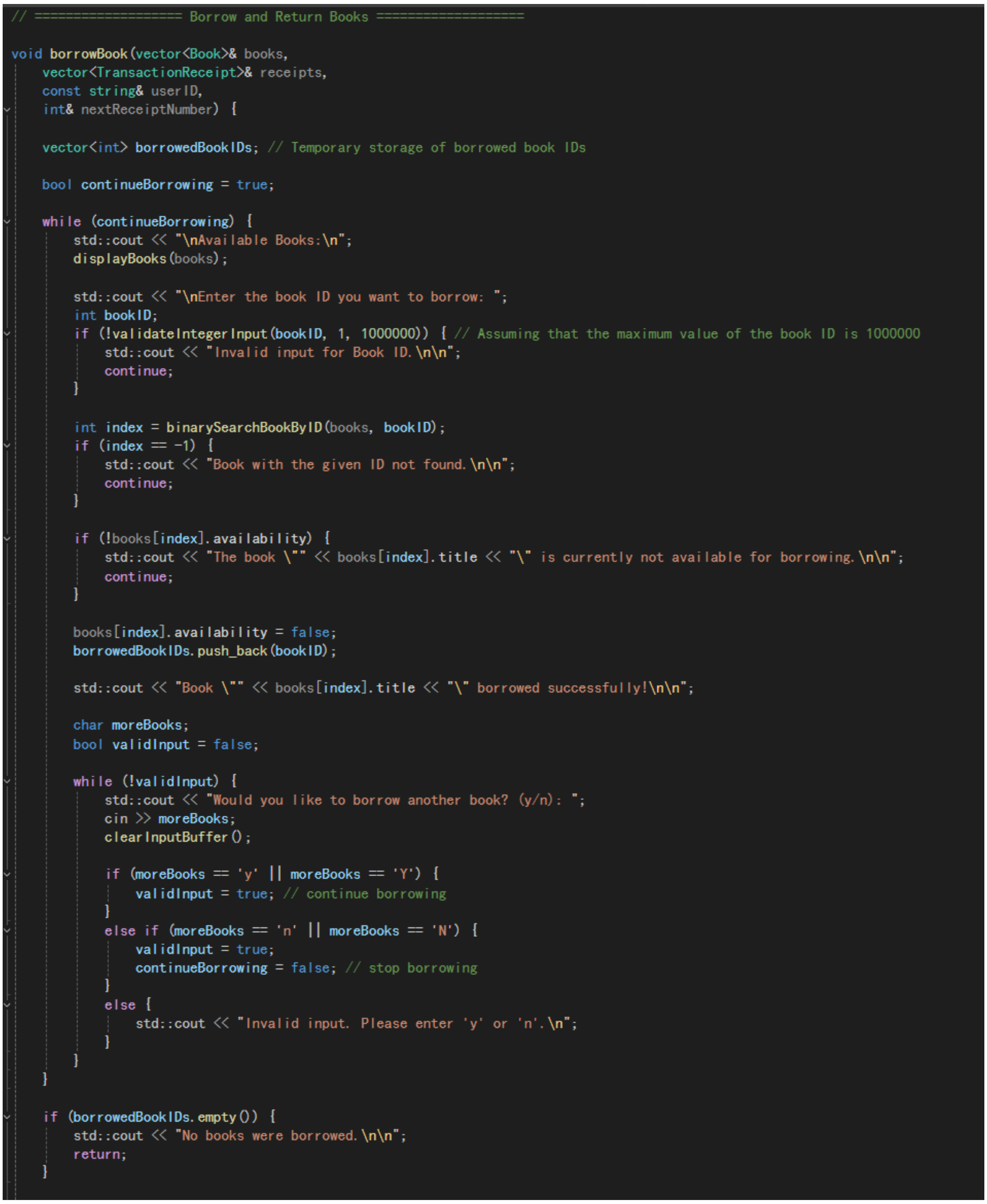
User.cpp



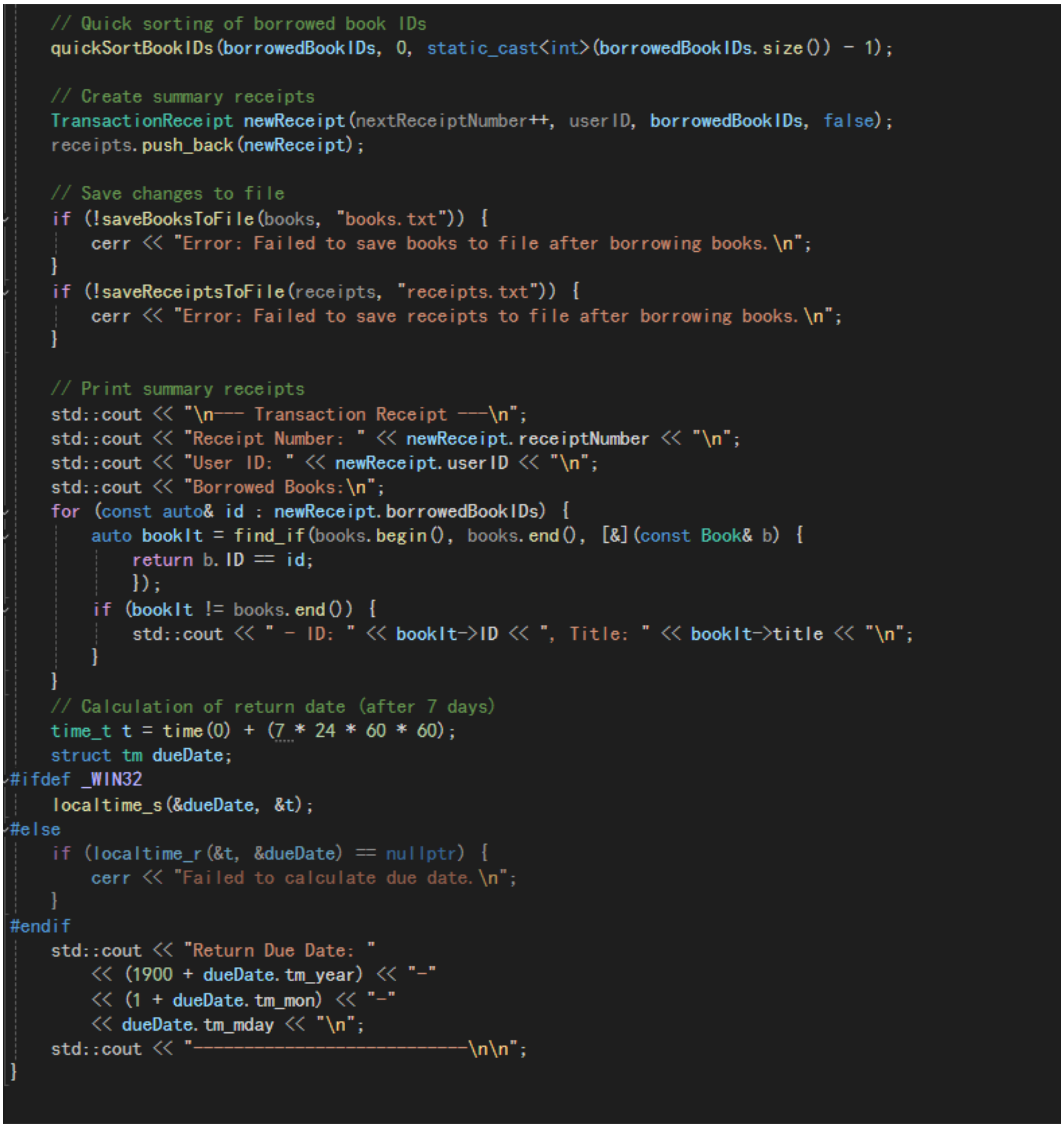
```cpp
#include <iostream>
#include <algorithm>
#include <ctime>
#include <sstream>
#include <stdexcept>
#include <vector>
#include <limits>

// Include relevant headers
#include "user.h"
#include "book.h"
#include "transaction.h"
#include "utils.h"

using namespace std;

// ================== User Constructors ==================

User::User() : username(""), password(""), userID("") {}

User::User(const std::string& uname, const std::string& pwd, const std::string& uid)
    : username(uname), password(pwd), userID(uid) {}
```



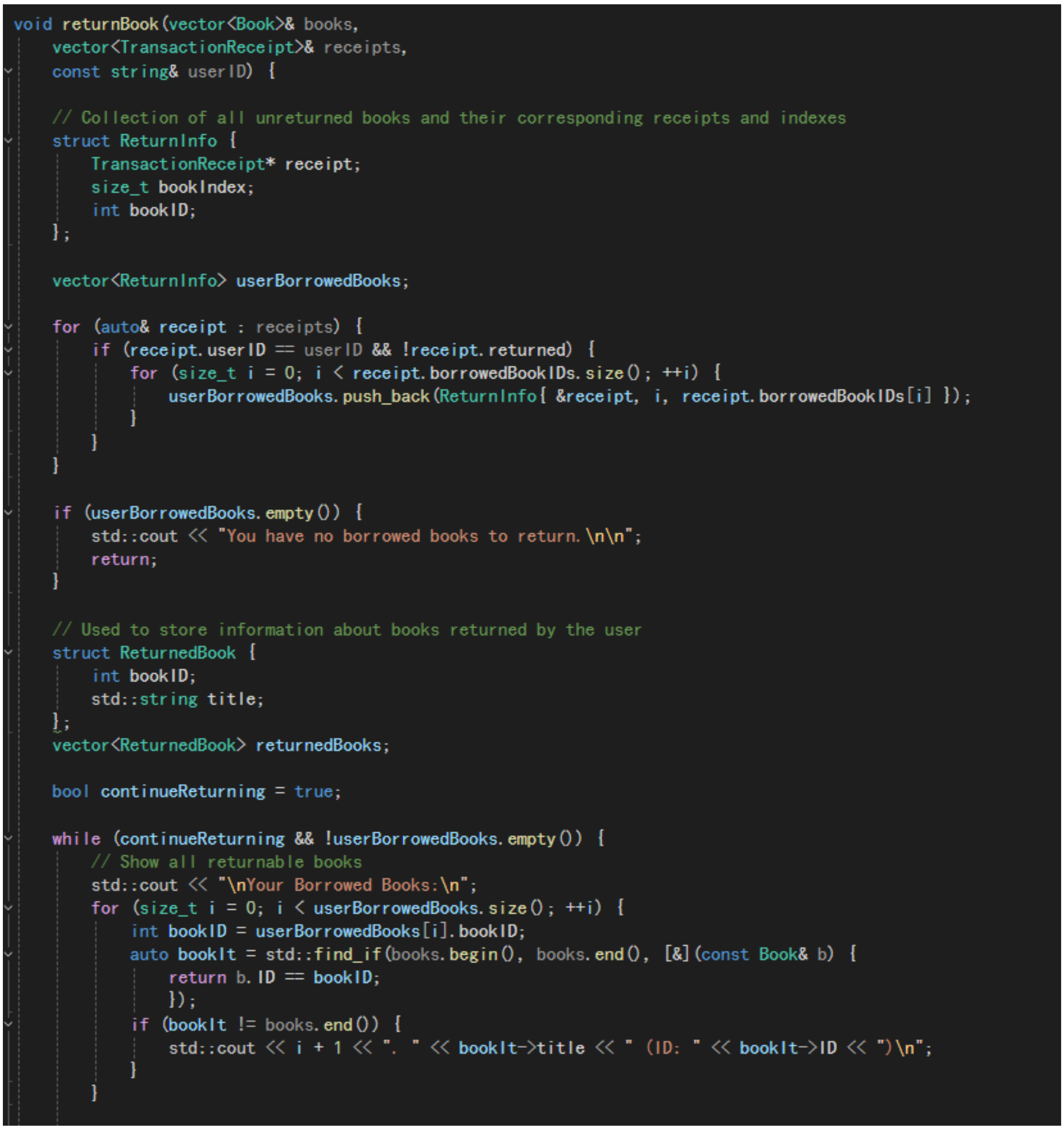
```cpp
// =================== User Panel ===================

void displayUserControlPanel(vector<Book>& books,
    vector<TransactionReceipt>& receipts,
    const string& userID,
    int& nextReceiptNumber) {
    int choice;
    do {
        std::cout << "===== User Control Panel =====\n"
            << "1. Search Books by Title\n"
            << "2. Borrow a Book\n"
            << "3. View My Borrowing Records\n"
            << "4. Return a Book\n"
            << "5. Return to Main Control Panel\n"
            << "Please select an option: ";

        if (!(cin >> choice)) {
            std::cout << "Invalid input. Please enter a number between 1 and 5.\n\n";
            clearInputBuffer();
            continue;
        }
        clearInputBuffer();

        switch (choice) {
        case 1:
            searchBookByTitle(books);
            break;
        case 2:
            borrowBook(books, receipts, userID, nextReceiptNumber);
            break;
        case 3:
            viewTransactionHistoryByUserID(receipts, userID);
            break;
        case 4:
            returnBook(books, receipts, userID);
            break;
        case 5:
            std::cout << "Returning to Main Control Panel...\n\n";
            return;
        default:
            std::cout << "Invalid option. Please try again.\n\n";
            break;
        }
    } while (choice != 5);
}
```



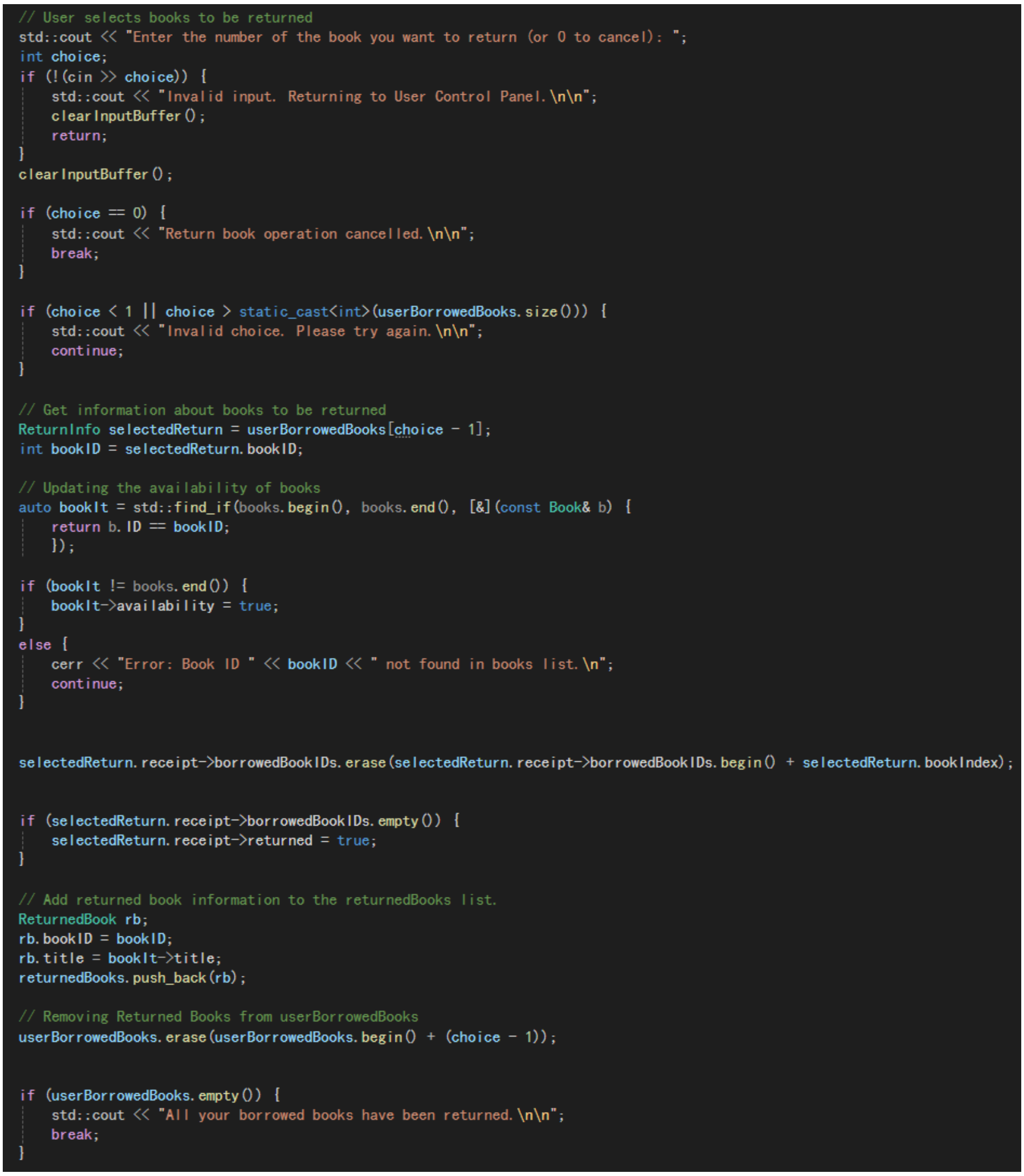
```cpp
// =================== Borrow and Return Books ===================

void borrowBook(vector<Book>& books,
    vector<TransactionReceipt>& receipts,
    const string& userID,
    int& nextReceiptNumber) {

    vector<int> borrowedBookIDs; // Temporary storage of borrowed book IDs

    bool continueBorrowing = true;

    while (continueBorrowing) {
        std::cout << "\nAvailable Books:\n";
        displayBooks(books);

        std::cout << "\nEnter the book ID you want to borrow: ";
        int bookID;
        if (!validateIntegerInput(bookID, 1, 1000000)) { // Assuming that the maximum value of the book ID is 1000000
            std::cout << "Invalid input for Book ID.\n\n";
            continue;
        }

        int index = binarySearchBookByID(books, bookID);
        if (index == -1) {
            std::cout << "Book with the given ID not found.\n\n";
            continue;
        }

        if (!books[index].availability) {
            std::cout << "The book \"" << books[index].title << "\" is currently not available for borrowing.\n\n";
            continue;
        }

        books[index].availability = false;
        borrowedBookIDs.push_back(bookID);

        std::cout << "Book \"" << books[index].title << "\" borrowed successfully!\n\n";

        char moreBooks;
        bool validInput = false;

        while (!validInput) {
            std::cout << "Would you like to borrow another book? (y/n): ";
            cin >> moreBooks;
            clearInputBuffer();

            if (moreBooks == 'y' || moreBooks == 'Y') {
                validInput = true; // continue borrowing
            }
            else if (moreBooks == 'n' || moreBooks == 'N') {
                validInput = true;
                continueBorrowing = false; // stop borrowing
            }
            else {
                std::cout << "Invalid input. Please enter 'y' or 'n'.\n";
            }
        }
    }

    if (borrowedBookIDs.empty()) {
        std::cout << "No books were borrowed.\n\n";
        return;
    }
```



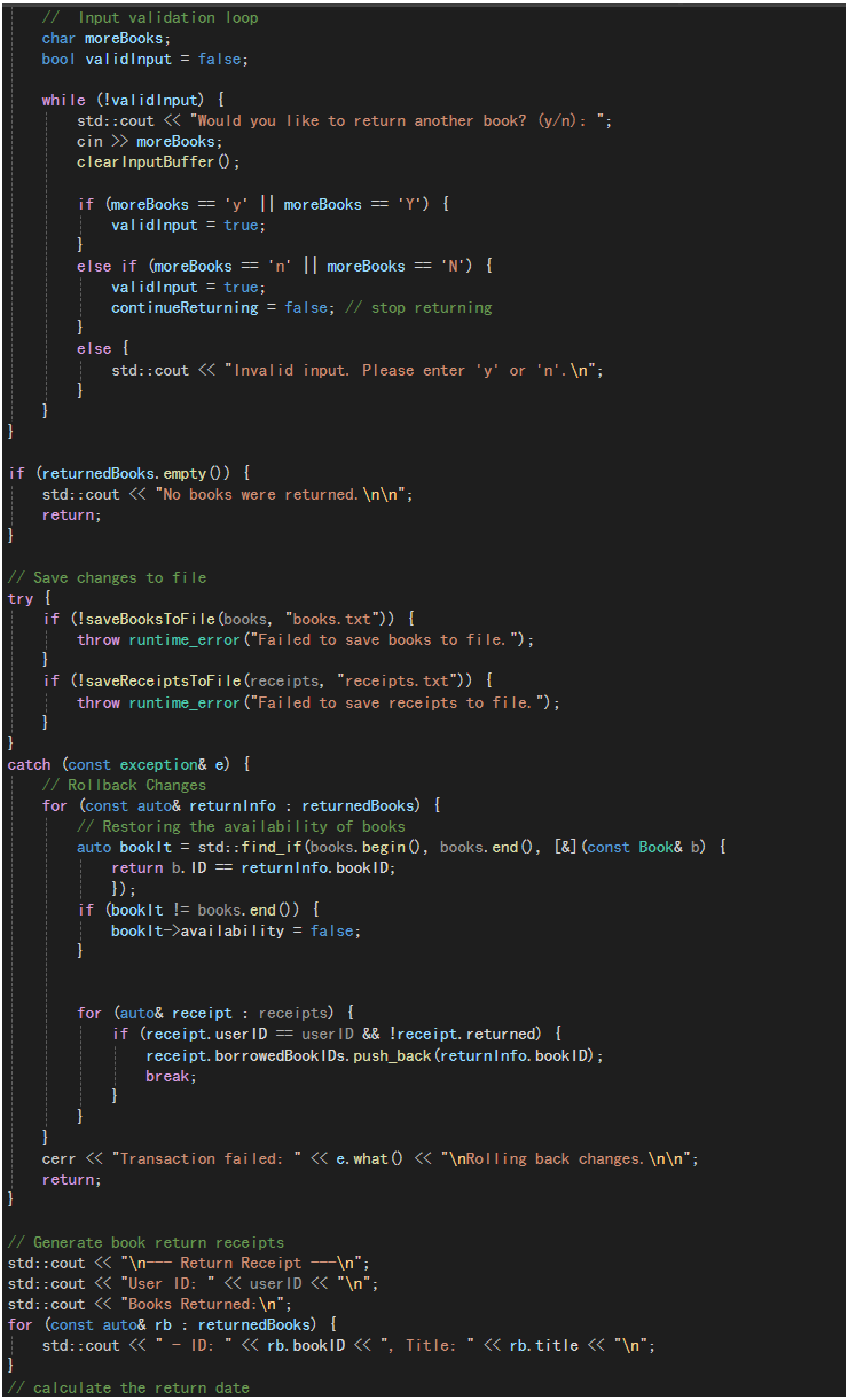
```
    // Quick sorting of borrowed book IDs
    quickSortBookIDs(borrowedBookIDs, 0, static_cast<int>(borrowedBookIDs.size()) - 1);

    // Create summary receipts
    TransactionReceipt newReceipt(nextReceiptNumber++, userID, borrowedBookIDs, false);
    receipts.push_back(newReceipt);

    // Save changes to file
    if (!saveBooksToFile(books, "books.txt")) {
        cerr << "Error: Failed to save books to file after borrowing books.\n";
    }
    if (!saveReceiptsToFile(receipts, "receipts.txt")) {
        cerr << "Error: Failed to save receipts to file after borrowing books.\n";
    }

    // Print summary receipts
    std::cout << "\n--- Transaction Receipt ---\n";
    std::cout << "Receipt Number: " << newReceipt.receiptNumber << "\n";
    std::cout << "User ID: " << newReceipt.userID << "\n";
    std::cout << "Borrowed Books:\n";
    for (const auto& id : newReceipt.borrowedBookIDs) {
        auto bookIt = find_if(books.begin(), books.end(), [&](const Book& b) {
            return b.ID == id;
            });
        if (bookIt != books.end()) {
            std::cout << " - ID: " << bookIt->ID << ", Title: " << bookIt->title << "\n";
        }
    }
    // Calculation of return date (after 7 days)
    time_t t = time(0) + (7 * 24 * 60 * 60);
    struct tm dueDate;
#ifdef _WIN32
    localtime_s(&dueDate, &t);
#else
    if (localtime_r(&t, &dueDate) == nullptr) {
        cerr << "Failed to calculate due date.\n";
    }
#endif
    std::cout << "Return Due Date: "
        << (1900 + dueDate.tm_year) << "-"
        << (1 + dueDate.tm_mon) << "-"
        << dueDate.tm_mday << "\n";
    std::cout << "--------------------------\n\n";
}
```



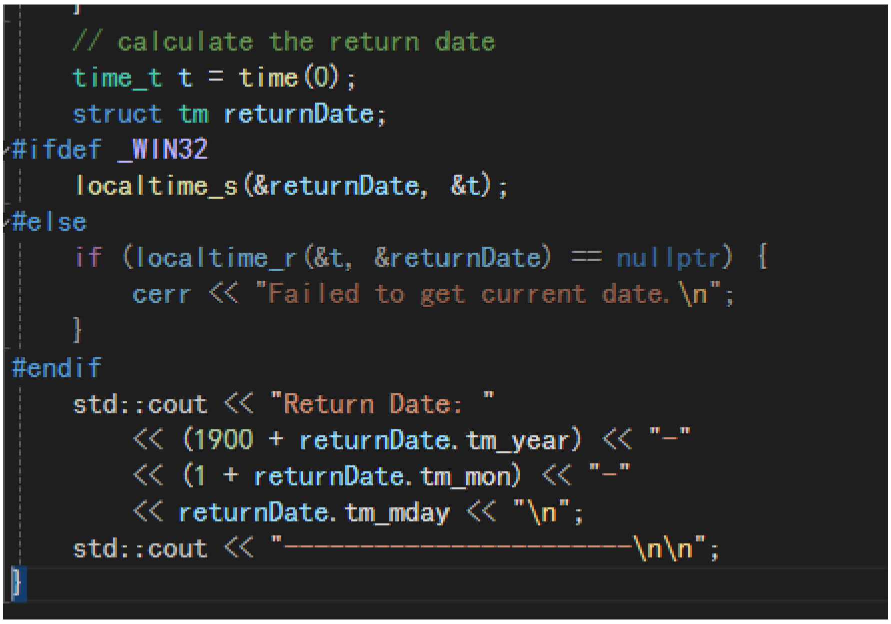
```
void returnBook(vector<Book>& books,
    vector<TransactionReceipt>& receipts,
    const string& userID) {

    // Collection of all unreturned books and their corresponding receipts and indexes
    struct ReturnInfo {
        TransactionReceipt* receipt;
        size_t bookIndex;
        int bookID;
    };

    vector<ReturnInfo> userBorrowedBooks;

    for (auto& receipt : receipts) {
        if (receipt.userID == userID && !receipt.returned) {
            for (size_t i = 0; i < receipt.borrowedBookIDs.size(); ++i) {
                userBorrowedBooks.push_back(ReturnInfo{ &receipt, i, receipt.borrowedBookIDs[i] });
            }
        }
    }

    if (userBorrowedBooks.empty()) {
        std::cout << "You have no borrowed books to return.\n\n";
        return;
    }

    // Used to store information about books returned by the user
    struct ReturnedBook {
        int bookID;
        std::string title;
    };
    vector<ReturnedBook> returnedBooks;

    bool continueReturning = true;

    while (continueReturning && !userBorrowedBooks.empty()) {
        // Show all returnable books
        std::cout << "\nYour Borrowed Books:\n";
        for (size_t i = 0; i < userBorrowedBooks.size(); ++i) {
            int bookID = userBorrowedBooks[i].bookID;
            auto bookIt = std::find_if(books.begin(), books.end(), [&](const Book& b) {
                return b.ID == bookID;
                });
            if (bookIt != books.end()) {
                std::cout << i + 1 << ". " << bookIt->title << " (ID: " << bookIt->ID << ")\n";
            }
        }
```



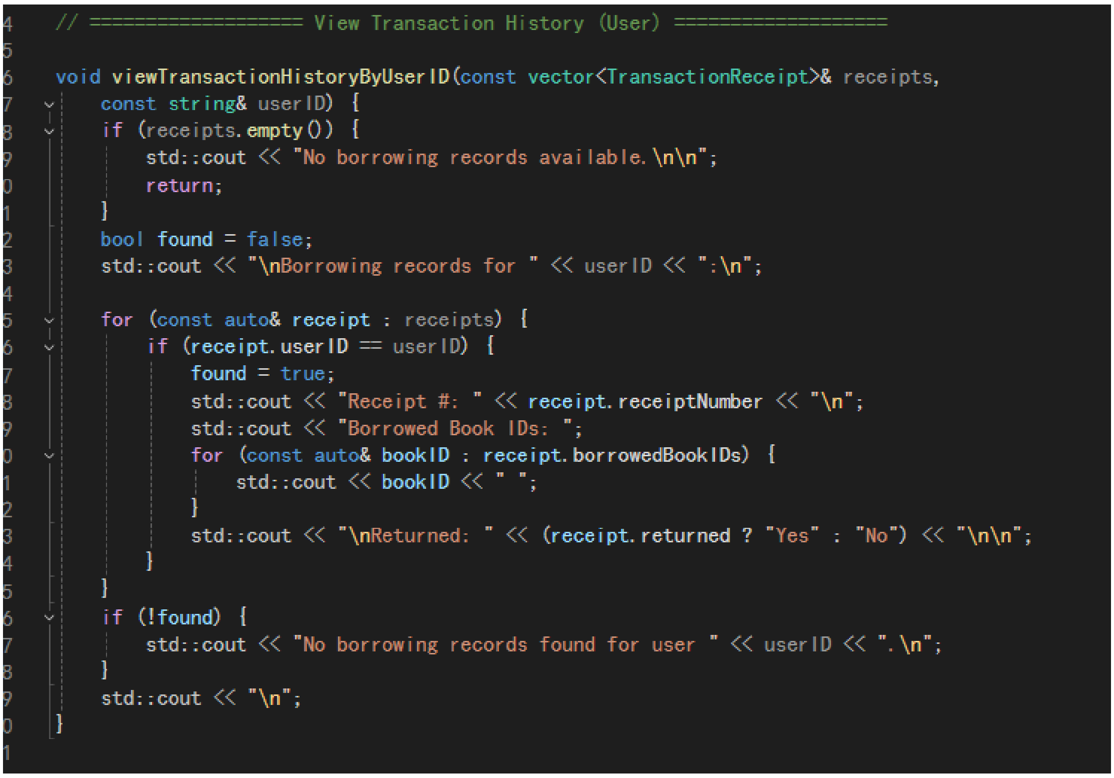
```
// User selects books to be returned
std::cout << "Enter the number of the book you want to return (or 0 to cancel): ";
int choice;
if (!(cin >> choice)) {
    std::cout << "Invalid input. Returning to User Control Panel.\n\n";
    clearInputBuffer();
    return;
}
clearInputBuffer();

if (choice == 0) {
    std::cout << "Return book operation cancelled.\n\n";
    break;
}

if (choice < 1 || choice > static_cast<int>(userBorrowedBooks.size())) {
    std::cout << "Invalid choice. Please try again.\n\n";
    continue;
}

// Get information about books to be returned
ReturnInfo selectedReturn = userBorrowedBooks[choice - 1];
int bookID = selectedReturn.bookID;

// Updating the availability of books
auto bookIt = std::find_if(books.begin(), books.end(), [&](const Book& b) {
    return b.ID == bookID;
    });

if (bookIt != books.end()) {
    bookIt->availability = true;
}
else {
    cerr << "Error: Book ID " << bookID << " not found in books list.\n";
    continue;
}


selectedReturn.receipt->borrowedBookIDs.erase(selectedReturn.receipt->borrowedBookIDs.begin() + selectedReturn.bookIndex);


if (selectedReturn.receipt->borrowedBookIDs.empty()) {
    selectedReturn.receipt->returned = true;
}

// Add returned book information to the returnedBooks list.
ReturnedBook rb;
rb.bookID = bookID;
rb.title = bookIt->title;
returnedBooks.push_back(rb);

// Removing Returned Books from userBorrowedBooks
userBorrowedBooks.erase(userBorrowedBooks.begin() + (choice - 1));


if (userBorrowedBooks.empty()) {
    std::cout << "All your borrowed books have been returned.\n\n";
    break;
}
```



```
    //  Input validation loop
    char moreBooks;
    bool validInput = false;

    while (!validInput) {
        std::cout << "Would you like to return another book? (y/n): ";
        cin >> moreBooks;
        clearInputBuffer();

        if (moreBooks == 'y' || moreBooks == 'Y') {
            validInput = true;
        }
        else if (moreBooks == 'n' || moreBooks == 'N') {
            validInput = true;
            continueReturning = false; // stop returning
        }
        else {
            std::cout << "Invalid input. Please enter 'y' or 'n'.\n";
        }
    }
}

if (returnedBooks.empty()) {
    std::cout << "No books were returned.\n\n";
    return;
}

// Save changes to file
try {
    if (!saveBooksToFile(books, "books.txt")) {
        throw runtime_error("Failed to save books to file.");
    }
    if (!saveReceiptsToFile(receipts, "receipts.txt")) {
        throw runtime_error("Failed to save receipts to file.");
    }
}
catch (const exception& e) {
    // Rollback Changes
    for (const auto& returnInfo : returnedBooks) {
        // Restoring the availability of books
        auto bookIt = std::find_if(books.begin(), books.end(), [&](const Book& b) {
            return b.ID == returnInfo.bookID;
            });
        if (bookIt != books.end()) {
            bookIt->availability = false;
        }


        for (auto& receipt : receipts) {
            if (receipt.userID == userID && !receipt.returned) {
                receipt.borrowedBookIDs.push_back(returnInfo.bookID);
                break;
            }
        }
    }
    cerr << "Transaction failed: " << e.what() << "\nRolling back changes.\n\n";
    return;
}

// Generate book return receipts
std::cout << "\n--- Return Receipt ---\n";
std::cout << "User ID: " << userID << "\n";
std::cout << "Books Returned:\n";
for (const auto& rb : returnedBooks) {
    std::cout << " - ID: " << rb.bookID << ", Title: " << rb.title << "\n";
}
// calculate the return date
```

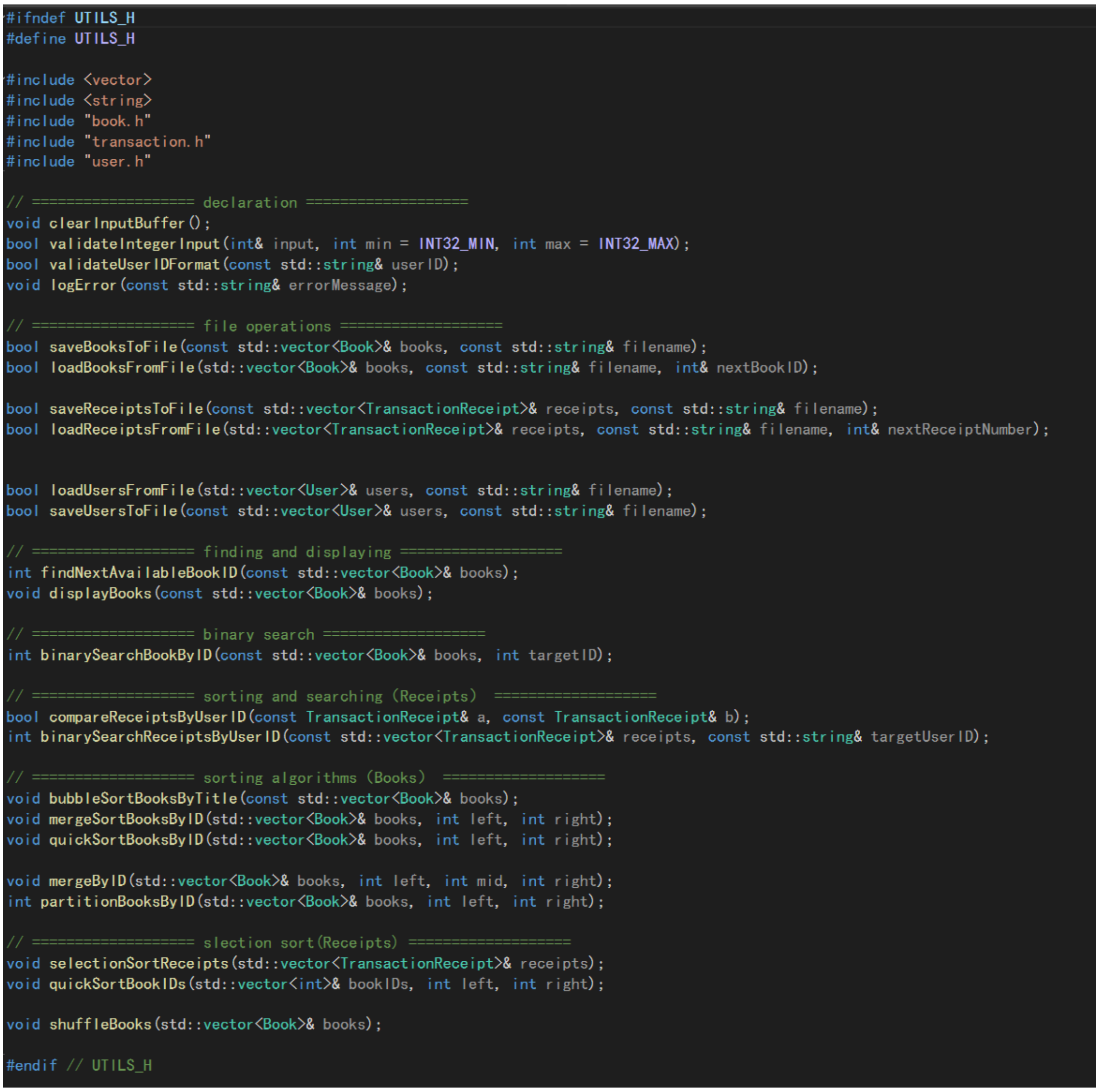


```
    }
    // calculate the return date
    time_t t = time(0);
    struct tm returnDate;
#ifdef _WIN32
    localtime_s(&returnDate, &t);
#else
    if (localtime_r(&t, &returnDate) == nullptr) {
        cerr << "Failed to get current date.\n";
    }
#endif
    std::cout << "Return Date: "
        << (1900 + returnDate.tm_year) << "-"
        << (1 + returnDate.tm_mon) << "-"
        << returnDate.tm_mday << "\n";
    std::cout << "----------------------\n\n";
}
```



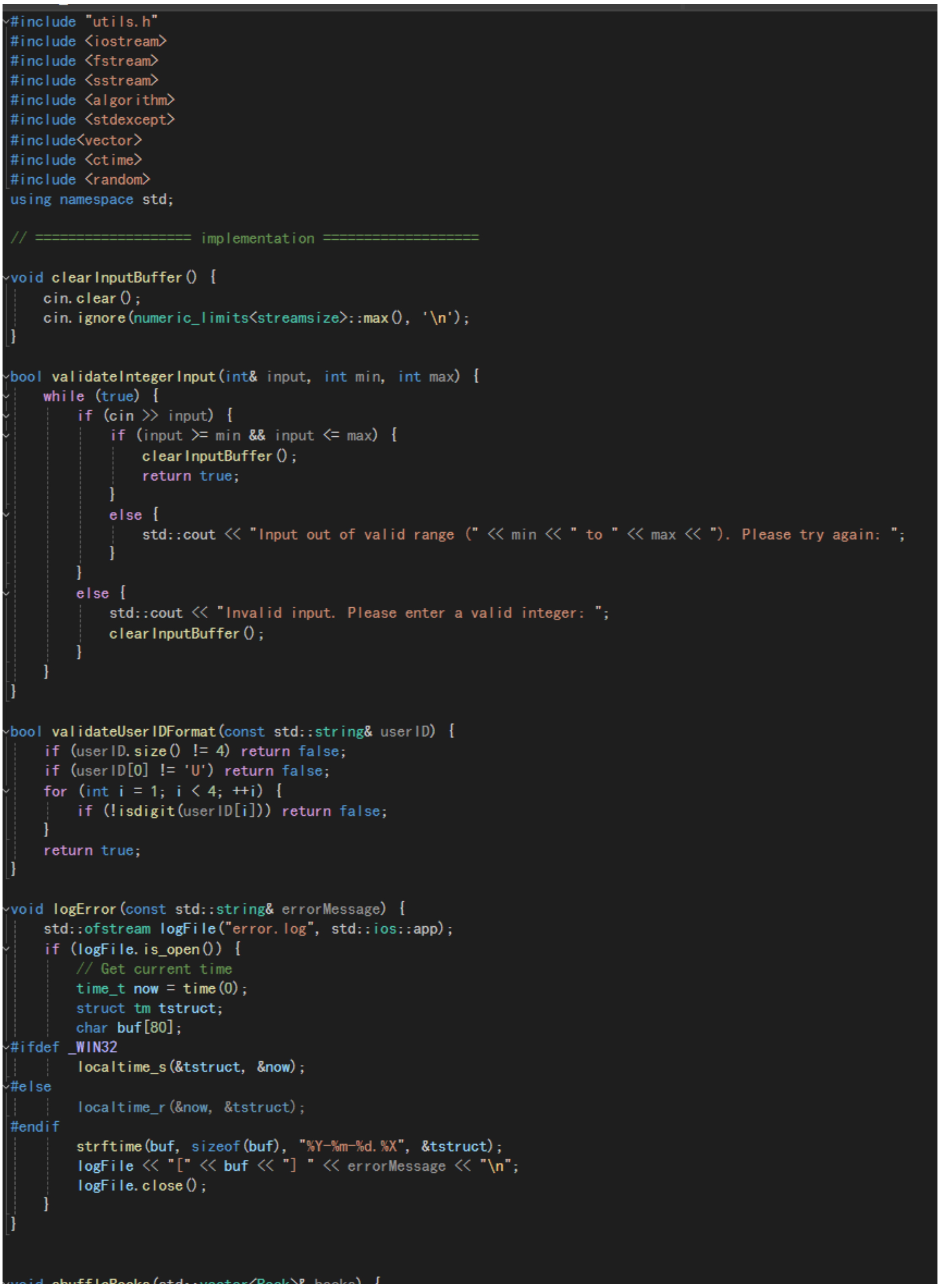
```
// ================== View Transaction History (User) ==================

void viewTransactionHistoryByUserID(const vector<TransactionReceipt>& receipts,
    const string& userID) {
    if (receipts.empty()) {
        std::cout << "No borrowing records available.\n\n";
        return;
    }
    bool found = false;
    std::cout << "\nBorrowing records for " << userID << ":\n";

    for (const auto& receipt : receipts) {
        if (receipt.userID == userID) {
            found = true;
            std::cout << "Receipt #: " << receipt.receiptNumber << "\n";
            std::cout << "Borrowed Book IDs: ";
            for (const auto& bookID : receipt.borrowedBookIDs) {
                std::cout << bookID << " ";
            }
            std::cout << "\nReturned: " << (receipt.returned ? "Yes" : "No") << "\n\n";
        }
    }
    if (!found) {
        std::cout << "No borrowing records found for user " << userID << ".\n";
    }
    std::cout << "\n";
}
```

Utils.h

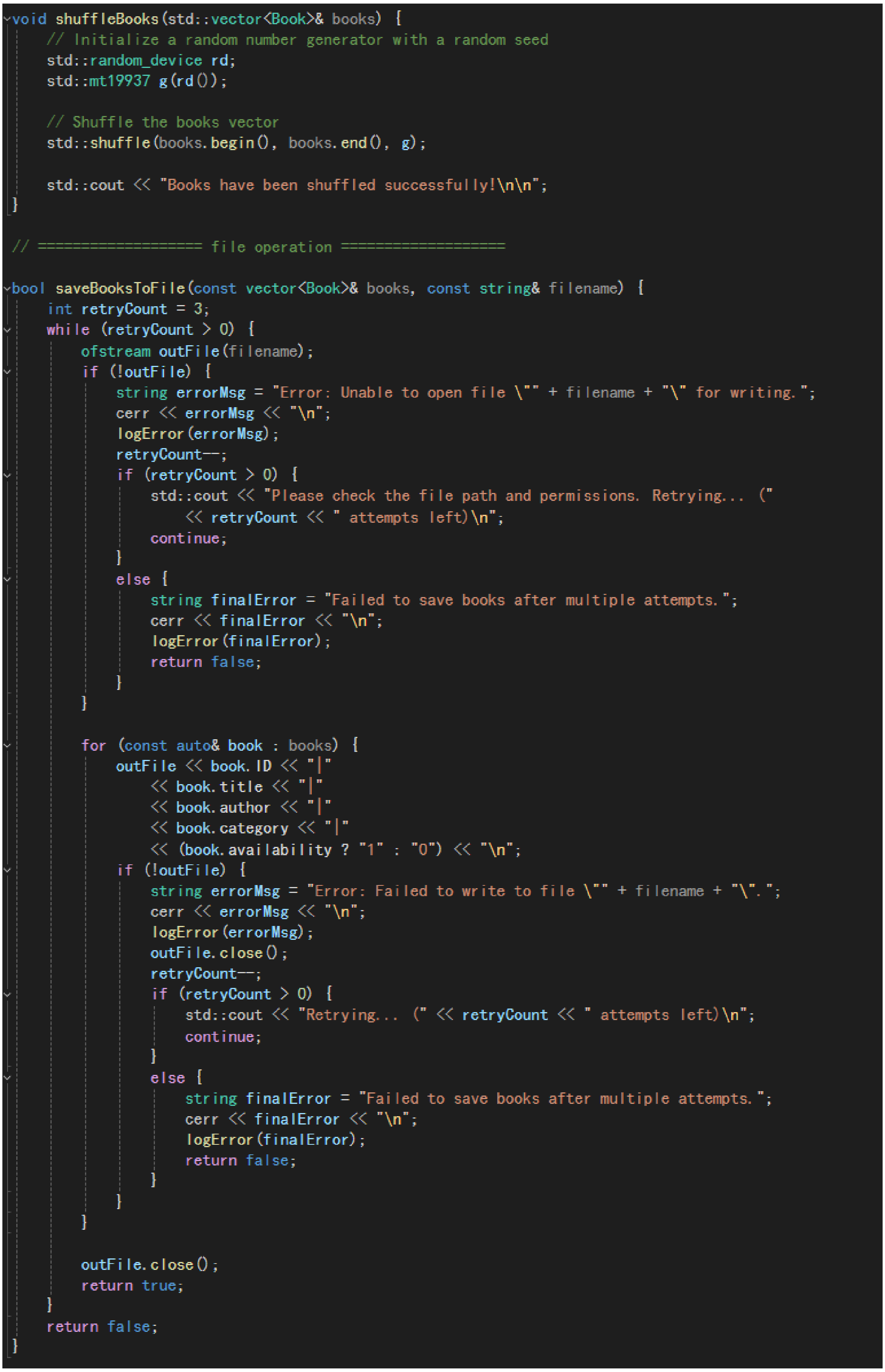


```cpp
#ifndef UTILS_H
#define UTILS_H

#include <vector>
#include <string>
#include "book.h"
#include "transaction.h"
#include "user.h"

// =================== declaration ===================
void clearInputBuffer();
bool validateIntegerInput(int& input, int min = INT32_MIN, int max = INT32_MAX);
bool validateUserIDFormat(const std::string& userID);
void logError(const std::string& errorMessage);

// =================== file operations ===================
bool saveBooksToFile(const std::vector<Book>& books, const std::string& filename);
bool loadBooksFromFile(std::vector<Book>& books, const std::string& filename, int& nextBookID);

bool saveReceiptsToFile(const std::vector<TransactionReceipt>& receipts, const std::string& filename);
bool loadReceiptsFromFile(std::vector<TransactionReceipt>& receipts, const std::string& filename, int& nextReceiptNumber);


bool loadUsersFromFile(std::vector<User>& users, const std::string& filename);
bool saveUsersToFile(const std::vector<User>& users, const std::string& filename);

// =================== finding and displaying ===================
int findNextAvailableBookID(const std::vector<Book>& books);
void displayBooks(const std::vector<Book>& books);

// =================== binary search ===================
int binarySearchBookByID(const std::vector<Book>& books, int targetID);

// =================== sorting and searching (Receipts)  ===================
bool compareReceiptsByUserID(const TransactionReceipt& a, const TransactionReceipt& b);
int binarySearchReceiptsByUserID(const std::vector<TransactionReceipt>& receipts, const std::string& targetUserID);

// =================== sorting algorithms (Books)  ===================
void bubbleSortBooksByTitle(const std::vector<Book>& books);
void mergeSortBooksByID(std::vector<Book>& books, int left, int right);
void quickSortBooksByID(std::vector<Book>& books, int left, int right);

void mergeByID(std::vector<Book>& books, int left, int mid, int right);
int partitionBooksByID(std::vector<Book>& books, int left, int right);

// =================== slection sort(Receipts) ===================
void selectionSortReceipts(std::vector<TransactionReceipt>& receipts);
void quickSortBookIDs(std::vector<int>& bookIDs, int left, int right);

void shuffleBooks(std::vector<Book>& books);

#endif // UTILS_H
```

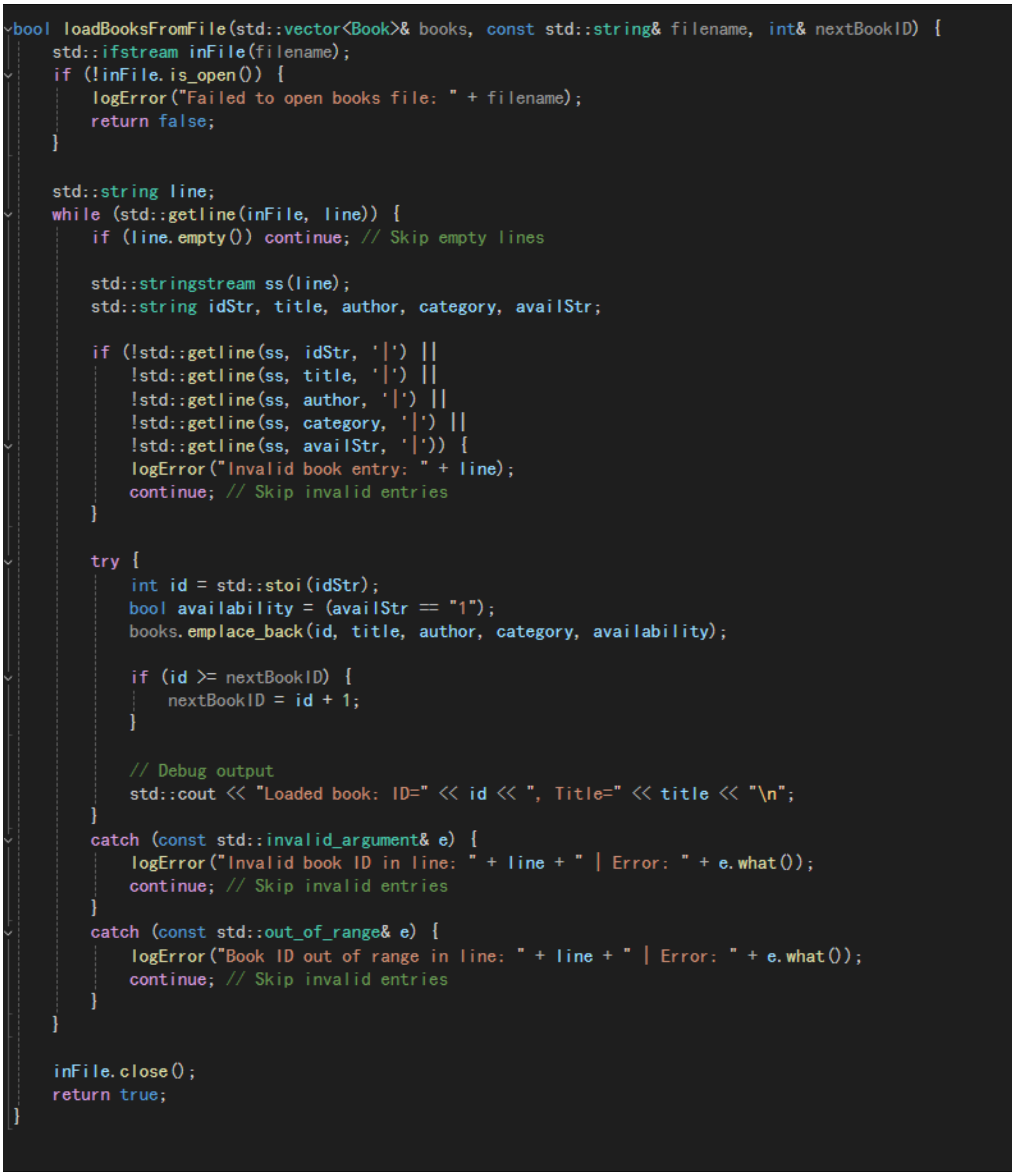
Utils.cpp



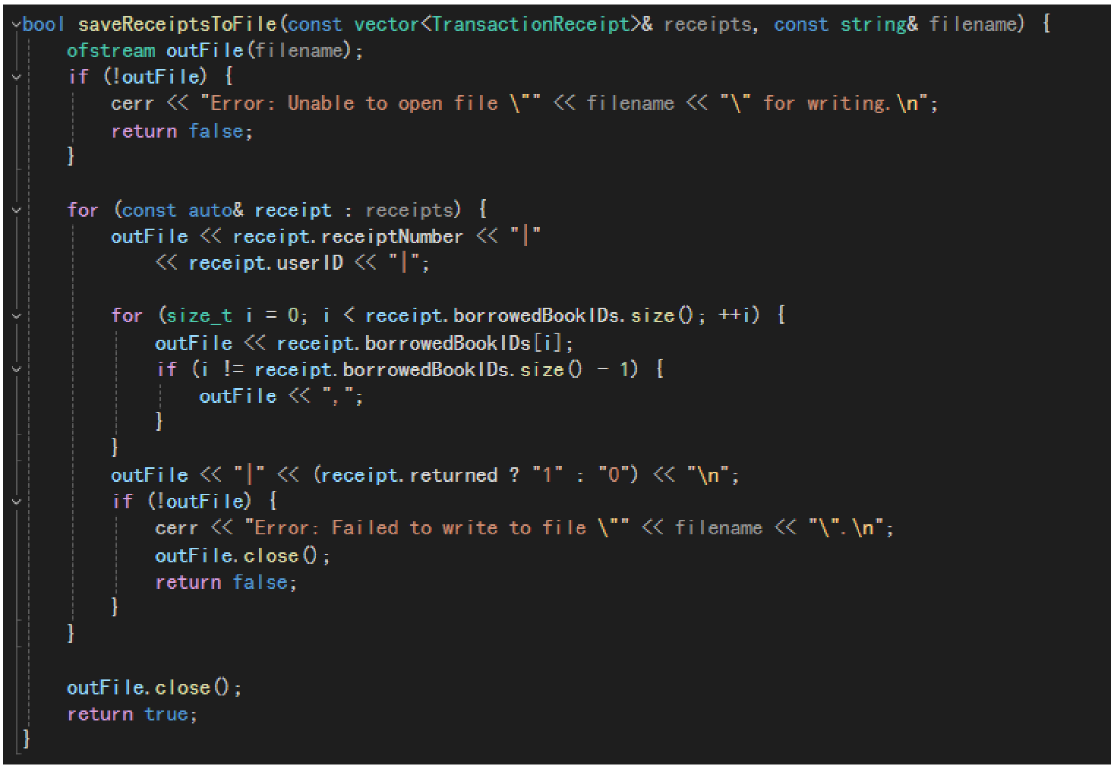
```cpp
#include "utils.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <stdexcept>
#include<vector>
#include <ctime>
#include <random>
using namespace std;

// ================== implementation ==================

void clearInputBuffer() {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

bool validateIntegerInput(int& input, int min, int max) {
    while (true) {
        if (cin >> input) {
            if (input >= min && input <= max) {
                clearInputBuffer();
                return true;
            }
            else {
                std::cout << "Input out of valid range (" << min << " to " << max << "). Please try again: ";
            }
        }
        else {
            std::cout << "Invalid input. Please enter a valid integer: ";
            clearInputBuffer();
        }
    }
}

bool validateUserIDFormat(const std::string& userID) {
    if (userID.size() != 4) return false;
    if (userID[0] != 'U') return false;
    for (int i = 1; i < 4; ++i) {
        if (!isdigit(userID[i])) return false;
    }
    return true;
}

void logError(const std::string& errorMessage) {
    std::ofstream logFile("error.log", std::ios::app);
    if (logFile.is_open()) {
        // Get current time
        time_t now = time(0);
        struct tm tstruct;
        char buf[80];
#ifdef _WIN32
        localtime_s(&tstruct, &now);
#else
        localtime_r(&now, &tstruct);
#endif
        strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);
        logFile << "[" << buf << "] " << errorMessage << "\n";
        logFile.close();
    }
}


void shuffleBooks(std::vector<Book>& books) {
```



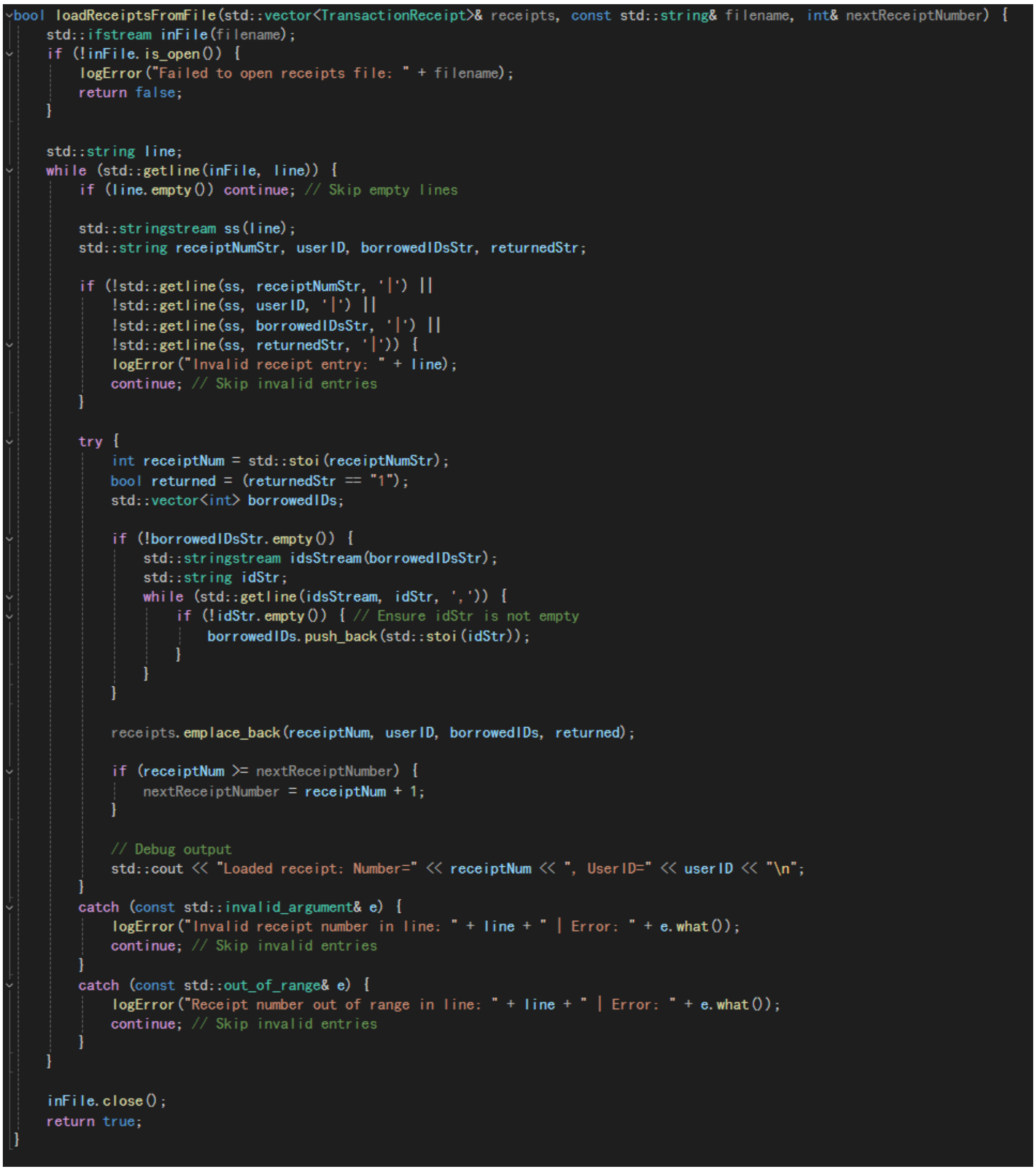
```
void shuffleBooks(std::vector<Book>& books) {
    // Initialize a random number generator with a random seed
    std::random_device rd;
    std::mt19937 g(rd());

    // Shuffle the books vector
    std::shuffle(books.begin(), books.end(), g);

    std::cout << "Books have been shuffled successfully!\n\n";
}

// =================== file operation ===================

bool saveBooksToFile(const vector<Book>& books, const string& filename) {
    int retryCount = 3;
    while (retryCount > 0) {
        ofstream outFile(filename);
        if (!outFile) {
            string errorMsg = "Error: Unable to open file \"" + filename + "\" for writing.";
            cerr << errorMsg << "\n";
            logError(errorMsg);
            retryCount--;
            if (retryCount > 0) {
                std::cout << "Please check the file path and permissions. Retrying... ("
                    << retryCount << " attempts left)\n";
                continue;
            }
            else {
                string finalError = "Failed to save books after multiple attempts.";
                cerr << finalError << "\n";
                logError(finalError);
                return false;
            }
        }

        for (const auto& book : books) {
            outFile << book.ID << "|"
                << book.title << "|"
                << book.author << "|"
                << book.category << "|"
                << (book.availability ? "1" : "0") << "\n";
            if (!outFile) {
                string errorMsg = "Error: Failed to write to file \"" + filename + "\".";
                cerr << errorMsg << "\n";
                logError(errorMsg);
                outFile.close();
                retryCount--;
                if (retryCount > 0) {
                    std::cout << "Retrying... (" << retryCount << " attempts left)\n";
                    continue;
                }
                else {
                    string finalError = "Failed to save books after multiple attempts.";
                    cerr << finalError << "\n";
                    logError(finalError);
                    return false;
                }
            }
        }

        outFile.close();
        return true;
    }
    return false;
}
```



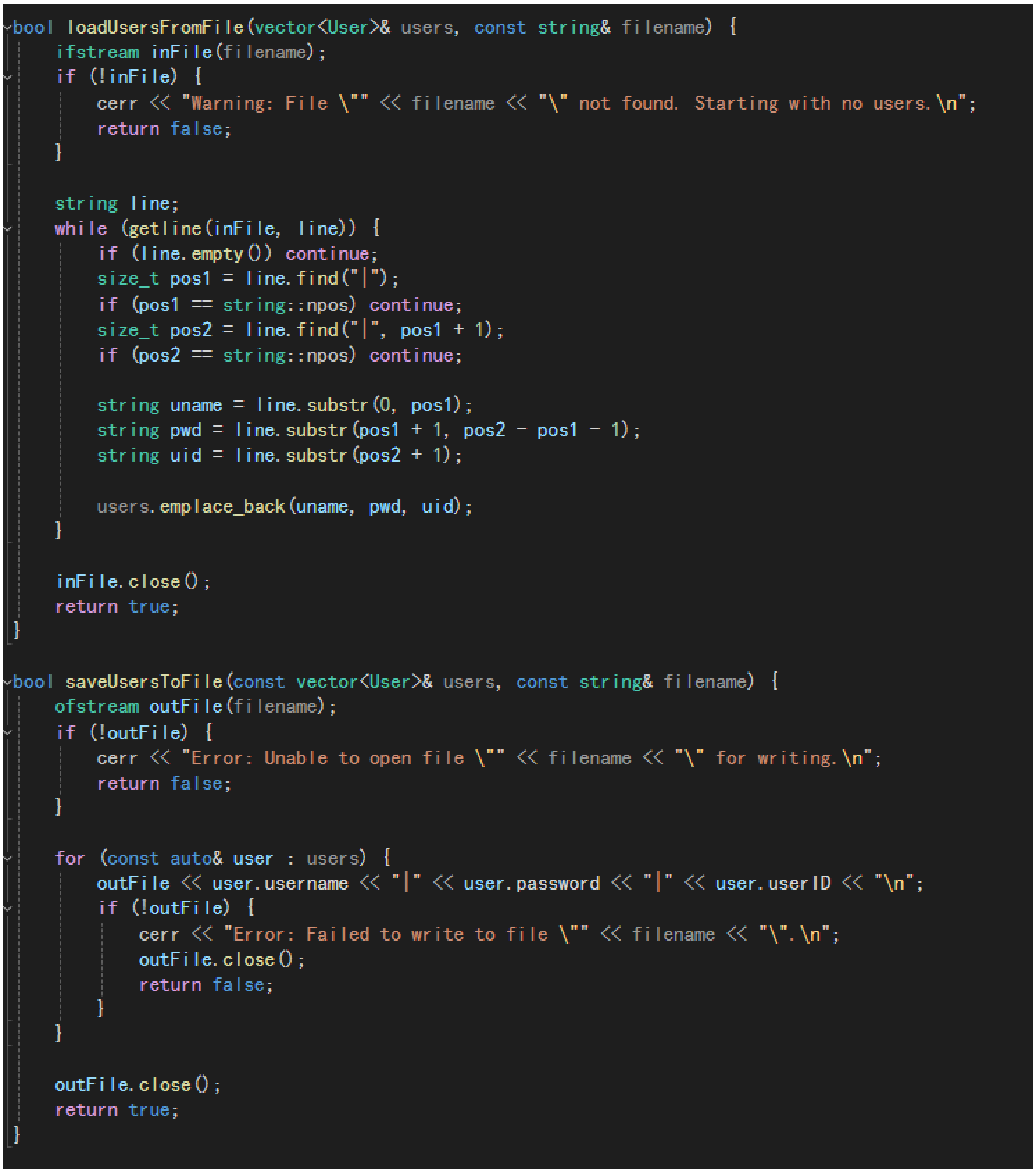
```
bool loadBooksFromFile(std::vector<Book>& books, const std::string& filename, int& nextBookID) {
    std::ifstream inFile(filename);
    if (!inFile.is_open()) {
        logError("Failed to open books file: " + filename);
        return false;
    }

    std::string line;
    while (std::getline(inFile, line)) {
        if (line.empty()) continue; // Skip empty lines

        std::stringstream ss(line);
        std::string idStr, title, author, category, availStr;

        if (!std::getline(ss, idStr, '|') ||
            !std::getline(ss, title, '|') ||
            !std::getline(ss, author, '|') ||
            !std::getline(ss, category, '|') ||
            !std::getline(ss, availStr, '|')) {
            logError("Invalid book entry: " + line);
            continue; // Skip invalid entries
        }

        try {
            int id = std::stoi(idStr);
            bool availability = (availStr == "1");
            books.emplace_back(id, title, author, category, availability);

            if (id >= nextBookID) {
                nextBookID = id + 1;
            }

            // Debug output
            std::cout << "Loaded book: ID=" << id << ", Title=" << title << "\n";
        }
        catch (const std::invalid_argument& e) {
            logError("Invalid book ID in line: " + line + " | Error: " + e.what());
            continue; // Skip invalid entries
        }
        catch (const std::out_of_range& e) {
            logError("Book ID out of range in line: " + line + " | Error: " + e.what());
            continue; // Skip invalid entries
        }
    }

    inFile.close();
    return true;
}
```



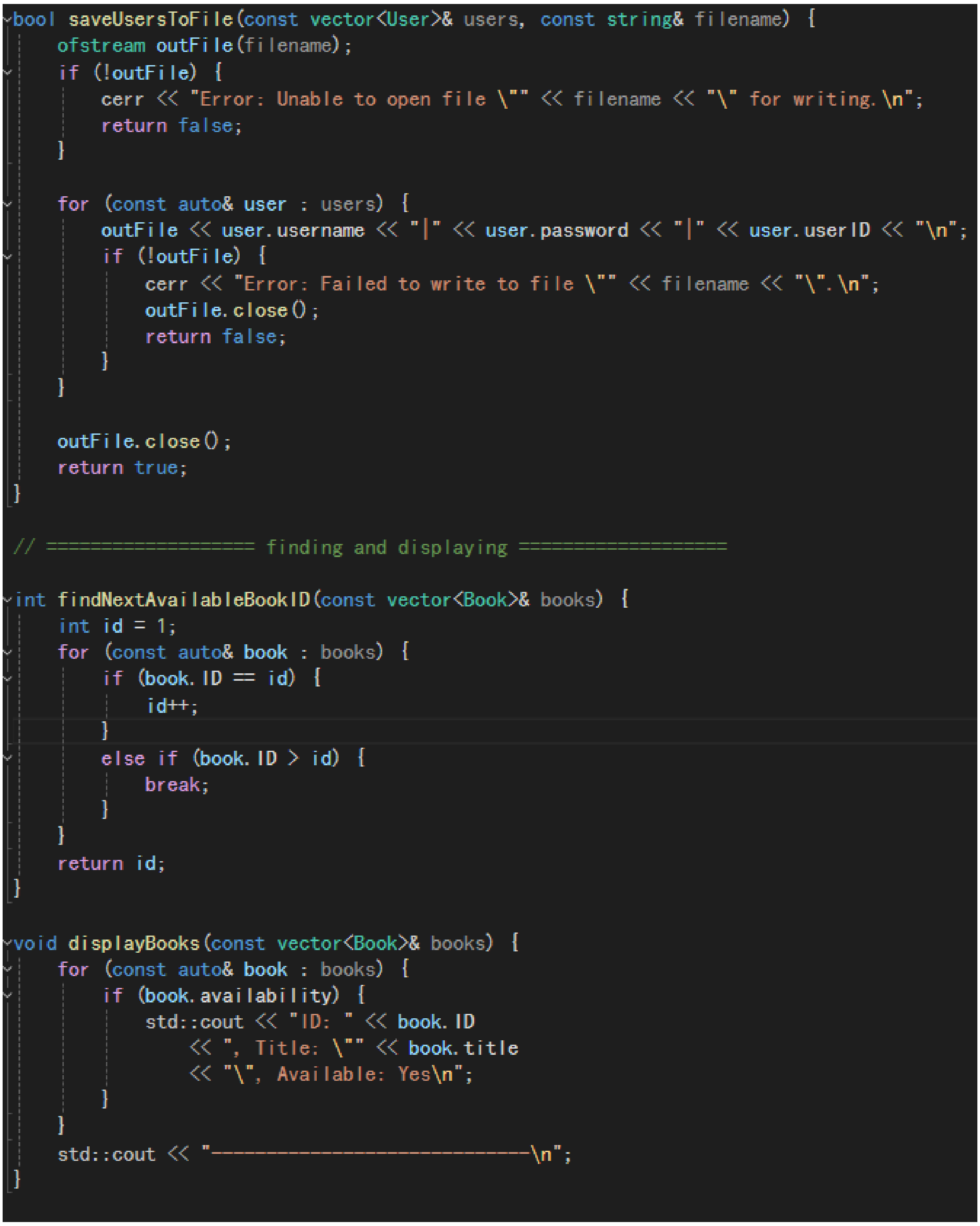
```
bool saveReceiptsToFile(const vector<TransactionReceipt>& receipts, const string& filename) {
    ofstream outFile(filename);
    if (!outFile) {
        cerr << "Error: Unable to open file \"" << filename << "\" for writing.\n";
        return false;
    }

    for (const auto& receipt : receipts) {
        outFile << receipt.receiptNumber << "|"
            << receipt.userID << "|";

        for (size_t i = 0; i < receipt.borrowedBookIDs.size(); ++i) {
            outFile << receipt.borrowedBookIDs[i];
            if (i != receipt.borrowedBookIDs.size() - 1) {
                outFile << ",";
            }
        }
        outFile << "|" << (receipt.returned ? "1" : "0") << "\n";
        if (!outFile) {
            cerr << "Error: Failed to write to file \"" << filename << "\".\n";
            outFile.close();
            return false;
        }
    }

    outFile.close();
    return true;
}
```



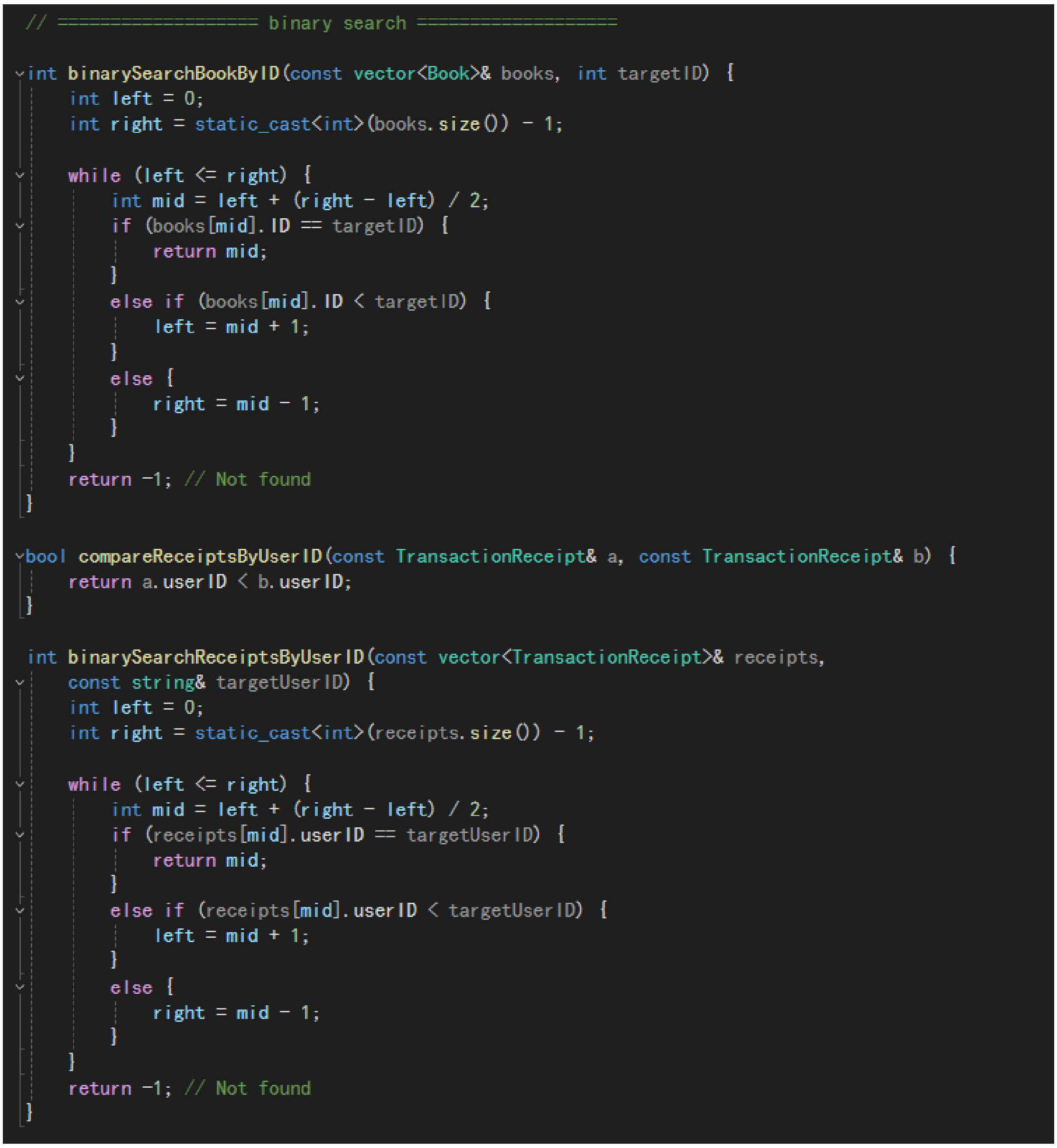
```
bool loadReceiptsFromFile(std::vector<TransactionReceipt>& receipts, const std::string& filename, int& nextReceiptNumber) {
    std::ifstream inFile(filename);
    if (!inFile.is_open()) {
        logError("Failed to open receipts file: " + filename);
        return false;
    }

    std::string line;
    while (std::getline(inFile, line)) {
        if (line.empty()) continue; // Skip empty lines

        std::stringstream ss(line);
        std::string receiptNumStr, userID, borrowedIDsStr, returnedStr;

        if (!std::getline(ss, receiptNumStr, '|') ||
            !std::getline(ss, userID, '|') ||
            !std::getline(ss, borrowedIDsStr, '|') ||
            !std::getline(ss, returnedStr, '|')) {
            logError("Invalid receipt entry: " + line);
            continue; // Skip invalid entries
        }

        try {
            int receiptNum = std::stoi(receiptNumStr);
            bool returned = (returnedStr == "1");
            std::vector<int> borrowedIDs;

            if (!borrowedIDsStr.empty()) {
                std::stringstream idsStream(borrowedIDsStr);
                std::string idStr;
                while (std::getline(idsStream, idStr, ',')) {
                    if (!idStr.empty()) { // Ensure idStr is not empty
                        borrowedIDs.push_back(std::stoi(idStr));
                    }
                }
            }

            receipts.emplace_back(receiptNum, userID, borrowedIDs, returned);

            if (receiptNum >= nextReceiptNumber) {
                nextReceiptNumber = receiptNum + 1;
            }

            // Debug output
            std::cout << "Loaded receipt: Number=" << receiptNum << ", UserID=" << userID << "\n";
        }
        catch (const std::invalid_argument& e) {
            logError("Invalid receipt number in line: " + line + " | Error: " + e.what());
            continue; // Skip invalid entries
        }
        catch (const std::out_of_range& e) {
            logError("Receipt number out of range in line: " + line + " | Error: " + e.what());
            continue; // Skip invalid entries
        }
    }

    inFile.close();
    return true;
}
```



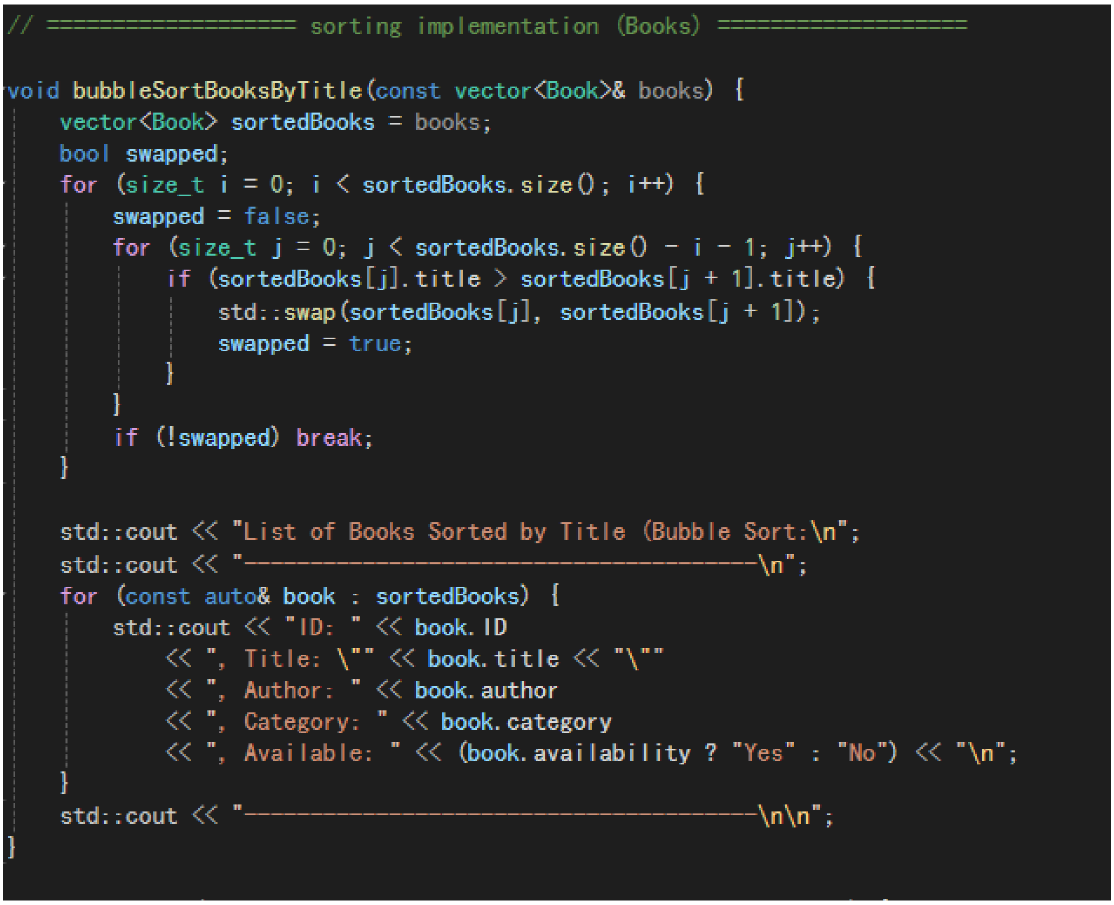
```cpp
bool loadUsersFromFile(vector<User>& users, const string& filename) {
    ifstream inFile(filename);
    if (!inFile) {
        cerr << "Warning: File \"" << filename << "\" not found. Starting with no users.\n";
        return false;
    }

    string line;
    while (getline(inFile, line)) {
        if (line.empty()) continue;
        size_t pos1 = line.find("|");
        if (pos1 == string::npos) continue;
        size_t pos2 = line.find("|", pos1 + 1);
        if (pos2 == string::npos) continue;

        string uname = line.substr(0, pos1);
        string pwd = line.substr(pos1 + 1, pos2 - pos1 - 1);
        string uid = line.substr(pos2 + 1);

        users.emplace_back(uname, pwd, uid);
    }

    inFile.close();
    return true;
}

bool saveUsersToFile(const vector<User>& users, const string& filename) {
    ofstream outFile(filename);
    if (!outFile) {
        cerr << "Error: Unable to open file \"" << filename << "\" for writing.\n";
        return false;
    }

    for (const auto& user : users) {
        outFile << user.username << "|" << user.password << "|" << user.userID << "\n";
        if (!outFile) {
            cerr << "Error: Failed to write to file \"" << filename << "\".\n";
            outFile.close();
            return false;
        }
    }

    outFile.close();
    return true;
}
```



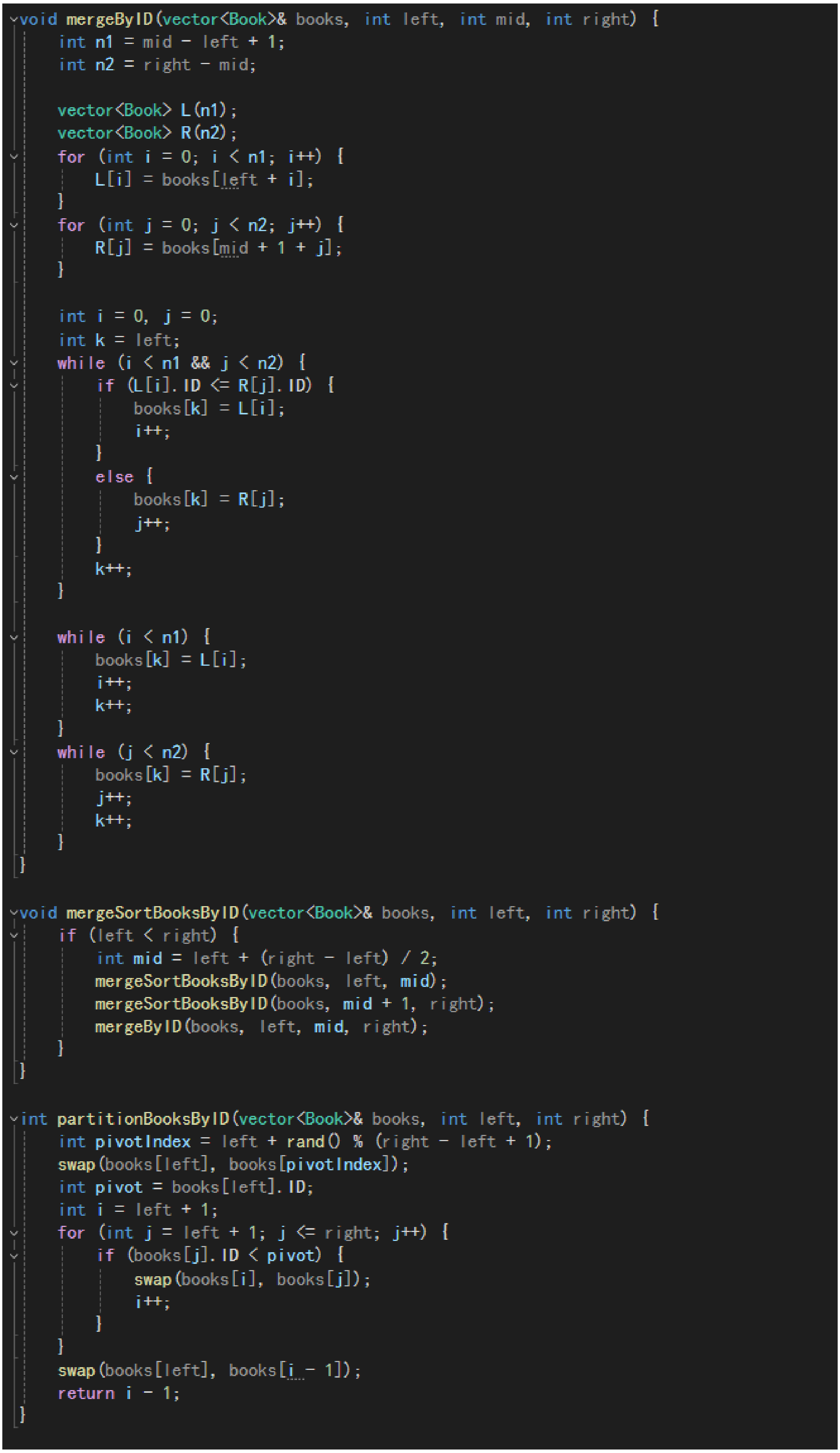
```cpp
bool saveUsersToFile(const vector<User>& users, const string& filename) {
    ofstream outFile(filename);
    if (!outFile) {
        cerr << "Error: Unable to open file \"" << filename << "\" for writing.\n";
        return false;
    }

    for (const auto& user : users) {
        outFile << user.username << "|" << user.password << "|" << user.userID << "\n";
        if (!outFile) {
            cerr << "Error: Failed to write to file \"" << filename << "\".\n";
            outFile.close();
            return false;
        }
    }

    outFile.close();
    return true;
}

// =================== finding and displaying ===================

int findNextAvailableBookID(const vector<Book>& books) {
    int id = 1;
    for (const auto& book : books) {
        if (book.ID == id) {
            id++;
        }
        else if (book.ID > id) {
            break;
        }
    }
    return id;
}

void displayBooks(const vector<Book>& books) {
    for (const auto& book : books) {
        if (book.availability) {
            std::cout << "ID: " << book.ID
                << ", Title: \"" << book.title
                << "\", Available: Yes\n";
        }
    }
    std::cout << "-----------------------------\n";
}
```



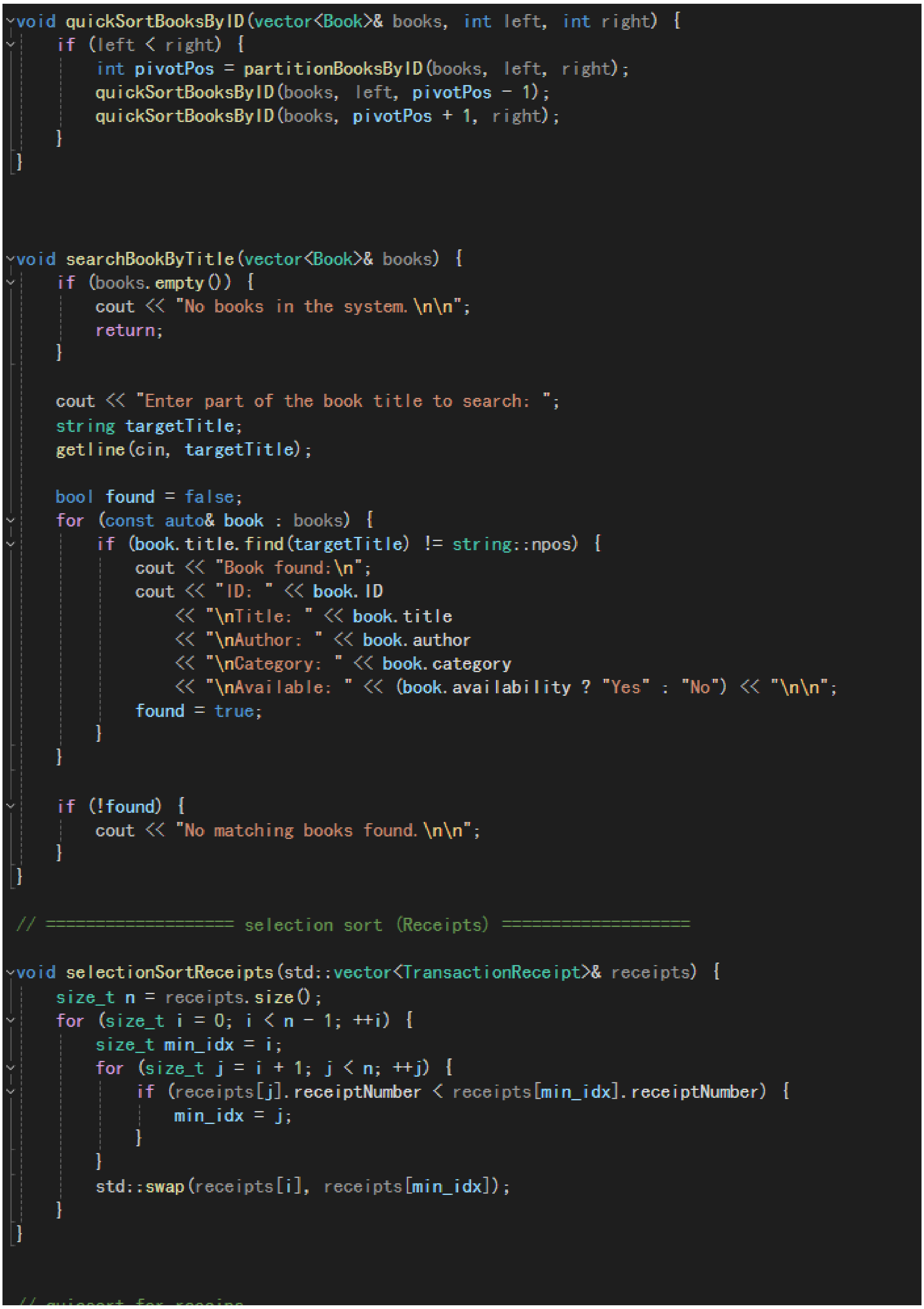
```
// =================== binary search ===================

int binarySearchBookByID(const vector<Book>& books, int targetID) {
    int left = 0;
    int right = static_cast<int>(books.size()) - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (books[mid].ID == targetID) {
            return mid;
        }
        else if (books[mid].ID < targetID) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1; // Not found
}

bool compareReceiptsByUserID(const TransactionReceipt& a, const TransactionReceipt& b) {
    return a.userID < b.userID;
}

int binarySearchReceiptsByUserID(const vector<TransactionReceipt>& receipts,
    const string& targetUserID) {
    int left = 0;
    int right = static_cast<int>(receipts.size()) - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (receipts[mid].userID == targetUserID) {
            return mid;
        }
        else if (receipts[mid].userID < targetUserID) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1; // Not found
}
```



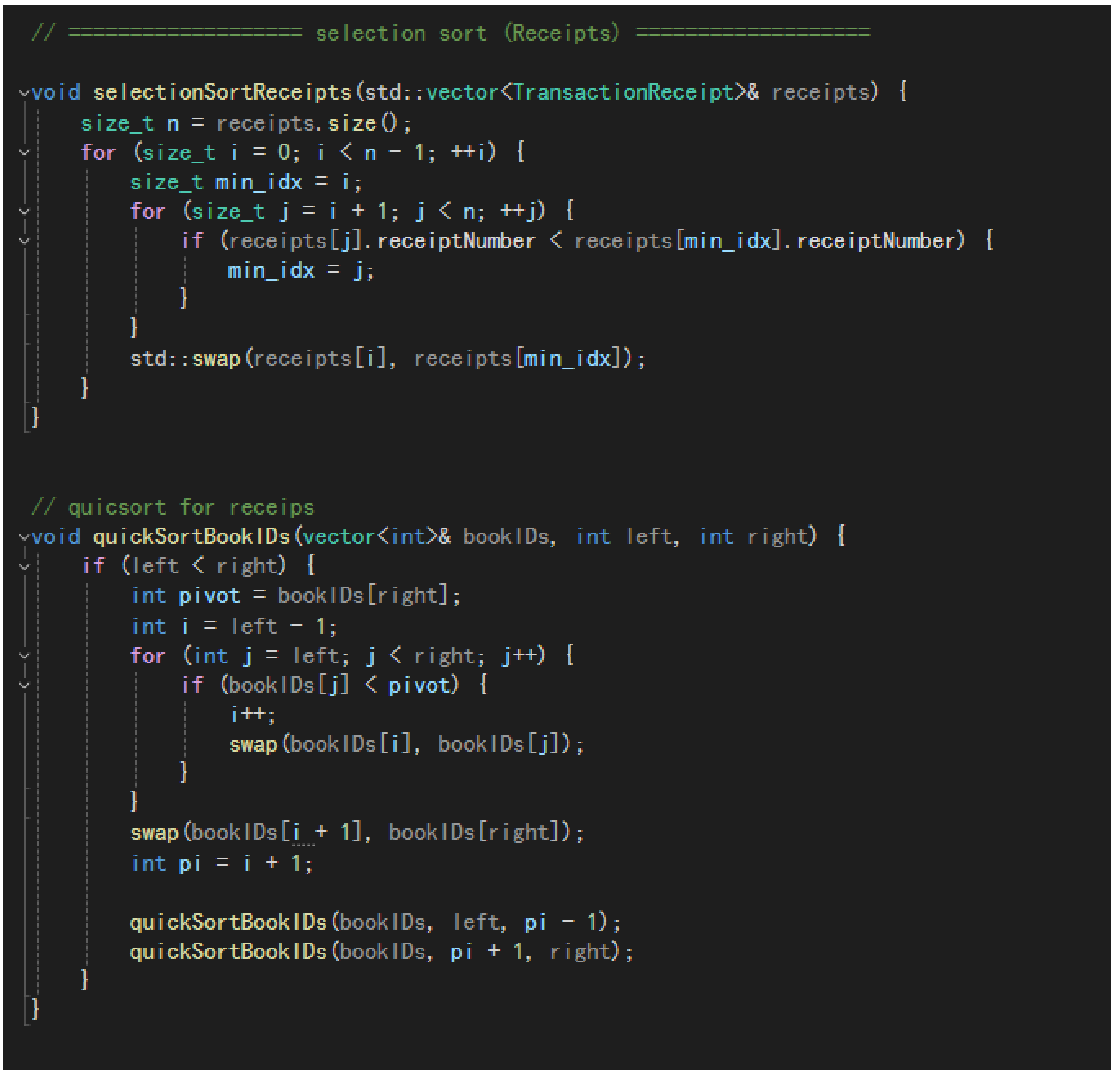
```cpp
// =================== sorting implementation (Books) ===================

void bubbleSortBooksByTitle(const vector<Book>& books) {
    vector<Book> sortedBooks = books;
    bool swapped;
    for (size_t i = 0; i < sortedBooks.size(); i++) {
        swapped = false;
        for (size_t j = 0; j < sortedBooks.size() - i - 1; j++) {
            if (sortedBooks[j].title > sortedBooks[j + 1].title) {
                std::swap(sortedBooks[j], sortedBooks[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }

    std::cout << "List of Books Sorted by Title (Bubble Sort:\n";
    std::cout << "---------------------------------------\n";
    for (const auto& book : sortedBooks) {
        std::cout << "ID: " << book.ID
            << ", Title: \"" << book.title << "\""
            << ", Author: " << book.author
            << ", Category: " << book.category
            << ", Available: " << (book.availability ? "Yes" : "No") << "\n";
    }
    std::cout << "---------------------------------------\n\n";
}
```



```
void mergeByID(vector<Book>& books, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<Book> L(n1);
    vector<Book> R(n2);
    for (int i = 0; i < n1; i++) {
        L[i] = books[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = books[mid + 1 + j];
    }

    int i = 0, j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (L[i].ID <= R[j].ID) {
            books[k] = L[i];
            i++;
        }
        else {
            books[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        books[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        books[k] = R[j];
        j++;
        k++;
    }
}

void mergeSortBooksByID(vector<Book>& books, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortBooksByID(books, left, mid);
        mergeSortBooksByID(books, mid + 1, right);
        mergeByID(books, left, mid, right);
    }
}

int partitionBooksByID(vector<Book>& books, int left, int right) {
    int pivotIndex = left + rand() % (right - left + 1);
    swap(books[left], books[pivotIndex]);
    int pivot = books[left].ID;
    int i = left + 1;
    for (int j = left + 1; j <= right; j++) {
        if (books[j].ID < pivot) {
            swap(books[i], books[j]);
            i++;
        }
    }
    swap(books[left], books[i - 1]);
    return i - 1;
}
```

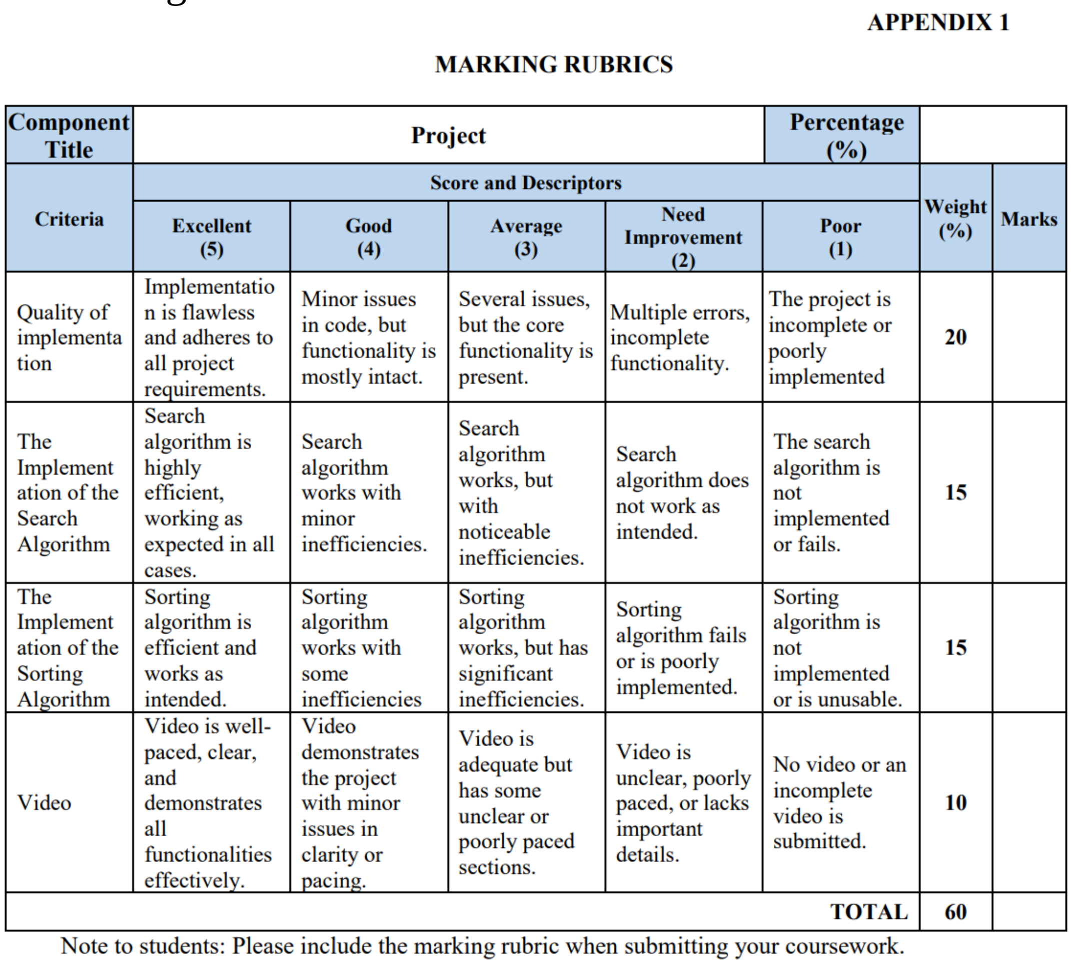


```
void quickSortBooksByID(vector<Book>& books, int left, int right) {
    if (left < right) {
        int pivotPos = partitionBooksByID(books, left, right);
        quickSortBooksByID(books, left, pivotPos - 1);
        quickSortBooksByID(books, pivotPos + 1, right);
    }
}



void searchBookByTitle(vector<Book>& books) {
    if (books.empty()) {
        cout << "No books in the system.\n\n";
        return;
    }

    cout << "Enter part of the book title to search: ";
    string targetTitle;
    getline(cin, targetTitle);

    bool found = false;
    for (const auto& book : books) {
        if (book.title.find(targetTitle) != string::npos) {
            cout << "Book found:\n";
            cout << "ID: " << book.ID
                << "\nTitle: " << book.title
                << "\nAuthor: " << book.author
                << "\nCategory: " << book.category
                << "\nAvailable: " << (book.availability ? "Yes" : "No") << "\n\n";
            found = true;
        }
    }

    if (!found) {
        cout << "No matching books found.\n\n";
    }
}

// ================== selection sort (Receipts) ==================

void selectionSortReceipts(std::vector<TransactionReceipt>& receipts) {
    size_t n = receipts.size();
    for (size_t i = 0; i < n - 1; ++i) {
        size_t min_idx = i;
        for (size_t j = i + 1; j < n; ++j) {
            if (receipts[j].receiptNumber < receipts[min_idx].receiptNumber) {
                min_idx = j;
            }
        }
        std::swap(receipts[i], receipts[min_idx]);
    }
}


// quicsort for receips
```



```
// =================== selection sort (Receipts) ===================

void selectionSortReceipts(std::vector<TransactionReceipt>& receipts) {
    size_t n = receipts.size();
    for (size_t i = 0; i < n - 1; ++i) {
        size_t min_idx = i;
        for (size_t j = i + 1; j < n; ++j) {
            if (receipts[j].receiptNumber < receipts[min_idx].receiptNumber) {
                min_idx = j;
            }
        }
        std::swap(receipts[i], receipts[min_idx]);
    }
}


// quicsort for receips
void quickSortBookIDs(vector<int>& bookIDs, int left, int right) {
    if (left < right) {
        int pivot = bookIDs[right];
        int i = left - 1;
        for (int j = left; j < right; j++) {
            if (bookIDs[j] < pivot) {
                i++;
                swap(bookIDs[i], bookIDs[j]);
            }
        }
        swap(bookIDs[i + 1], bookIDs[right]);
        int pi = i + 1;

        quickSortBookIDs(bookIDs, left, pi - 1);
        quickSortBookIDs(bookIDs, pi + 1, right);
    }
}
```

10. Marking Rubric



APPENDIX 1

MARKING RUBRICS

Component Title	Project				Percentage (%)		
Criteria	Score and Descriptors					Weight (%)	Marks
	Excellent (5)	Good (4)	Average (3)	Need Improvement (2)	Poor (1)		
Quality of implementation	Implementation is flawless and adheres to all project requirements.	Minor issues in code, but functionality is mostly intact.	Several issues, but the core functionality is present.	Multiple errors, incomplete functionality.	The project is incomplete or poorly implemented	20	
The Implementation of the Search Algorithm	Search algorithm is highly efficient, working as expected in all cases.	Search algorithm works with minor inefficiencies.	Search algorithm works, but with noticeable inefficiencies.	Search algorithm does not work as intended.	The search algorithm is not implemented or fails.	15	
The Implementation of the Sorting Algorithm	Sorting algorithm is efficient and works as intended.	Sorting algorithm works with some inefficiencies	Sorting algorithm works, but has significant inefficiencies.	Sorting algorithm fails or is poorly implemented.	Sorting algorithm is not implemented or is unusable.	15	
Video	Video is well-paced, clear, and demonstrates all functionalities effectively.	Video demonstrates the project with minor issues in clarity or pacing.	Video is adequate but has some unclear or poorly paced sections.	Video is unclear, poorly paced, or lacks important details.	No video or an incomplete video is submitted.	10	
					TOTAL	60	

Note to students: Please include the marking rubric when submitting your coursework.

Component Title	**Report and Individual work**				**Percentage (%)**		
Criteria	**Score and Descriptors**					**Weight (%)**	**Marks**
	Excellent (5)	**Good (4)**	**Average (3)**	**Need Improvement (2)**	**Poor (1)**		
Quality and the Report Format	The report is well-organized, clearly written, and follows the required format with no errors.	The report is organized and follows the format with few minor issues.	The report is readable, but lacks clarity or has formatting issues.	The report lacks organization or has significant formatting issues.	The report is poorly structured and difficult to follow.	**10**	
The Justification for Choosing the Search Algorithm and the Comparison Between Them	Clear, well-reasoned justification with thorough comparison of multiple search algorithms.	Good justification with comparison of at least two algorithms.	Justification is provided but lacks depth or clarity in comparison	Justification is unclear or lacks a solid comparison of algorithms.	No clear justification or comparison of search algorithms.	**10**	
The Justification for Choosing the Sort Algorithms and the Comparison Between Them	Well-argued justification with detailed comparison of sorting algorithms	Clear justification and comparison, but lacks some depth	Justification is present but does not fully explain the choice or comparison.	Justification is weak, lacks clarity, or does not compare sorting algorithms effectively.	No justification or comparison of sorting algorithms.	**10**	
Each Student's Contribution to the Project	Clear and detailed description of each student’s contribution with well-documented roles.	Descriptions of contributions are clear but could provide more detail.	Contributions are mentioned, but lack detail or clarity.	Contributions are vague or incomplete.	No clear description of contributions or lacks details.	**10**	
						40	

Note to students: Please include the marking rubric when submitting your coursework.