CMPE 126 Fall 2024
Final Project Assignment Lab Report
Instructed by Professor Mortezair
Submitted by Alex Garcia
Julian Zhuang
Newton Tran
Submission date: 12/09/2024

SJSU SAN JOSÉ STATE UNIVERSITY

Final Project Assignment Lab Report

Abstract:

To solidify an understanding of all the material students learned in CMPE 126 Data Structures and Algorithms, the group was assigned to incorporate queues, stacks, linked lists, binary trees, hashing, STL, sorting, and searching algorithms into a single overarching program that uses multiple of these concepts. Through various stages of creative implementation, the group successfully created a Library Management System that uses queues and linked lists, vectors, and multiple sorting algorithms to manage all aspects of the library.

Introduction:

For the main focus of this project assignment, the group set themselves a goal to create a simple and easy-to-use Library Management System. The group divided the work into three overarching sections: Book Management, User Management, and a Borrow/Return System. To start, the group managed books using vectors to organize and keep track of books listed in the library. Multiple implementations worked throughout these vectors, including functions to add to, search through, and sort the books. For sorting the books, while keeping time complexity and space complexity in mind, the group found the best 3 sorting algorithms to use for sorting by book ID, book title, and author: STL sort, quick sort, and bubble sort respectively.

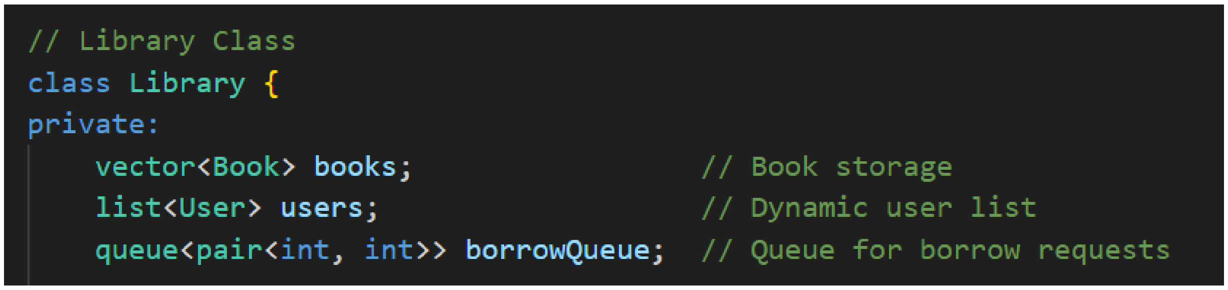
Body:

This program creates a library management system where users can interact with books and manage their borrowing process. The Library class organizes books using a vector, users in a list, and handles borrow requests with a queue. Users can add new books to the library with addBook by providing an ID, title, and author. Similarly, they can add themselves as library members using addUser, supplying their ID and name.

The system allows users to remove themselves with removeUser, which finds their ID in the list and deletes it. To verify membership, isUserExist checks if a specific user ID is present. Borrowing is handled through borrowBook, where users request books by providing their user ID and the book's ID. These requests are stored in a queue and processed later. During processing, the program checks if the requested book exists and is available, then marks it as unavailable if successfully borrowed.

Books can be searched by title using searchBookByTitle, which scans the vector to find a match. Sorting books is possible by ID, author, or title using algorithms like std::sort, bubble sort, and quicksort, respectively. The library can also randomize the book order with randomizeBooks for variety. Users can view all books and their availability status with displayBooks. The main function demonstrates these features, guiding users through adding books, registering users, making borrow requests, and performing search and sort operations, showcasing how the library system works interactively.

Data Structures or STL Containers:

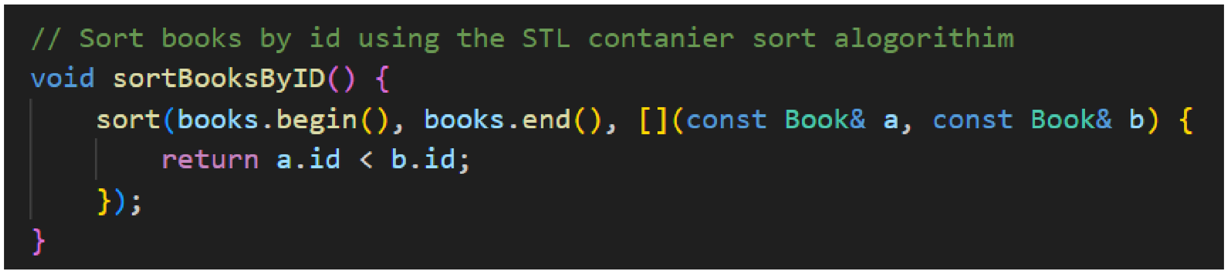


```cpp
// Library Class
class Library {
private:
    vector<Book> books;                  // Book storage
    list<User> users;                    // Dynamic user list
    queue<pair<int, int>> borrowQueue;   // Queue for borrow requests
```

We apply vectors to store and manage the list of books, which provides dynamic resizing, allowing books to be added easily. For example, books are added (addBook), searched (searchBookByTitle), and sorted (sortBooksByID, sortBooksByAuthor, sortBooksByTitle). STL list, whose implementation is linked list, provides advantages in insertion and removal of the user data. The examples of its usage are users are added (addUser), displayed (displayUsers), checked for existence (isUserExist), and removed (removeUser). Finally, queue handles borrow requests in a First-In-First-Out (FIFO) manner, which is good for sequential tasks like (borrowBook) and (processBorrowRequests).

Algorithms:

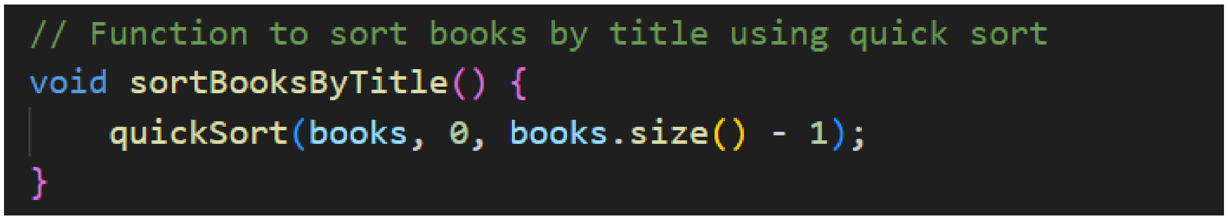
Sorting by ID with Implementation of STL Sort Algorithm



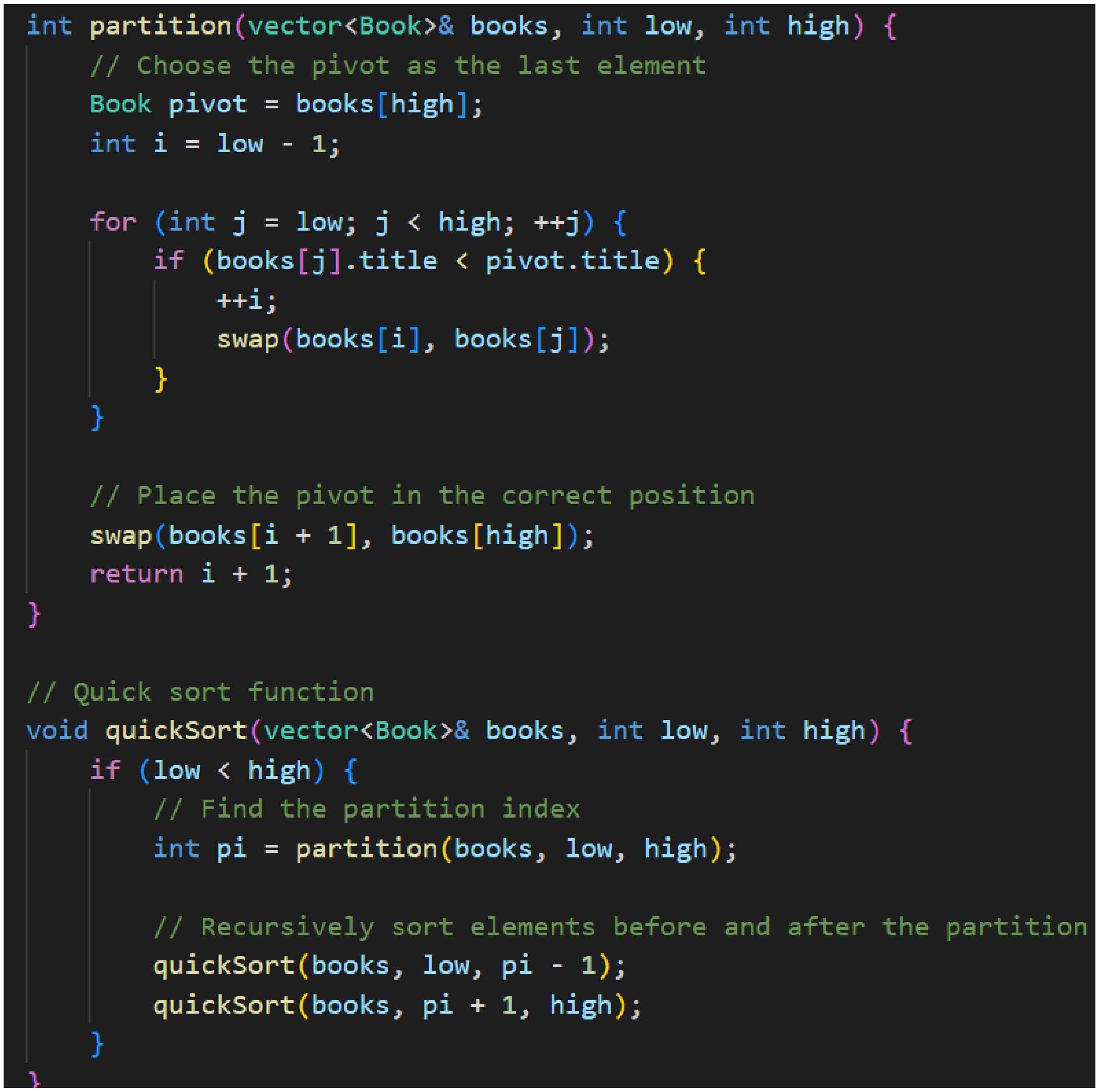
```cpp
// Sort books by id using the STL contanier sort alogorithim
void sortBooksByID() {
    sort(books.begin(), books.end(), [](const Book& a, const Book& b) {
        return a.id < b.id;
    });
}
```

Big O analysis: This algorithm consists of 2 parts, the sorting STL and the Lambda function. This is the lambda function: [](const Book& a, const Book& b) { return a.id < b.id; }. This piece of code has a time complexity of O(1). The sort function is built into the C++ code and is a hybrid of QuickSort, HeapSort, and InsertionSort. Essentially, it generally just works like a QuickSort except when it has the worst-case scenario where there is a bad pivot point. In that case, it switches to HeapSort which guarantees O(n*logn) for all cases. Also, after partitioning, once the subarray is small enough, it switches to insertion sort which further optimizes time complexity. Because of all this, the best, average, and worst case of this function is O(n*logn)

Sorting Title with Implementation of Quick Sort



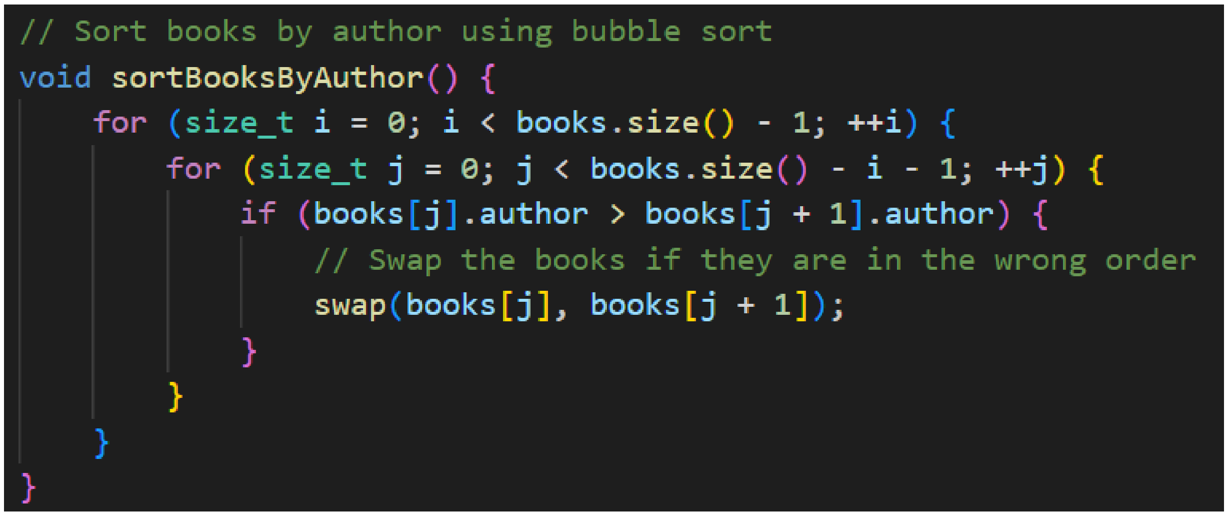
```cpp
// Function to sort books by title using quick sort
void sortBooksByTitle() {
    quickSort(books, 0, books.size() - 1);
}
```



```cpp
int partition(vector<Book>& books, int low, int high) {
    // Choose the pivot as the last element
    Book pivot = books[high];
    int i = low - 1;

    for (int j = low; j < high; ++j) {
        if (books[j].title < pivot.title) {
            ++i;
            swap(books[i], books[j]);
        }
    }

    // Place the pivot in the correct position
    swap(books[i + 1], books[high]);
    return i + 1;
}

// Quick sort function
void quickSort(vector<Book>& books, int low, int high) {
    if (low < high) {
        // Find the partition index
        int pi = partition(books, low, high);

        // Recursively sort elements before and after the partition
        quickSort(books, low, pi - 1);
        quickSort(books, pi + 1, high);
    }
}
```

Big O analysis: The sortBooksByTitle function is really just a quickSort under a different name. Quicksort is a recursive function so the time complexity really depends on the partitioning function. Partitioning is essentially just splitting the elements into 2 parts. Ideally, it's 2 equal parts. The partition function is continuously called until it subdivides all elements into singular elements. This has a time complexity of O(logn). Total time complexity is multiplying O(n) with O(logn) which gives us O(n*logn). This is the best-case scenario. The average case is that the initial pivot divides the elements only somewhat evenly. However, it is still close enough that the recursion depth, the number of partitions until each element is isolated, that the time complexity of this is still O(logn). This means that the average case is the same as the best case, that being O(n*logn). The worst-case scenario is if the pivot is the first or last element of the array. This would mean the recursion depth is going to be O(n) since you are going through each element one by one. This means the total time complexity is O(n) * O(n) which equals O(n^2) in the worst case. This is easily remedied by having the pivot located close to the center of the array of elements.

Sorting by Author with Implementation of <u>Bubble Sort</u>



```
// Sort books by author using bubble sort
void sortBooksByAuthor() {
    for (size_t i = 0; i < books.size() - 1; ++i) {
        for (size_t j = 0; j < books.size() - i - 1; ++j) {
            if (books[j].author > books[j + 1].author) {
                // Swap the books if they are in the wrong order
                swap(books[j], books[j + 1]);
            }
        }
    }
}
```

Big O analysis: This is a BubbleSort algorithm. The outer loop has a time complexity of O(n). The second/inner loop has a time complexity of O(n^2). This is because the total number of comparisons can be written as $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$. This simplifies out to $\frac{n^2}{2}$ which in time complexity is just O(n^2). Lastly, the swap operation at the end is just O(1) as it doesn't matter what n is, it'll always be the same.

<u>Conclusion:</u>

To summarize the group's findings from the completion of this lab, the group successfully created an overarching coding program that utilizes multiple concepts from our lectures in CMPE 126. We learned how to create a larger, in-depth project while implementing sorting algorithms, storage methods, and containers, and how to connect that to a real-world application. This assignment served as the perfect method for the group's individuals to practice their comprehension of the course material. It helped in our debugging skills, thought processes, and how we approach similar problems. A major challenge the group faced was, as mentioned prior, debugging the code. Although our thought process was carefully and methodically put together, we faced issues including compilation errors, syntax errors, and minor spelling mistakes which caused our program to fail. In addition, the group needed to manage our time and schedules to be able to meet and discuss the work of the project. However through all this, the group was able to complete this project in full, practicing already learned concepts, and learning how all of this will be used in our future careers.