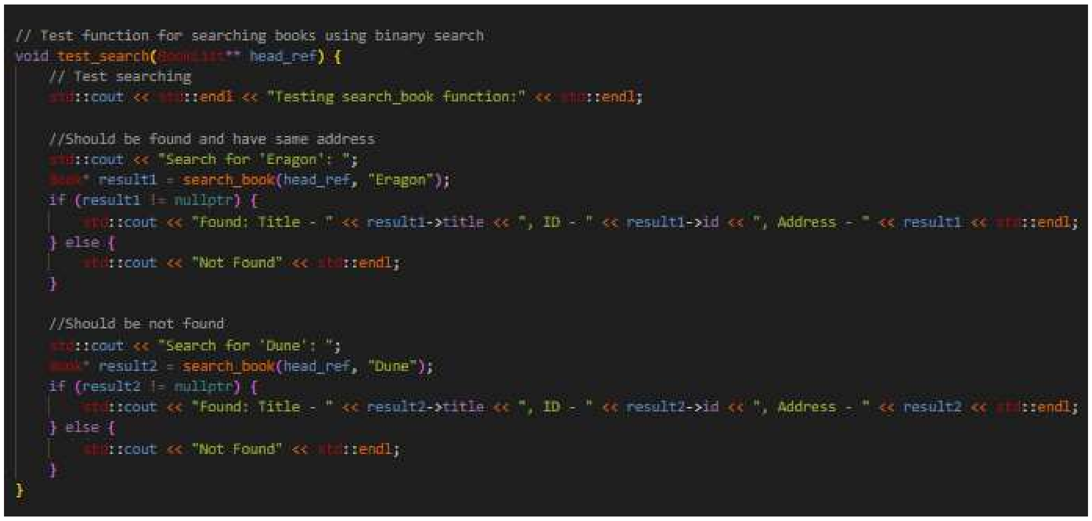
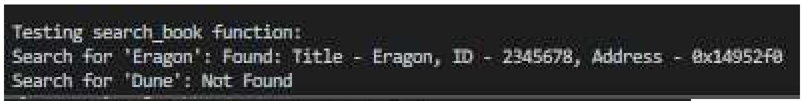
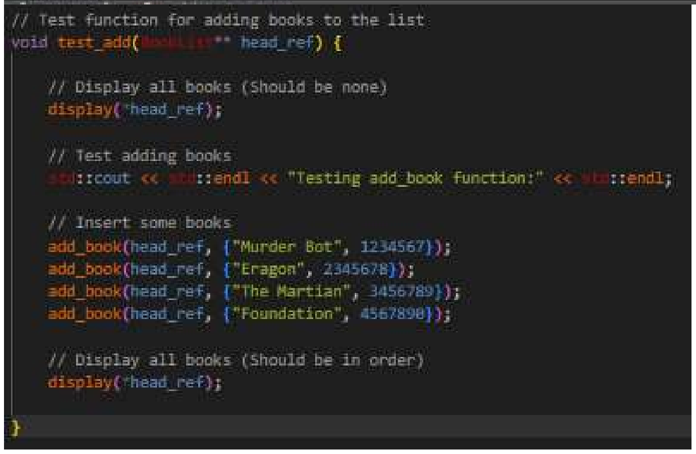
1. Follow along with the in-class exercise on this, do your best to get it working, and turn in what you come up with here!
2. Be sure to include at least one test for each function or piece of functionality that should verify that your code is working! No slacking, you should start writing some tests *before* you write your implementations (just spend a few minutes thinking about the design and then write a few tests using natural language (English is preferred for me to be able to read it)



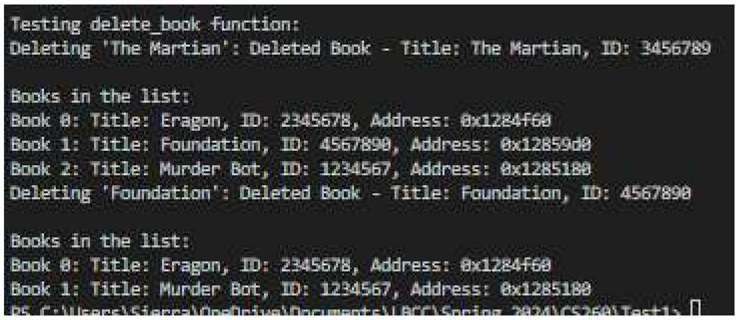
```cpp
// Test function for searching books using binary search
void test_search(BookList** head_ref) {
    // Test searching
    std::cout << std::endl << "Testing search_book function:" << std::endl;

    //Should be found and have same address
    std::cout << "Search for 'Eragon': ";
    Book* result1 = search_book(head_ref, "Eragon");
    if (result1 != nullptr) {
        std::cout << "Found: Title - " << result1->title << ", ID - " << result1->id << ", Address - " << result1 << std::endl;
    } else {
        std::cout << "Not Found" << std::endl;
    }

    //Should be not found
    std::cout << "Search for 'Dune': ";
    Book* result2 = search_book(head_ref, "Dune");
    if (result2 != nullptr) {
        std::cout << "Found: Title - " << result2->title << ", ID - " << result2->id << ", Address - " << result2 << std::endl;
    } else {
        std::cout << "Not Found" << std::endl;
    }
}
```



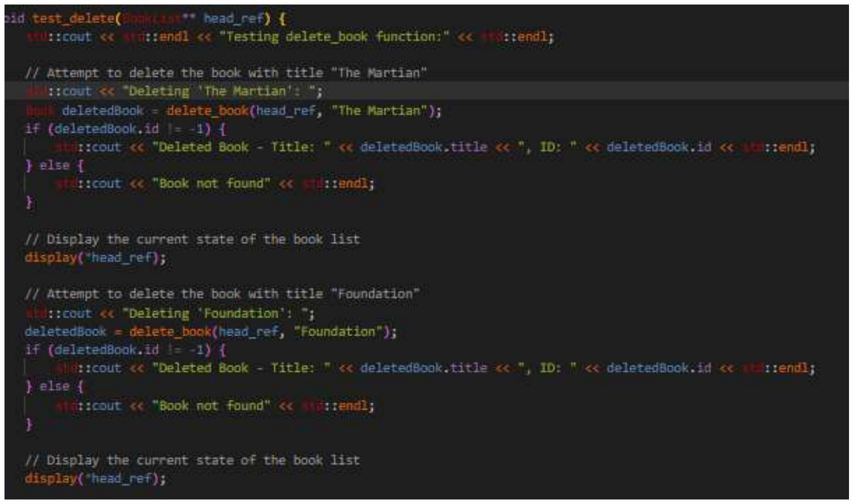
```
Testing search_book function:
Search for 'Eragon': Found: Title - Eragon, ID - 2345678, Address - 0x14952f0
Search for 'Dune': Not Found
```



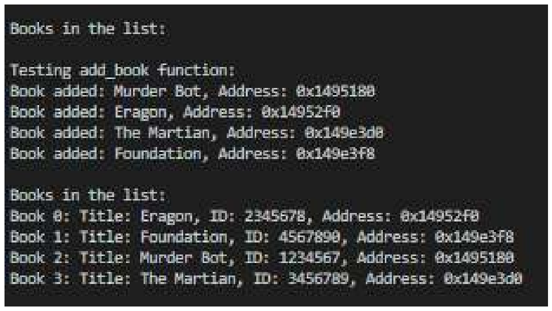
```cpp
// Test function for adding books to the list
void test_add(BookList** head_ref) {

    // Display all books (Should be none)
    display(*head_ref);

    // Test adding books
    std::cout << std::endl << "Testing add_book function:" << std::endl;

    // Insert some books
    add_book(head_ref, {"Murder Bot", 1234567});
    add_book(head_ref, {"Eragon", 2345678});
    add_book(head_ref, {"The Martian", 3456789});
    add_book(head_ref, {"Foundation", 4567890});

    // Display all books (Should be in order)
    display(*head_ref);

}
```



```
Testing delete_book function:
Deleting 'The Martian': Deleted Book - Title: The Martian, ID: 3456789

Books in the list:
Book 0: Title: Eragon, ID: 2345678, Address: 0x1284f60
Book 1: Title: Foundation, ID: 4567890, Address: 0x12859d0
Book 2: Title: Murder Bot, ID: 1234567, Address: 0x1285180
Deleting 'Foundation': Deleted Book - Title: Foundation, ID: 4567890

Books in the list:
Book 0: Title: Eragon, ID: 2345678, Address: 0x1284f60
Book 1: Title: Murder Bot, ID: 1234567, Address: 0x1285180
PS C:\Users\Sierra\OneDrive\Documents\LBCC\Spring 2024\CS260\Test1>
```



```
oid test_delete(BookList** head_ref) {
    std::cout << std::endl << "Testing delete_book function:" << std::endl;

    // Attempt to delete the book with title "The Martian"
    std::cout << "Deleting 'The Martian': ";
    Book deletedBook = delete_book(head_ref, "The Martian");
    if (deletedBook.id != -1) {
        std::cout << "Deleted Book - Title: " << deletedBook.title << ", ID: " << deletedBook.id << std::endl;
    } else {
        std::cout << "Book not found" << std::endl;
    }

    // Display the current state of the book list
    display(*head_ref);

    // Attempt to delete the book with title "Foundation"
    std::cout << "Deleting 'Foundation': ";
    deletedBook = delete_book(head_ref, "Foundation");
    if (deletedBook.id != -1) {
        std::cout << "Deleted Book - Title: " << deletedBook.title << ", ID: " << deletedBook.id << std::endl;
    } else {
        std::cout << "Book not found" << std::endl;
    }

    // Display the current state of the book list
    display(*head_ref);
```

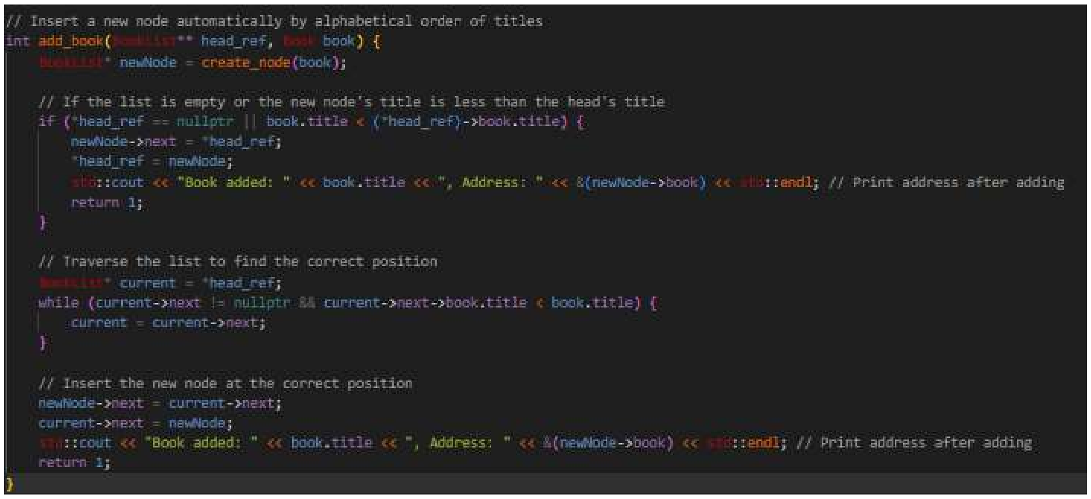


```
Books in the list:

Testing add_book function:
Book added: Murder Bot, Address: 0x1495180
Book added: Eragon, Address: 0x14952f0
Book added: The Martian, Address: 0x149e3d0
Book added: Foundation, Address: 0x149e3f8

Books in the list:
Book 0: Title: Eragon, ID: 2345678, Address: 0x14952f0
Book 1: Title: Foundation, ID: 4567890, Address: 0x149e3f8
Book 2: Title: Murder Bot, ID: 1234567, Address: 0x1495180
Book 3: Title: The Martian, ID: 3456789, Address: 0x149e3d0
```

3. Create an array-based list or a linked-list (**and a bonus for attempting both**) that:
   Source: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

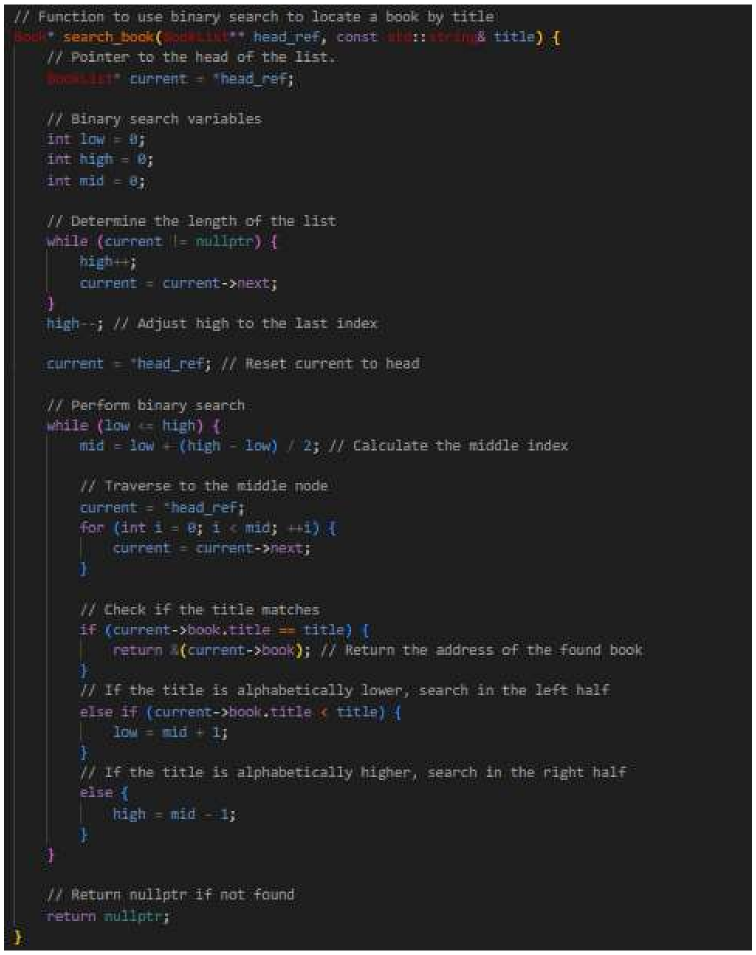
   Delete Function is very similar to one I designed in 133C. A lot of my code is similar to previous weeks. I updated my comments, because I had a misunderstanding about the previous assignment and felt I added too many.

   1. automatically inserts values in the correct position based on some order of sorting (perhaps ascending integers or lexicographical sorting of words)



```
// Insert a new node automatically by alphabetical order of titles
int add_book(BookList** head_ref, Book book) {
    BookList* newNode = create_node(book);

    // If the list is empty or the new node's title is less than the head's title
    if (*head_ref == nullptr || book.title < (*head_ref)->book.title) {
        newNode->next = *head_ref;
        *head_ref = newNode;
        std::cout << "Book added: " << book.title << ", Address: " << &(newNode->book) << std::endl; // Print address after adding
        return 1;
    }

    // Traverse the list to find the correct position
    BookList* current = *head_ref;
    while (current->next != nullptr && current->next->book.title < book.title) {
        current = current->next;
    }

    // Insert the new node at the correct position
    newNode->next = current->next;
    current->next = newNode;
    std::cout << "Book added: " << book.title << ", Address: " << &(newNode->book) << std::endl; // Print address after adding
    return 1;
}
```

2. efficiently searches for elements (likely binary search for the array list, but what about the linked-list?)



```cpp
// Function to use binary search to locate a book by title
Book* search_book(BookList** head_ref, const std::string& title) {
    // Pointer to the head of the list.
    BookList* current = *head_ref;

    // Binary search variables
    int low = 0;
    int high = 0;
    int mid = 0;

    // Determine the length of the list
    while (current != nullptr) {
        high++;
        current = current->next;
    }
    high--; // Adjust high to the last index

    current = *head_ref; // Reset current to head

    // Perform binary search
    while (low <= high) {
        mid = low + (high - low) / 2; // Calculate the middle index

        // Traverse to the middle node
        current = *head_ref;
        for (int i = 0; i < mid; ++i) {
            current = current->next;
        }

        // Check if the title matches
        if (current->book.title == title) {
            return &(current->book); // Return the address of the found book
        }
        // If the title is alphabetically lower, search in the left half
        else if (current->book.title < title) {
            low = mid + 1;
        }
        // If the title is alphabetically higher, search in the right half
        else {
            high = mid - 1;
        }
    }

    // Return nullptr if not found
    return nullptr;
}
```

4. Make a chart to compare the algorithmic complexity (use Big-O notation) of your insert, remove, and search algorithms you used for your structures

Operation	Best Case	Worst Case	Expected Case	Comments
Insert	O(log n)	O(log n)	O(log n)	**Best Case:** Inserting into a perfectly balanced tree, requires traversing log(n) nodes. **Worst Case:** Inserting into a balanced tree is still O(log n) due to rebalancing. **Expected Case:** Balanced trees maintain O(log n) complexity through balancing operations.
Remove	O(log n)	O(log n)	O(log n)	**Best Case:** Removing a leaf or node with one child in a balanced tree. **Worst Case:** Requires O(log n) time due to rebalancing. **Expected Case:** Balanced trees ensure O(log n) complexity through rebalancing operations.
Search	O(1)	O(log n)	O(log n)	**Best Case:** Target is at the root. **Worst Case:** Requires traversing the height of the tree, O(log n). **Expected Case:** Balanced trees maintain O(log n) search time due to the nature of binary search.

5. Once you have implemented and tested your code, add to the README file what line(s) of code or inputs and outputs show your work meeting each of the above requirements (or better, include a small screen snip of where it meets the requirement!).

   DONE!