Design (Inspiration taken heavily from my previous work)

main.cpp:

Include book.h and test.h

1. int main()
    - test_add()
    - test_peek_a_book()
    - test_delete()

test.h

Declares the functions:

1. test_add
2. test_peek_a_book
3. test_delete

test.cpp:

Implement the test functions:

1. test_add:
    - Create an empty BookList pointer.
    - Add three books to the list using add_book, each with a different position.
    - Print the books added.
2. test_delete:
    - Create an empty BookList pointer.
    - Add books to the list.
    - Delete book in position 1 from list.
3. test_peek_a_book:
    - Create an empty BookList pointer.
    - Add books to the list.
    - Peek at a book in a certain position, printing its details.

book.h

Contains:

1. The struct for Book
    - Title (pointer to a string (title))
    - Book id (integer)
2. The struct for BookList
    - Allows for linked list
3. Declares the functions
    - Create_node
    - Add_book
    - Delete_book
    - Peek_a_book

book.cpp

1. Create_node(BookList **head_ref, Book book)
    - Creates new node for linked list of books
2. Add_book(BookList **head_ref)
    - Inserts a new book into the list at the specified position

    Delete_book(BookList **head_ref)

    - Removes a book from the list at the specified position and returns it.
3. peek_a_book(BookList **head_ref)
    - Returns the book at the specified position without removing it.

1. Create a linked-list that allows:
    1. an add function that takes a value and inserts it into a given position into the list (example: myList.add(someValue, somePosition))

```
// Function to insert a new book at a given position in the linked list.
int add_book(BookList **head_ref, Book book, int position) {
    // Create a new node with the given book information.
    BookList* newNode = create_node(book);
    // If inserting at the beginning (position 0) or the list is empty.
    if (position == 0 || *head_ref == nullptr) {
        // Point the new node to the current head.
        newNode->next = *head_ref;
        // Update the head to the new node.
        *head_ref = newNode;
        // Return 1 to indicate success.
        return 1;
    }
    // Start from the head of the list.
    BookList* current = *head_ref;
    // Traverse the list to find the node just before the insertion point.
    for (int i = 0; i < position - 1 && current->next != nullptr; ++i) {
        // Move to the next node.
        current = current->next;
    }
    // Insert the new node at the desired position.
    newNode->next = current->next;
    current->next = newNode;
    // Return 1 to indicate success.
    return 1;
}
```

2. a remove function that takes a position and removes the value stored at that position of the list and returns it
(example: myList.remove(somePosition))

```
// Function to remove the book at a given position in the list and return it.
Book delete_book(BookList **head_ref, int position) {
    // If the list is empty, return a default book.
    if (*head_ref == nullptr) {
        return {"", -1}; // No book found, return a default empty book.
    }
    // Pointer to the head of the list.
    BookList* temp = *head_ref;
    // If removing the head (position 0).
    if (position == 0) {
        // Update the head to the next node.
        *head_ref = temp->next;
        // Store the book from the deleted node.
        Book deletedBook = temp->book;
        // Free the memory of the deleted node.
        delete temp;
        // Return the deleted book.
        return deletedBook;
    }
    // Traverse the list to find the node just before the removal point.
    for (int i = 0; temp != nullptr && i < position - 1; ++i) {
        temp = temp->next;
    }
    // If the desired position is out of bounds.
    if (temp == nullptr || temp->next == nullptr) {
        return {"", -1}; // No book found, return a default empty book.
    }
    // Pointer to the node to be deleted.
    BookList* next = temp->next->next;
    Book deletedBook = temp->next->book;
    // Free the memory of the node to be deleted.
    delete temp->next;

    // Update the next pointer of the previous node.
    temp->next = next;

    // Return the deleted book.
    return deletedBook;
}
```

3. a get function that takes a position and returns that value without removing it (example: multistage(somePosition))

```
// Function to return the book at a given position without removing it.
Book peek_a_book(BookList **head_ref, int position) {
    // Pointer to the head of the list.
    BookList* current = *head_ref;
    // Traverse the list to the desired position.
    for (int i = 0; current != nullptr && i < position; ++i) {
        current = current->next;
    }
    // If the position is out of bounds.
    if (current == nullptr) {
        return {"", -1}; // No book found, return a default empty book.
    }
    // Return the book at the desired position.
    return current->book;
}
```

2. Be sure to include at least one test function for each piece of functionality that should verify that your code is working! This should be at least one test per behavior, likely more. You can make these tests in a source file with a main where your tests are either directly in the main or inside their own standalone functions (please do not neglect the importance of testing!)

```
void print_books(BookList* head) {
    BookList* current = head;
    int position = 0;
    while (current != nullptr) {
        cout << "Position: " << position << ", Title: " << current->book.title << ",
ID: " << current->book.id << endl;
        current = current->next;
        ++position;
    }
}
void test_add() {
    BookList* head = nullptr;
    // Test adding books
    Book book1 = {"Murder Bot", 1234567};
    Book book2 = {"Eragon", 2345678};
    Book book3 = {"The Martian", 3456789};
    add_book(&head, book1, 0);  // Adding at position 0
    add_book(&head, book2, 1);  // Adding at position 1
    add_book(&head, book3, 1);  // Adding at position 1 (between Murder Bot and
Eragon)
    // Print the books in the order they were added with their positions
    cout << "Books after adding:" << endl;
    print_books(head);
}
```

```
Testing add_book function:

Books after adding:
Position: 0, Title: Murder Bot, ID: 1234567
Position: 1, Title: The Martian, ID: 3456789
Position: 2, Title: Eragon, ID: 2345678
```

```
void test_peek_a_book() {
    BookList* head = nullptr;

    // Adding books
    Book book1 = {"Murder Bot", 1234567};
    Book book2 = {"Eragon", 2345678};
    Book book3 = {"The Martian", 3456789};

    add_book(&head, book1, 0);
    add_book(&head, book2, 1);
    add_book(&head, book3, 1);

    // Get and print
    cout << "Getting book at position 1:" << endl;
    Book peekedBook = peek_a_book(&head, 1);
    cout << "Got Book - Title: " << peekedBook.title << ", ID: " << peekedBook.id
<< endl;
}
```

```
Testing peek_a_book function:

Getting book at position 1:
Got Book - Title: The Martian, ID: 3456789
```

```
void test_delete() {
    BookList* head = nullptr;
    // Adding books
    Book book1 = {"Murder Bot", 1234567};
    Book book2 = {"Eragon", 2345678};
    Book book3 = {"The Martian", 3456789};
    add_book(&head, book1, 0);
    add_book(&head, book2, 1);
    add_book(&head, book3, 1);
    // Remove and print
    cout << "Removing book at position 1:" << endl;
    Book deletedBook = delete_book(&head, 1);
```

```
    cout << "Removed Book - Title: " << deletedBook.title << ", ID: " <<
deletedBook.id << endl;
    // Print remaining books with their positions
    cout << "Books after removing:" << endl;
    print_books(head);
}
```

```
Testing delete_book function:

Removing book at position 1:
Removed Book - Title: The Martian, ID: 3456789
Books after removing:
Position: 0, Title: Murder Bot, ID: 1234567
Position: 1, Title: Eragon, ID: 2345678
```

3. 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

4. (**Note**: we will cover the analysis of some of this in class next week, then we will have you analyze the next ones!)
   Attempt to analyze the complexity of your implementation with line-by-line analysis,

I think if I understand this correctly:

1. **create_node:**
   - Memory allocation: O(1)
   - Copying title: O(n) **where n = length of title**
   - Copying ID: O(1)
   - Initializing next pointer: O(1)
   - Total complexity: O(n)
2. **add_book:**
   - Memory allocation in create_node: O(1)
   - If condition check: O(1)
   - Assignment operations: O(1)
   - Traversing to insertion point: O(n) **where n = position in list**
   - Insertion: O(1)
   - Total complexity: Best case O(1) where the book is added to the first position, worst case O(n), where n is a non-first position in the list.
3. **delete_book:**
   - If condition check: O(1)
   - Pointer assignment: O(1)

- Traversing to deletion point: O(n) **where n = position in list**
- Deletion: O(1)
- Total complexity: Best case O(1) where the book is deleted from the first position, worst case O(n), where n is a non-first position in the list.

4. **peek_a_book:**
    - Traversing to desired position: O(n) **where n = position in list**
    - Total complexity: Best case O(1) where the book is in the first position, worst case O(n), where n is a non-first position in the list.