1. Create a design, **before** you start coding, that shows how your binary tree functions and what attributes it keeps track of to function (yes, you can add to this design once you start coding, but please get some design down to start with and make note of when you add new design features based on your implementation work🙂)

Design Overview:

General Plan:

The Binary Search Tree (BST) will be designed to store books, with each node containing book information and pointers to its left and right child nodes. The BST will support insertion, deletion, and search operations, along with various traversal methods. The program will be broken down into 5 sections, main.cpp, book.cpp, book.h, test.cpp, and test.h.

main.cpp:

int main(){

BookList* root = nullptr;

call tests

book.h:

Book: Represents a book with attributes title and id.

BookList: Node structure containing a Book and pointers to its left and right child nodes.

Headers for all the functions used in book.cpp

book.cpp:

BookList* create_node(Book book)

BookList* add_book(BookList* root, Book book)

BookList* delete_book(BookList* root, const std::string& title)

Book* search_book(BookList* root, const std::string& title)

Tree traversal functions (start with in-order)

test.h

Headers for all the tests

test.cpp

test_add

test_delete

test_search

test_traversals

2. Create some tests (at least one per function), **before** you start coding, that you want your Binary Search Tree (BST) to pass as evidence that it would be working correctly if it passed the tests.

Tests are very similar to my previous assignments; however they will print the different traversals.

Add:

```
void test_add(BookNode** root) {
    std::cout << "Before adding books:" << std::endl;
    in_order_traversal(*root);

    std::cout << std::endl << "Testing add_book function:" << std::endl;

    // Add some books
    *root = add_book(*root, {"Murder Bot", 1234567});
    *root = add_book(*root, {"Eragon", 2345678});
    *root = add_book(*root, {"The Martian", 3456789});
    *root = add_book(*root, {"Foundation", 4567890});

    std::cout << "After adding books:" << std::endl;
    test_traversals(*root);
}
```

```
Testing add_book function:
After adding books:

Testing in_order_traversal function:
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: The Martian, ID: 3456789, Address: 0x10fc5d0

Testing pre_order_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
Title: The Martian, ID: 3456789, Address: 0x10fc5d0

Testing post_order_traversal function:
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: The Martian, ID: 3456789, Address: 0x10fc5d0
Title: Murder Bot, ID: 1234567, Address: 0x10fc780

Testing breadth_first_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: The Martian, ID: 3456789, Address: 0x10fc5d0
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
```

Search:

```
// Test function for searching books in the BST
void test_search(BookNode** root) {
    std::cout << std::endl << "Testing search_book function:" << std::endl;

    // Search for 'Eragon'
    std::cout << "Search for 'Eragon': ";
    Book* result1 = search_book(*root, "Eragon");
    if (result1 != nullptr) {
        std::cout << "Found: Title - " << result1->title << ", ID - " << result1->id << "
    } else {
        std::cout << "Not Found" << std::endl;
    }

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

```
Testing search_book function:
Search for 'Eragon': Found: Title - Eragon, ID - 2345678, Address - 0x10fc930
Search for 'Dune': Not Found
```

Delete:

```
void test_delete(BookNode** root) {
    std::cout << std::endl << "Testing delete_book function:" << std::endl;

    // Attempt to delete the book with title "The Martian"
    std::cout << "Deleting 'The Martian': ";
    *root = delete_book(*root, "The Martian");

    std::cout << std::endl << "After deleting 'The Martian':" << std::endl;
    test_traversals(*root);

    // Attempt to delete the book with title "Foundation"
    std::cout << std::endl << "Deleting 'Foundation': ";
    *root = delete_book(*root, "Foundation");

    std::cout << std::endl << "After deleting 'Foundation':" << std::endl;
    test_traversals(*root);
}
```

```
Testing delete_book function:
Deleting 'The Martian':
After deleting 'The Martian':

Testing in_order_traversal function:
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
Title: Murder Bot, ID: 1234567, Address: 0x10fc780

Testing pre_order_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Foundation, ID: 4567890, Address: 0x10fc8a0

Testing post_order_traversal function:
Title: Foundation, ID: 4567890, Address: 0x10fc8a0
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Murder Bot, ID: 1234567, Address: 0x10fc780

Testing breadth_first_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Foundation, ID: 4567890, Address: 0x10fc8a0

Deleting 'Foundation':
After deleting 'Foundation':

Testing in_order_traversal function:
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Murder Bot, ID: 1234567, Address: 0x10fc780

Testing pre_order_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930

Testing post_order_traversal function:
Title: Eragon, ID: 2345678, Address: 0x10fc930
Title: Murder Bot, ID: 1234567, Address: 0x10fc780

Testing breadth_first_traversal function:
Title: Murder Bot, ID: 1234567, Address: 0x10fc780
Title: Eragon, ID: 2345678, Address: 0x10fc930
```

Traversals:

```
void test_traversals(BookNode* root) {
    std::cout << std::endl << "Testing in_order_traversal function:" << std::endl;
    in_order_traversal(root);

    std::cout << std::endl << "Testing pre_order_traversal function:" << std::endl;
    pre_order_traversal(root);

    std::cout << std::endl << "Testing post_order_traversal function:" << std::endl;
    post_order_traversal(root);

    std::cout << std::endl << "Testing breadth_first_traversal function:" << std::endl;
    breadth_first_traversal(root);
}
```

3. Implement a binary search tree that includes:

    1. Nodes to store values,

```
// Define the node struct
struct BookNode {
    Book book;
    BookNode* left;
    BookNode* right;
};
```

```
// Define the struct for a book
struct Book {
    std::string title; // Use std::string for the title
    int id;
};
```

    2. an add function that adds a new value in the appropriate location based on our ordering rules, (I likely used less than or equal to going to the left and greater than values going to the right)

```
// Add a book alphabetically
BookNode* add_book(BookNode* root, Book book) {
    if (root == nullptr) {
        return create_node(book);
    }
    if (book.title < root->book.title) {
        root->left = add_book(root->left, book);
    } else {
        root->right = add_book(root->right, book);
    }
    return root;
}
```

    3. a remove function that finds and removes a value and then picks an appropriate replacement node, (successor is a term often used for this)

        source: https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

```
// Delete a book
BookNode* delete_book(BookNode* root, const std::string& title) {
    if (root == nullptr)
        return root;

    // If the title to be deleted is smaller than the root's title,
    // then it lies in the left subtree
    if (title < root->book.title) {
        root->left = delete_book(root->left, title);
        return root;
    }
    // If the title to be deleted is greater than the root's title,
    // then it lies in the right subtree
    else if (title > root->book.title) {
        root->right = delete_book(root->right, title);
        return root;
    }

    // If title is same as root's title, then this is the node to be deleted
    // Node with only one child or no child
    if (root->left == nullptr) {
        BookNode* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == nullptr) {
        BookNode* temp = root->left;
        delete root;
        return temp;
    }
```

Continued….

```
    // Node with two children: Get the inorder successor (smallest
    // in the right subtree)
    BookNode* succParent = root;
    BookNode* succ = root->right;
    while (succ->left != nullptr) {
        succParent = succ;
        succ = succ->left;
    }

    // Copy the inorder successor's content to this node
    root->book = succ->book;

    // Delete the inorder successor
    if (succParent->left == succ)
        succParent->left = succ->right;
    else
        succParent->right = succ->right;

    delete succ;
    return root;
}
```

4. we have at least one tree traversal function (I recommend starting with an in-order traversal!) **Bonus** if you implement the three common traversals (pre-order, post-order, in-order) **More Bonus** if you also include a breadth-first traversal (sometimes called a level-order search)

   Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

```
// In-order traversal
void in_order_traversal(BookNode* root) {
    if (root != nullptr) {
        in_order_traversal(root->left);
        std::cout << "Title: " << root->book.title << ", ID: " << root->book.id << ", Adc
        in_order_traversal(root->right);
    }
}

// Pre-order traversal
void pre_order_traversal(BookNode* root) {
    if (root != nullptr) {
        std::cout << "Title: " << root->book.title << ", ID: " << root->book.id << ", Adc
        pre_order_traversal(root->left);
        pre_order_traversal(root->right);
    }
}

// Post-order traversal
void post_order_traversal(BookNode* root) {
    if (root != nullptr) {
        post_order_traversal(root->left);
        post_order_traversal(root->right);
        std::cout << "Title: " << root->book.title << ", ID: " << root->book.id << ", Adc
    }
}

// Breadth-first traversal
void breadth_first_traversal(BookNode* root) {
    if (root == nullptr)
        return;

    std::queue<BookNode*> q;
    q.push(root);

    while (!q.empty()) {
        BookNode* current = q.front();
        q.pop();
        std::cout << "Title: " << current->book.title << ", ID: " << current->book.id <<
        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

4. Analyze and compare the complexity of insert and search as compared to a binary tree without any order in its nodes (what is the run-time of an unordered tree...?).

   Source: https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/ and chatgbt to check my descriptions.

Big O Notation	Analysis
Ordered Insert	
O(1)	Adding to an empty tree is constant time because the new node can directly become the root.
O(logn)	In an ordered tree, adding is logarithmic time since it involves traversing from the root to the leaf, choosing the appropriate side based on the value.
O(n)	If the tree is unbalanced, adding can degrade to linear time, where each insertion may require traversing through the entire height of the tree.
Ordered Search	
O(1)	Finding the root is constant time, as it's the first node to be checked.
O(logn)	In an ordered tree, searching is logarithmic time because it involves traversing from the root to a leaf, choosing the appropriate side based on the value.
O(n)	If the tree is unbalanced, searching can become linear time, where each search may require traversing through the entire height of the tree.
Unordered Insert	
O(1)	In an unordered tree, inserting is also constant time since there's no specific order to follow.
Unordered Search	
O(n)	Searching in an unordered tree can degrade to linear time in the worst-case scenario, as every node may need to be checked to find the target value.

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!!