# Implementation

# [ binary\_search\_tree.hpp ]

 **Constructor and Destructor**: The constructor initializes an empty tree with root set to nullptr and count set to zero. The destructor calls the clear method to deallocate memory and reset the tree.

 **Insertion**: The insert method adds a new item to the tree. If the current node is nullptr, it creates a new binary\_node with the given item and increments the count. If the item’s key is less than or greater than the current node's key, it recursively calls itself on the left or right child, respectively. If the keys are equal, it updates the current node's data.

 **Removal**: The remove method deletes an item from the tree based on its key. It locates the node containing the key, and if the node has no children, it simply removes it. If it has one child, it replaces the node with that child. If it has two children, it finds the minimum node in the right subtree, copies its value to the node being deleted, and removes the minimum node.

 **Search Operations**: The contains method checks if a key exists in the tree by traversing left or right based on the comparison with the current node's key. The find method returns a pointer to the data associated with the given key, if found.

 **Finding Minimum and Maximum**: The find\_min and find\_max methods retrieve the minimum and maximum values in the tree. They throw an exception if the tree is empty. The corresponding helper methods navigate through the left or right children to locate the minimum or maximum node.

 **Clear Method**: The clear method recursively deallocates all nodes in the tree, resetting the count to zero.

 **Size and Empty Checks**: The size method returns the total number of elements in the tree, while the empty method checks if the tree is devoid of elements.

 **Printing Methods**: The tree can be printed in three different orders: in-order, pre-order, and post-order. Each printing method recursively traverses the tree, outputting the data from the nodes in the specified order.

 **Operator Overloading**: The insertion operator << is overloaded to allow for easy printing of the binary search tree's contents in in-order format.

# [ hash\_table.hpp ]

 **Constructor and Destructor**: The constructor initializes the hash table with a specified number of cells, setting the capacity and resizing the table vector accordingly. The destructor calls the clear method to remove all elements from the table.

 **Insertion**: The insert method adds a new item to the hash table. It first computes the index using the hash function with the item's key. The item is then added to the list at that index, and the count is incremented.

 **Removal**: The remove method deletes an item from the table based on its key. It calculates the index using the hash function and then calls list\_remove to remove the item from the corresponding list. If the item is successfully removed, the count is decremented.

 **Retrieval**: The get method retrieves an item based on its key. It computes the index using the hash function and calls list\_find to search for the item in the corresponding list.

 **Contains Check**: The contains method checks if a specific key exists in the hash table. It calculates the index using the hash function and calls list\_contains to verify the presence of the key in the list.

 **Clear Method**: The clear method empties all lists in the table and resets the count to zero.

 **Empty Check**: The empty method checks if the hash table is empty by returning true if the count of items is zero.

 **Size Method**: The size method returns the number of items currently stored in the hash table.

 **Print Method**: The print method outputs the contents of the hash table to the specified output stream. It displays each index and the items stored at that index.

 **Operator Overloading**: The insertion operator << is overloaded to facilitate easy printing of the hash table's contents.

# [ book\_genre\_stat.c ]

 **Constructors**: The class has two constructors. The default constructor initializes the genre as an empty string, the count as zero, and the total rating as zero. The second constructor takes a string parameter to initialize the genre while setting the count and total rating to zero.

 **Getters**: The get\_key method returns the genre as a constant reference. The get\_count method returns the current count of books in the genre.

 **Count Management**: The increment\_count method increases the count by one, while the decrement\_count method decreases the count by one, ensuring that it does not go below zero.

 **Rating Management**: The add\_rating method adds a rating to the total rating, and the subtract\_rating method removes a rating from the total rating.

 **Average Rating**: The average\_rating method computes the average rating for the genre. If the count is zero, it returns 0.0 to avoid division by zero.

 **Operator Overloading**: The code overloads the output operator << to allow easy printing of book\_genre\_stats objects in the format "(genre, count, average\_rating)". It also overloads the equality operator == to compare two book\_genre\_stats objects based on their genre keys. Additionally, it overloads the less than operator < and the greater than operator > to facilitate comparison based on genre keys.

# [ book.cpp ]

 **Constructors**: The class provides two constructors. The default constructor initializes the isbn, title, and genre as empty strings and the rating as zero. The parameterized constructor allows the initialization of these attributes with specific values.

 **Getters**: The class includes several getter methods:

get\_key returns the ISBN of the book as a constant reference.

get\_title returns the title of the book.

get\_genre returns the genre of the book.

get\_rating returns the book's rating.

 **Operator Overloading**:

The equality operator == is overloaded to allow comparison of two book objects based on their ISBNs.

The less than operator < and greater than operator > are also overloaded to facilitate comparisons based on the ISBNs, which allows for sorting and ordering of book objects.

 **Output Operator Overloading**: The output operator << is overloaded to enable easy printing of book objects. It formats the output as "(ISBN, Title, Genre, Rating)", providing a clear representation of the book's details.

# [ inventory.cpp ]

 **Constructors and Destructor**: The default constructor initializes the inventory with an empty binary search tree (bst) and a hash table (ht) with a capacity of 101. The destructor calls the clear method to clean up resources.

 **Add Book**: The add\_book method adds a new book to the inventory. It inserts the book into the BST and retrieves the genre. If the genre does not already exist in the hash table, it creates a new book\_genre\_stats entry for that genre. It then updates the genre statistics by incrementing the count and adding the book's rating.

 **Remove Book**: The remove\_book method removes a book from the inventory based on its ISBN. It first finds the book in the BST. If found, it retrieves the genre and updates the genre statistics by decrementing the count and subtracting the book's rating before removing the book from the BST.

 **Check Book Existence**: The book\_exists method checks if a book with a given ISBN exists in the inventory by querying the BST.

 **Get Book**: The get\_book method retrieves a pointer to a book with the specified ISBN from the BST.

 **Display Books**: The display\_books method outputs the contents of the BST, showing all the books in the inventory.

 **Clear Inventory**: The clear method removes all books and resets the statistics in both the BST and the hash table.

 **Genre Count**: The genre\_count method returns the number of books in a specified genre by accessing the genre statistics in the hash table.

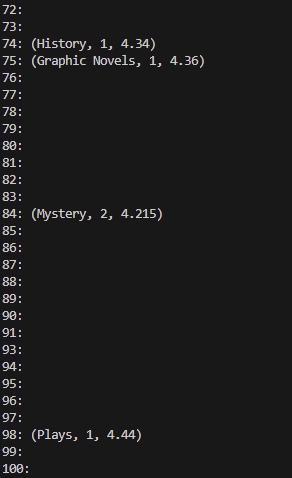
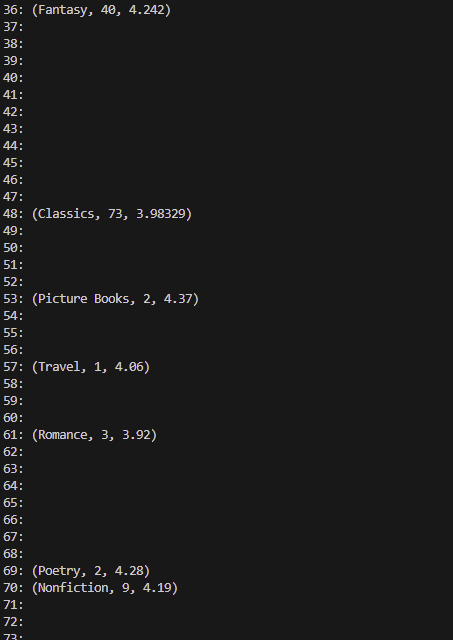
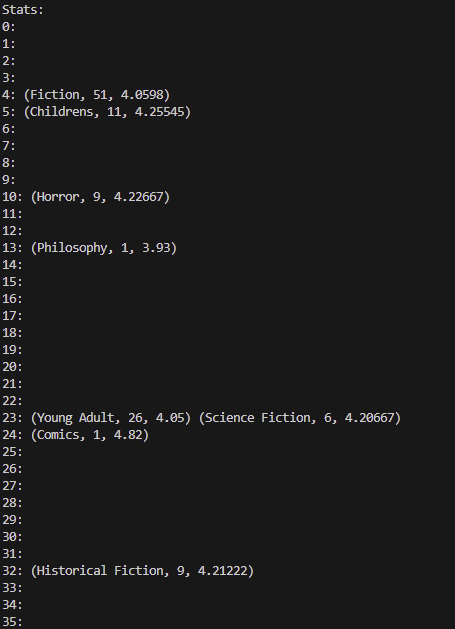
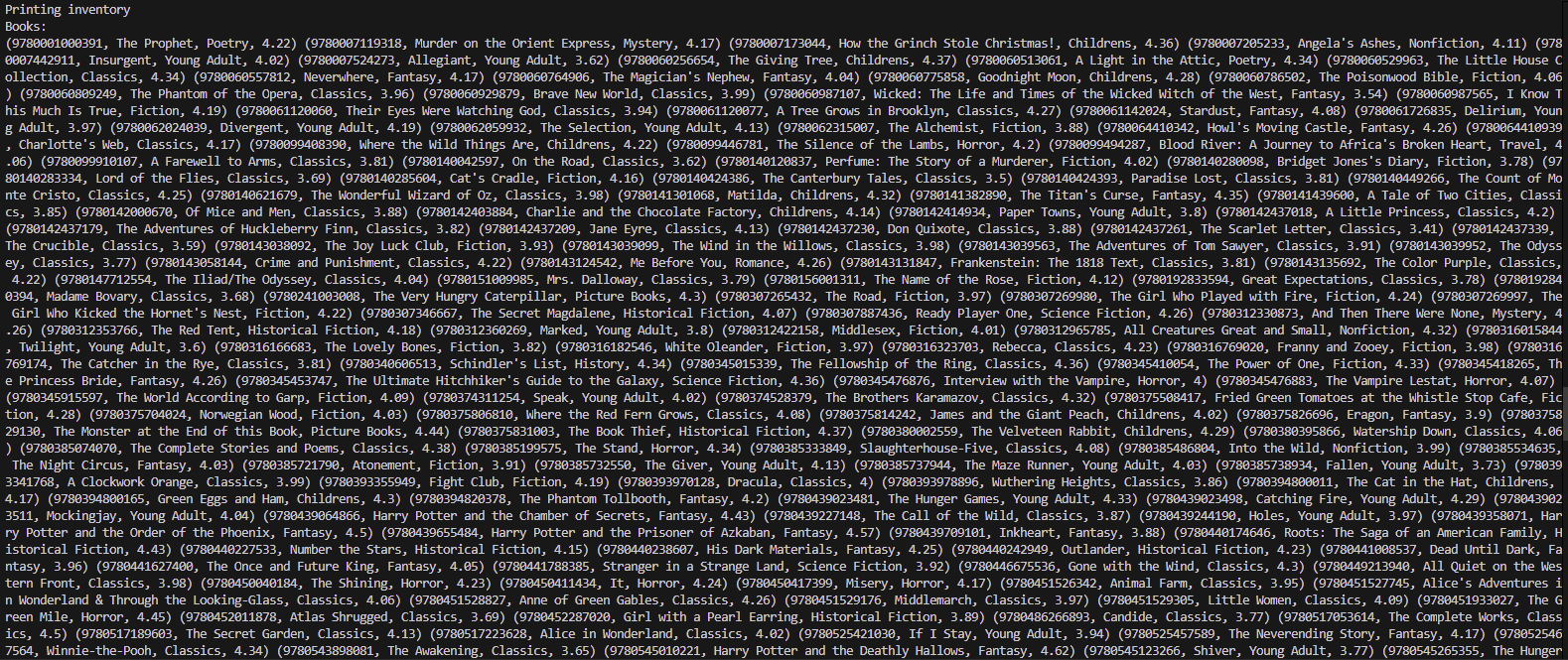
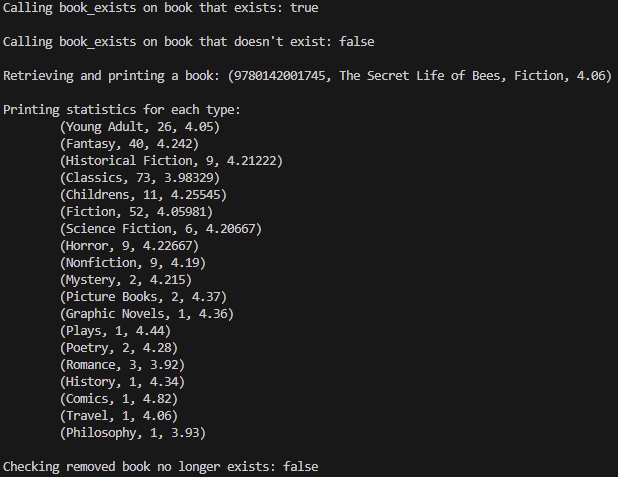
 **Genre Average Rating**: The genre\_average\_rating method retrieves the average rating for a specified genre from the hash table.

 **Get Genre Statistics**: The get\_stats method returns the book\_genre\_stats object associated with a specific genre from the hash table.

 **Print Method**: The print method formats the output of the inventory, displaying both the books and their genre statistics.

 **Operator Overloading**: The output operator << is overloaded to facilitate easy printing of the inventory's contents.

# Result



# Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Test Description | Expected Result | Test Result |
| 1 | Add a book to the inventory | Book should be added successfully, count in BST should increase by 1 | Passed |
| 2 | Add multiple books of the same genre | Genre count should reflect the total number of books added of that genre | Passed |
| 3 | Retrieve a book by ISBN | Should return the correct book object corresponding to the given ISBN | Passed |
| 4 | Check if a book exists in the inventory | Should return true for existing ISBN, false for a non-existing ISBN | Passed |
| 5 | Remove a book from the inventory | Book should be removed successfully, count in BST should decrease by 1, genre count should also reflect this | Passed |
| 6 | |  | | --- | |  |  |  | | --- | | Calculate the average rating of a genre | | Should return the correct average based on the added ratings of books in that genre | Passed |
| 7 | Clear the inventory | Both BST and hash table should be empty, counts should reset to 0 | Passed |
| 8 | Print inventory | Should output the correct formatted list of books and their statistics | Passed |

# Reflection

Through this assignment, I gained valuable experience in designing and implementing a data management system using various data structures. Working with the binary search tree (BST) for storing books and a hash table for managing genre statistics allowed me to deepen my understanding of their respective strengths and weaknesses. I realized the importance of ensuring efficient data retrieval and organization, which reinforced my skills in choosing appropriate data structures based on specific use cases.

In terms of improvements, I recognized the need for enhanced error checking and exception handling throughout the implementation. For instance, when adding books, it would be beneficial to verify that the ISBN is unique and follows a specific format, as this could prevent potential issues in data integrity. Additionally, I considered exploring other data structures, such as balanced trees (like AVL or Red-Black trees), which could improve search and insert efficiency in cases with a large volume of data. Overall, this assignment helped me appreciate the nuances of data structure selection and the importance of robust error handling in software development.