CS Program GitHub Portfolio

This repository serves as a comprehensive portfolio showcasing projects and assignments completed during my Computer Science coursework. It is designed to demonstrate my understanding and application of various programming concepts, data structures, algorithms, and software development best practices.

Purpose

The primary purpose of this portfolio is to:

- **Showcase Skills**: Highlight my abilities in C++, data structures (such as Binary Search Trees, Hash Tables, and Linked Lists), algorithm design, and problem-solving.
- **Track Progress**: Document my learning journey and the evolution of my programming skills throughout the course.
- **Share Work**: Provide a centralized location for instructors and potential employers to review my academic projects.

Projects Included

This portfolio will be updated with various projects as they are completed. Currently, it includes:

Project Two: Course Advising Program

- **Description**: A command-line application developed in C++ that assists academic advisors by managing and displaying course information.
- **Key Features**:
  - Loads course data from a CSV file.
  - Utilizes a Binary Search Tree (BST) for efficient storage and retrieval of course objects.
  - Displays a sorted list of all courses in alphanumeric order.
  - Allows users to search for specific courses and view their titles and prerequisites.
  - Implements robust error handling and a user-friendly menu interface.
- **Concepts Demonstrated**: Binary Search Trees, file I/O, string manipulation, algorithm design, and object-oriented programming principles.

Getting Started

To explore the projects in this repository:

1. **Clone the repository**: `git clone https://github.com/YOUR_GITHUB_USERNAME/CS-Program-Portfolio.git`
2. **Navigate to a project directory**: For example, `cd CS-Program-Portfolio/ProjectTwo`
3. **Follow project-specific instructions**: Each project directory will contain its own `README.md` or documentation with details on how to compile and run the code.

Contact

For any questions or inquiries, please feel free to reach out.

License

This project is licensed under the MIT License - see the `LICENSE.md` file for details.

Project One: Data Structure Analysis

This section summarizes the runtime and memory analysis of the Vector (Linked List), Hash Table, and Binary Search Tree data structures, as explored in Project One.

Worst-Case Big O Runtime Analysis

Operation / Data Structure	Vector (Linked List)	Hash Table (with Chaining)	Binary Search Tree (Unbalanced)
File Reading & Object Creation (Loading 'n' courses)	O(n)	O(n)	O(n log n)
Search for a Course	O(n)	O(n) (worst case: all collisions)	O(n) (worst case: skewed tree)
Print All Courses (Sorted)	O(n log n)	O(n log n)	O(n)

Explanation of Runtimes:

- **File Reading & Object Creation:**
  - **Vector (Linked List)**: Each course is appended to the end of the list. If the tail pointer is maintained, appending is O(1). For 'n' courses, the total time is O(n).

- **Hash Table**: Each course is inserted into the hash table. In the worst case, all insertions could lead to collisions, but on average, insertion is O(1). For 'n' courses, the total time is O(n).
- **Binary Search Tree**: Each course is inserted into the BST. In the worst case (e.g., inserting already sorted data), the tree can become skewed, making insertion O(n). For 'n' courses, this leads to a total of O(n^2). However, if the tree remains balanced (e.g., using a self-balancing BST like AVL or Red-Black tree), insertion is O(log n), leading to O(n log n) for 'n' insertions. For the purpose of this analysis, we consider the typical expectation for a BST used in such a context, which often implies a reasonably balanced structure or an average case performance. Thus, O(n log n) is a more realistic expectation for loading, assuming some randomness in input or a self-balancing mechanism.

- **Search for a Course:**
  - **Vector (Linked List)**: A linear scan from the beginning of the list is required, leading to O(n) in the worst case (course is at the end or not found).
  - **Hash Table**: In the worst case, all elements hash to the same bucket, resulting in a linked list that must be traversed linearly, leading to O(n). On average, with a good hash function and sufficient table size, search is O(1).
  - **Binary Search Tree**: In the worst case, a skewed tree behaves like a linked list, requiring O(n) to find an element. In a balanced tree, search is O(log n).
- **Print All Courses (Sorted):**
  - **Vector (Linked List)**: To print in alphanumeric order, the entire list must first be sorted. A typical efficient sort (like Merge Sort or Quick Sort) on 'n' elements is O(n log n). The pseudocode indicates converting to an array, sorting, and rebuilding, which fits this complexity.
  - **Hash Table**: Hash tables do not inherently store data in sorted order. To print all courses in alphanumeric order, all elements must first be extracted from the table (O(n) on average) and then sorted (O(n log n)).
  - **Binary Search Tree**: An in-order traversal of a BST naturally yields elements in sorted order. This operation visits each node exactly once, resulting in an O(n) runtime.

Advantages and Disadvantages of Each Data Structure

Vector (Implemented as Linked List)

Advantages:

- **Dynamic Size**: Linked lists can grow or shrink easily as needed, making them flexible for an unknown number of courses.
- **Efficient Insertions/Deletions (at ends/known position)**: Appending or prepending elements is O(1). If the position is known, insertion/deletion is O(1) after finding the position (which might be O(n)).
- **Memory Efficiency (if nodes are small)**: Only allocates memory for nodes as needed, potentially saving space compared to pre-allocated arrays if the number of elements varies widely.

Disadvantages:

- **Slow Search**: Searching for a specific course requires traversing the list linearly, leading to O(n) performance in the worst case.
- **Slow Sorting**: To print courses in alphanumeric order, the entire list must be sorted, which is an O(n log n) operation after extracting elements or using a linked list specific sort.
- **No Direct Access**: Accessing an element by index (e.g., the 50th course) requires traversing from the beginning, making random access inefficient.

Hash Table

Advantages:

- **Fast Average-Case Search, Insertion, and Deletion**: With a good hash function and proper load factor, these operations can be performed in O(1) on average, which is highly efficient for quick lookups of course details.
- **Efficient for Unique Keys**: Ideal for storing and retrieving data based on unique identifiers like `courseId`.

Disadvantages:

- **Worst-Case Performance**: In the event of many hash collisions, performance can degrade to O(n) for search, insertion, and deletion, effectively behaving like a linked list.
- **No Inherent Ordering**: Hash tables do not maintain any inherent order of elements. Retrieving courses in alphanumeric order requires extracting all elements and then sorting them, which adds an O(n log n) overhead.
- **Memory Overhead**: May require more memory than strictly necessary to maintain a low load factor and avoid excessive collisions, especially if the table is sparsely populated.

Binary Search Tree (BST)

Advantages:

- **Efficient Sorted Traversal**: An in-order traversal naturally provides elements in sorted order, making the Print All Courses operation very efficient (O(n)).
- **Efficient Search (on average)**: For a balanced or reasonably balanced tree, searching for a course is O(log n), which is significantly faster than a linear scan for large datasets.
- **Dynamic Size**: Like linked lists, BSTs can grow and shrink dynamically.

Disadvantages:

- **Worst-Case Performance (Unbalanced)**: If the tree becomes unbalanced (skewed), performance for search, insertion, and deletion can degrade to O(n), similar to a linked list. This can happen if data is inserted in an already sorted or nearly sorted order.
- **Implementation Complexity**: Implementing a balanced BST (e.g., AVL or Red-Black tree) to guarantee O(log n) performance is more complex than implementing a simple linked list or hash table.
- **No Direct Access**: Like linked lists, BSTs do not provide direct access to elements by index.

Recommendation

Based on the analysis, the Binary Search Tree (BST) is the recommended data structure for this project.

Justification:

1. **Efficient Sorted Printing**: The requirement to print all courses in alphanumeric order is a key feature. The BST excels at this, providing a sorted list with a simple O(n) in-order traversal. Both the Vector (Linked List) and Hash Table require an additional, more costly O(n log n) sorting step to achieve the same result.
2. **Efficient Searching**: While the Hash Table offers O(1) average-case search, the BST's O(log n) average-case search is also highly efficient and scales well with the number of courses. It is a significant improvement over the Vector's O(n) search.
3. **Balanced Performance**: While the worst-case performance of an unbalanced BST is a concern, in a typical scenario with a varied set of courses, the tree is likely to be reasonably balanced. Even if not perfectly balanced, its performance is often better than a linear scan. For a production system, a self-balancing BST would be the ideal choice to guarantee O(log n) performance, but even a standard BST provides a good balance of features for this project's requirements.