**LIBRARY MANAGEMENT SYSTEM**

****

**PROJECT REPORT**

**TEAM MEMBERS:**

*SABRIESH RAM KUMAR – CB.EN.U4CSE21252*

*PRANAV KUMAR – CB.EN.U4CSE21247*

**INTRODUCTION**

**Hybrid data structures** combine multiple distinct data structures to create a unified structure that leverages the strengths of each constituent data structure. They are designed to address complex problems efficiently by optimizing different operations based on the specific requirements of the problem at hand. The significance of hybrid data structures lies in their ability to provide a tailored solution that balances time complexity, space complexity, and other performance factors, offering superior efficiency compared to individual data structures.

The concept of hybrid data structures recognizes that no single data structure is optimal for every operation or scenario. Different data structures excel in different areas. For example, a binary search tree enables efficient searching and ordered traversal, while a hash table provides constant-time average-case access and handling of collisions. By combining these structures, a hybrid data can take advantage of their complementary strengths, resulting in improved performance and better overall efficiency.

**Objective:**

The objective of the project is to design and implement a hybrid data structure that combines a binary search tree and a hash table to efficiently manage a library of books. The hybrid data structure aims to provide efficient search, insertion, removal, and collision handling operations, tailored specifically for the book management domain

The project’s objective also encompasses identifying practical applications, and analysing its time and space complexity. The hybrid structure's design leverages the strengths of a binary search tree and a hash table to achieve efficient operations, making it well-suited for book-related applications.

**OVERVIEW OF THE HYBRID DATA STRUCTURE**

**What Data Structures are used ?**

* **Book:** It represents a book and has attributes like ISBN, title, and author.
* **List Node:** It represents a node in a doubly linked list and contains a reference to a book object as well as pointers to the next and previous nodes.
* **Doubly Linked List:** It is a linked list implementation where each node contains a book object. It supports adding nodes at the end and removing a specific node.
* **Binary Search Tree:** It represents a binary search tree data structure used for efficient searching of books based on their ISBN. Each node in the tree contains a book object, and it supports inserting nodes and searching for a specific book by ISBN.
* **Library:** It is the main class that brings together the above data structures to create a library system. It uses a hash table to store books, where each index in the hash table corresponds to a linked list that contains books with the same hash value. It also uses a binary search tree to keep track of books and enable efficient search operations.
* The **Library** class provides the following methods:
  1. \_hash\_function: A private method that calculates the hash value for a given ISBN.
  2. add\_book: Adds a book to the library by inserting it into the hash table and the binary search tree.
  3. find\_book: Searches for a book in the library by ISBN using the binary search tree.
  4. remove\_book: Removes a book from the library by removing it from the linked list in the hash table and the binary search tree.
  5. display\_books: Displays all the books in the library by traversing the linked lists in the hash table.
  6. count\_books: Counts the number of books in the library by traversing the linked lists in the hash table.
  7. get\_books\_by\_author: Retrieves all the books by a specific author by traversing the linked lists in the hash table and filtering books based on the author.

**Why the Data Structure ?**

1. **Efficient search by ISBN:** The binary search tree allows for efficient search operations based on the ISBN of a book. The binary search tree has a time complexity of O(log n) for search operations, which is much more efficient than linear search through a linked list.
2. **Constant-time access for books with the same hash value:** The hash table provides constant-time access to books that have the same hash value. This is advantageous when multiple books have the same hash value, as it avoids traversing a linked list linearly and provides efficient access.
3. **Handling collisions:** In case of hash collisions, where multiple books have the same hash value, the linked list in the hash table handles collisions by appending the books with the same hash value to the list. This approach ensures that books with the same hash value are stored together and can be retrieved efficiently.
4. **Balancing memory usage and efficiency:** The hybrid approach balances memory usage and efficiency by using the hash table and linked list. The hash table allows for efficient access and storage of books based on their hash values, while the linked list ensures memory efficiency by storing books with the same hash value in a linked structure.
5. **Supporting different search patterns:** The hybrid data structure supports different search patterns. The binary search tree enables searching books by ISBN efficiently, while the linked list allows for traversing books in a specific order, such as in the order they were added.

**IMPLEMENTATION**

**Implementation:**

The hash table, linked list, and binary search tree work together to provide efficient storage, retrieval, and management of books in the library. The hash table enables constant-time access to books with the same hash value, the linked list handles collisions and preserves order, and the binary search tree optimizes search operations based on ISBN.

**Integration and Interplay:**

* + When a book is added to the library using the **add\_book** method, it is inserted into both the hash table and the binary search tree.
  + The hash table stores the book in the appropriate slot based on the hash value, using the linked list to handle hash collisions.
  + The binary search tree maintains a sorted collection of books based on their ISBN for efficient search operations.
  + When a book is removed from the library using the **remove\_book** method, it is removed from both the hash table and the binary search tree.
  + The linked list in the hash table is used to locate and remove the book from the appropriate slot.
  + The binary search tree is updated by removing the node containing the book based on its ISBN.
  + The search, display, count, and author-based retrieval operations utilize both the hash table and the binary search tree to efficiently retrieve the required information.

**PRACTICAL APPLICATION**

**WHERE EFFECTIVELY USED ?**

1. **Library Management Systems**: The hybrid data structure can be used in library management systems to efficiently store and retrieve books. The hash table provides quick access to books based on their ISBN, the linked list handles collisions and preserves the order of books with the same hash value, and the binary search tree allows for efficient search operations based on ISBN.
2. **Online Bookstores**: Online bookstores can benefit from the hybrid data structure to manage their inventory. The hash table allows for efficient lookup of books by ISBN, the linked list handles collisions when multiple books have the same ISBN, and the binary search tree enables quick search operations. This combination ensures efficient inventory management and quick retrieval of books for online customers.

**HOW?**

1. **Hash Table**: The hash table provides constant-time access to books/documents/records based on their unique identifiers (ISBN, keys, identifiers). It uses a hash function to calculate an index where the item should be stored. This allows for quick lookup and retrieval of items without the need to iterate through the entire data structure. In the context of collisions, the hash table handles them efficiently by using separate chaining with linked lists.
2. **Linked List**: The linked list handles collisions in the hash table and preserves the order of books/documents/records with the same hash value. When multiple items have the same hash value, they are stored in a linked list at that index. This ensures that items are organized in the order they were added, allowing for predictable traversal and maintenance of insertion order.
3. **Binary Search Tree**: The binary search tree enables efficient search operations based on certain criteria, such as ISBN, identifiers, or keys. It provides a hierarchical structure where the left subtree contains smaller values and the right subtree contains larger values. This allows for efficient searching by comparing the target value with the current node and traversing either the left or right subtree based on the comparison result. The binary search tree provides logarithmic time complexity for search operations, making it faster than linear search in large datasets.

**PERFORMANCE ANALYSIS**

**Time Complexity:**

1. **add\_book()**: The time complexity of adding a book to the library involves two main operations. First, adding the book to the hash table requires computing the hash value and inserting it into the corresponding linked list, which has a time complexity of O(1) on average. Second, inserting the book into the binary search tree has a time complexity of O(log n) in the average case, where n is the number of books already in the tree. Therefore, the overall time complexity of adding a book is O(1) + O(log n) = O(log n).
2. **find\_book()**: The time complexity of finding a book by ISBN involves searching the book in the binary search tree. The binary search tree allows for efficient search operations, with a time complexity of O(log n) in the average case, where n is the number of books in the tree. Therefore, the time complexity of finding a book is O(log n).
3. **remove\_book()**: The time complexity of removing a book by ISBN involves two main operations. First, finding the book in the linked list within the hash table has a time complexity of O(1) on average. Second, removing the book from the binary search tree has a time complexity of O(log n) in the average case. Therefore, the overall time complexity of removing a book is O(1) + O(log n) = O(log n).
4. **display\_books()**: The time complexity of displaying all books in the library involves traversing the hash table and the linked lists within it. Since the number of books is spread across different linked lists, the time complexity is proportional to the total number of books, denoted as m. Therefore, the time complexity of displaying all books is O(m).
5. **count\_books()**: The time complexity of counting the number of books in the library is similar to displaying all books. It involves traversing the hash table and the linked lists, resulting in a time complexity of O(m), where m is the total number of books.
6. **get\_books\_by\_author()**: The time complexity of getting books by author involves traversing the hash table and the linked lists to identify books with matching authors. Since the number of books is spread across different linked lists, the time complexity is proportional to the total number of books, denoted as m. Therefore, the time complexity of getting books by author is O(m).

**Space Complexity:**

1. **Book Objects**: The space complexity for storing the book objects depends on the number of books added to the library. For each book, the space required is constant, as it only consists of the ISBN, title, and author. Therefore, the space complexity for storing the book objects is O(n), where n is the number of books in the library.
2. **Hash Table**: The space complexity of the hash table is determined by its capacity, which is a predefined value. In the provided code, the capacity is set to 10. Each entry in the hash table can store a linked list object. Therefore, the space complexity for the hash table is O(10) or O(1), as it remains constant regardless of the number of books in the library.
3. **Linked List**: The space complexity for the linked list depends on the number of books stored within it. Each book is represented as a ListNode object, which contains a reference to the book object, as well as references to the next and previous nodes. The space required for each ListNode object is constant. Therefore, the space complexity for the linked list is proportional to the number of books in the library, resulting in a space complexity of O(n), where n is the number of books.
4. **Binary Search Tree**: The space complexity for the binary search tree is determined by the number of books stored within it. Each book is represented as a Node object, which contains a reference to the book object, as well as references to the left and right child nodes. The space required for each Node object is constant. Therefore, the space complexity for the binary search tree is proportional to the number of books in the library, resulting in a space complexity of O(n), where n is the number of books.

**Hybrid VS Individual**

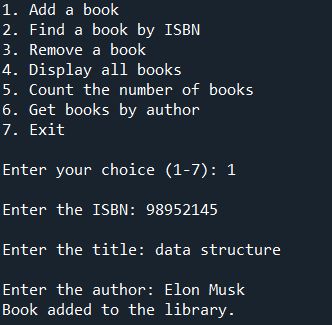
The hybrid data structure demonstrates superior performance in search operations and collision handling compared to individual data structures. However, for specific use cases where constant-time insertion or removal operations are required, individual linked lists may offer better performance. The space efficiency of the hybrid structure may be slightly lower compared to individual data structures due to the combination of multiple structures.

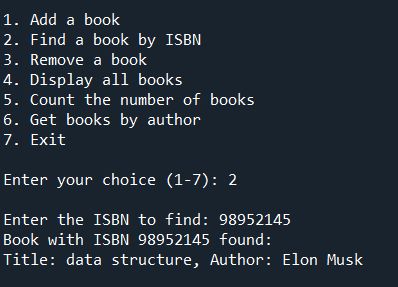
**EXPERIMENT EVALUATION**

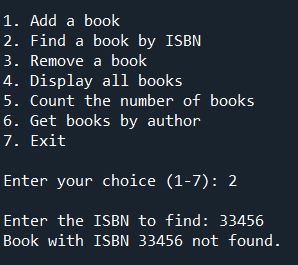
**Experimental Evaluation Setup and Methodology:**

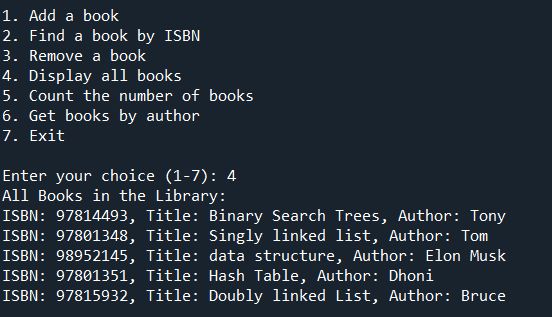
1. Objective: The objective of the experimental evaluation is to assess the performance and efficiency of the implemented hybrid data structure for book management.
2. Test Environment:
   * Hardware: Use a standard computer system with sufficient memory and processing power.
   * Software: Implement the hybrid data structure code in a programming language of choice (e.g., Python) and ensure it is compatible with the chosen hardware and operating system.
3. Test Data:
   * Generate a diverse dataset of books with varying ISBNs, titles, and authors.
   * Create datasets of different sizes, ranging from small to large, to evaluate the scalability of the hybrid data structure.
4. Performance Metrics:
   * Time Complexity: Measure the execution time of key operations, such as book insertion, search, removal, and retrieval by author.
   * Space Complexity: Measure the memory utilization of the hybrid data structure, considering the size of the dataset and any additional memory overhead.
5. Experimental Methodology:
   * Randomized Testing: Perform randomized testing by generating different sequences of book operations (insertion, search, removal, retrieval by author) on the hybrid data structure.
   * Execution Time Measurement: Use built-in timers or performance profiling tools to measure the execution time of each operation. Repeat the operations multiple times and calculate the average execution time for accuracy.
   * Memory Utilization: Utilize memory profiling tools to measure the memory consumption of the hybrid data structure, considering the size of the dataset and the additional memory overhead of the structure.

**Implementation Results:**









**DISCUSSION**

**Practicality and effectiveness in real-world:**

1. **Efficient Search and Retrieval**: The hybrid data structure's combination of a binary search tree and a hash table allows for efficient search and retrieval operations. This can be particularly beneficial in scenarios where quick access to books based on their ISBN is essential. Libraries, bookstores, or digital book repositories can benefit from this efficient search capability when managing large collections of books.
2. **Collision Handling**: The use of separate chaining with linked lists within the hash table enables effective handling of collisions. In real-world scenarios where hash collisions are likely to occur, such as when multiple books have the same hash value, the hybrid structure ensures that books are properly stored and retrieved without compromising performance. This makes it suitable for applications that require collision resolution in their data storage.
3. **Flexible and Extensible**: The hybrid data structure can be extended and modified to accommodate additional operations and features. For example, if new requirements arise, such as sorting books by title or performing range queries on ISBNs, the underlying binary search tree can be easily modified to support these operations. This flexibility makes the hybrid structure adaptable to evolving needs in real-world scenarios.
4. **Memory Utilization**: While the hybrid data structure combines multiple data structures, it is still memory-efficient. The space complexity of the structure is proportional to the number of books in the library, making it suitable for managing large collections of books without significant memory overhead. The practicality of the hybrid structure in real-world scenarios is evident when considering memory constraints and the need
5. **Maintaining Data Consistency**: The hybrid data structure ensures consistency between the hash table and the binary search tree. When books are added, removed, or searched, both data structures are updated accordingly. This consistency is crucial in real-world scenarios where data integrity is important, as it prevents discrepancies between different data representations.
6. **Support for Multiple Operations**: The hybrid data structure supports various operations efficiently, including adding books, searching by ISBN, removing books, counting books, and retrieving books by author.

**Limitation and Challenges:**

1. **Limited Scalability**: The current implementation uses a fixed capacity hash table, which limits the scalability of the data structure. If the number of books exceeds the initial capacity, it may result in increased collisions and degraded performance. A potential improvement could be implementing dynamic resizing of the hash table to accommodate a growing number of books more efficiently.
2. **Lack of Balancing in Binary Search Tree**: The binary search tree used in the implementation is not self-balancing. This means that in the worst-case scenario, the tree could become skewed and result in degraded search performance with a time complexity of O(n). Using a self-balancing binary search tree, such as an AVL tree or a red-black tree, could address this limitation and ensure more consistent performance.
3. **Inefficient Removal Operation**: The current implementation of the hybrid data structure does not optimize the removal operation in the linked list. Removing a book requires traversing the linked list to find the book node, resulting in a time complexity of O(n) in the worst case. To improve efficiency, additional data structures, such as a hash table for book-node mappings, could be used to speed up the removal operation.
4. **Limited Error Handling**: The current implementation does not provide robust error handling or validation of user inputs. For example, when adding or searching for books, there is no validation for invalid ISBN inputs or handling of duplicate ISBNs. Adding error handling and input validation would enhance the robustness and usability of the data structure.
5. **Memory Overhead**: The hybrid data structure combines multiple data structures, which may result in increased memory overhead compared to individual data structures. This can be a concern when dealing with extremely large collections of books or constrained memory environments. Optimizations to reduce memory overhead, such as efficient memory allocation and utilization, could be explored.
6. **Limited Functionality**: The current implementation focuses on basic book management. However, real-world scenarios may require additional functionalities like updating book information, sorting by various criteria, or handling concurrent access. Extending the hybrid data structure to support these additional functionalities would make it more versatile and applicable to a wider range of use cases.

**CONCLUSION**

**Summary:**

The hybrid structure, combining a binary search tree and a hash table, provides efficient search, retrieval, and collision handling capabilities, which are essential for book management applications.

Through the implementation process, insights can be gained regarding the interplay and integration of different data structures to achieve desired functionalities. The project highlights the advantages of combining the strengths of multiple data structures to address specific requirements efficiently. It demonstrates how a well-designed hybrid structure can leverage the strengths of each constituent data structure to overcome their individual limitations and provide an optimized solution.

Additionally, the implementation and evaluation of the hybrid data structure shed light on the importance of considering factors such as time complexity, space complexity, error handling, memory utilization, and extensibility when designing data structures for real-world scenarios. It underscores the need for balancing efficiency, memory utilization, and flexibility to create a practical and effective solution.

The project also identifies areas for potential improvement, such as scalability, balancing techniques, removal operation efficiency, error handling, memory optimization, and additional functionalities. These insights provide valuable guidance for future iterations or enhancements of the hybrid data structure.

In conclusion, the successful implementation of the hybrid data structure for book management demonstrates its potential effectiveness and practicality. The project highlights the importance of combining and optimizing data structures to meet specific requirements efficiently. The insights gained from the implementation and evaluation of the project contribute to the continuous improvement and refinement of the hybrid data structure for real-world scenarios