1. Implementation Process and Challenges

The implementation of the **Library Management System** involved creating three primary data structures: **Hash Table**, **Binary Search Tree (BST)**, and **Linked List**. Each structure was chosen based on its ability to handle specific tasks such as searching, storing books in sorted order, and managing checked-out books dynamically. The following sections describe the process, challenges, and solutions for each data structure.

1.1 Hash Table

Objective:

- Store and retrieve books efficiently using **ISBN**, **title**, or **author**.
- Ensure constant time complexity **O(1)** for search, insertion, and deletion operations.

Challenges:

- **Handling multiple keys**: Since the library system allows searching by ISBN, title, and author, a separate hash table was required for each key. This introduced the complexity of ensuring that all keys are synced and reflect the current state of the system.

Solutions:

- Three separate hash tables were implemented to handle ISBN, title, and author. The title and author hash tables reference the ISBN, which is the primary key for the book's details.

Critical Code Snippet:

```
def add_book(self, isbn, title, author):

    # Insert book details into hash tables

    self.books_by_isbn[isbn] = (title, author)

    self.books_by_title[title] = isbn

    if author not in self.books_by_author:

        self.books_by_author[author] = []

    self.books_by_author[author].append(isbn)
```

- This function ensures that each book is added to all relevant hash tables.
- Syncing the author's hash table required handling multiple books by the same author, which was done using a **list of ISBNs** for each author.

1.2 Binary Search Tree (BST)

Objective:

- Maintain an **ordered collection of books** by title to allow quick searches and retrieval in lexicographical order.

Challenges:

- **Tree imbalance**: A typical challenge with BSTs is that the tree can become unbalanced if the data is inserted in a skewed manner (e.g., if books are inserted in ascending or descending order of titles). This can degrade the time complexity from **O(log n)** to **O(n)**.

Solutions:

- While a **self-balancing tree** such as an AVL tree or red-black tree would prevent imbalance, for the proof of concept, a simple BST was implemented. This provides a functional system for ordered data retrieval but leaves room for future optimization through balancing.

Critical Code Snippet:

```
def insert(self, key, value):
    if self.root is None:
        self.root = TreeNode(key, value)
    else:
        self._insert(self.root, key, value)


def _insert(self, node, key, value):
    if key < node.key:
        if node.left is None:
            node.left = TreeNode(key, value)
        else:
            self._insert(node.left, key, value)
    elif key > node.key:
        if node.right is None:
            node.right = TreeNode(key, value)
        else:
            self._insert(node.right, key, value)
```

- The recursive function inserts books in lexicographical order based on their title. This forms the basis for ordered searches and potential future sorting features.

1.3 Linked List

Objective:

- Track the **checked-out books** for each user dynamically, allowing efficient insertion and deletion of books as they are checked out or returned.

Challenges:

- **Search inefficiency**: Linked lists have a linear search time **O(n)**, which is less efficient compared to other structures like hash tables. While linked lists are efficient for dynamic insertions and deletions, they are suboptimal for frequent lookups.

Solutions:

- For the current proof of concept, linked lists were chosen to manage checked-out books due to their simplicity and ease of implementation. However, future optimization could involve a hybrid data structure such as a hash table or an array for faster lookups.
- An **edge case** where a book being returned might not be found in the list was handled by traversing the list and checking each node.

Critical Code Snippet:

```
def return_book(self, book_title):
    prev = None
    curr = self.head
    while curr is not None:
        if curr.book_title == book_title:
            if prev:
                prev.next = curr.next
            else:
                self.head = curr.next
            return True
        prev = curr
        curr = curr.next
```

```
return False  # Book not found
```

- This function efficiently handles the deletion of a node (book) from the list, ensuring that books can be returned seamlessly.

---

2. Key Design Changes and Considerations

During the implementation, the following design changes were made:

1. **Multiple Hash Tables**:
    - Initially, a single hash table was planned to manage all book-related data. However, upon realizing the need for multiple types of searches (by ISBN, title, and author), I implemented separate hash tables for each attribute. This allows for faster searches depending on the user's query type.
2. **Tree Balance for BST**:
    - Although the current proof of concept uses a simple **unbalanced** BST, the potential for imbalance in larger datasets was identified. In future phases, this can be addressed by implementing a **self-balancing tree** such as a red-black tree or AVL tree to maintain logarithmic search time even as the dataset grows.
3. **Linked List Performance**:
    - While the linked list efficiently handles dynamic check-ins and check-outs, it lacks optimal search capabilities. For the current limited use case, the simplicity of the linked list suffices, but future optimizations could replace it with a data structure that supports faster search times, such as a hash table.

3. Next Steps for Full Implementation

For future phases, the following steps are planned to complete the full implementation of the Library Management System:

1. **Enhancing the Hash Table**:
    - Add handling for **hash collisions** using either chaining or open addressing techniques.
    - Implement **resizing** functionality to maintain optimal performance as the dataset grows.
2. **Balancing the Binary Search Tree**:
    - Replace the current simple BST with a **self-balancing tree** like an AVL tree or red-black tree. This will ensure the tree remains balanced and maintains efficient operations for larger datasets.
3. **Optimizing the Linked List**:

    - Replace the linked list for tracking checked-out books with a more efficient data structure, such as a **hash map** or an **array** that supports faster lookups.
4. **Real-Time Book Availability**:
    - Implement **real-time availability checks** for books. This could involve integrating the checked-out book list with the hash table to ensure books are marked as unavailable when checked out and available when returned.
5. **Concurrency Handling**:
    - Introduce mechanisms to handle **concurrency** in a real-world environment where multiple users might be accessing and modifying the system simultaneously. This could include adding **locking mechanisms** or using thread-safe data structures.
6. **Testing for Larger Datasets**:
    - Extend the test cases to include **stress testing** with larger datasets. This will allow us to identify any performance bottlenecks and further optimize the data structures for scalability.

Conclusion

The proof of concept successfully demonstrated the core functionality of the Library Management System using hash tables, binary search trees, and linked lists. Despite some challenges, such as handling multiple search keys and the potential for tree imbalance, the current implementation provides a solid foundation for future enhancements. The next phase will focus on optimizing the data structures for larger datasets, implementing concurrency control, and refining the system's real-time responsiveness.