ATENEO DE MANILA
UNIVERSITY

Final Project: Efficient Library System “Libapp” for Del Pilar Academy

MSYS 30

Data Structures and Algorithms

Group 8

Chloe Juliana G. Cruz

Maristel T. Lluch

Chloe Sachiel P. Manook

December 2, 2024

Table of Contents

Project Background

Context and Problem Statement

The Del Pilar Academy (DPA) Library is a school library located at Gen. E. Topacio Street in Imus, Cavite that houses a collection of books and other reading materials for the use of DPA's junior and senior high school students.

The library currently operates with an outdated manual system for tasks such as book circulation, requiring library assistants to dedicate much of their time to helping students borrow and return books using card catalogues. While the system is functional, its lack of user-friendliness and limited staff training result in underutilization. Additionally, the absence of a centralized record-keeping system makes it challenging for the staff to maintain accurate borrower data for reporting.

To address these challenges, the group has proposed automating the circulation process. This solution aims to streamline library operations, enhance data management, and support the school's accreditation efforts by providing reliable, data-driven reports.

Need

The DPA library has an existing computer-based system called My RDA (Resource Description & Access), introduced in 2015, which includes a primary module for Circulation. The Circulation Module is designed for recording book borrowing and returns. However, My RDA remains underutilized due to a lack of familiarity among the current library staff. Those initially trained to operate the system have left the school, and no succession plan was implemented. The system remains installed but is non-functional, with no updates or maintenance since its initial deployment. Furthermore, My RDA lacks essential features such as automated book borrowing and returning, book searching, and overdue fine calculation, which are critical to the library's needs.

Due to these shortcomings and missing functionalities, the library continues to rely heavily on manual processes. According to the client, this reliance is inefficient and time-consuming for the limited staff of three. Manual operations are prone to errors, such as misplaced records, illegible handwriting, and data entry mistakes, leading to issues like overlooked return dates and uncollected overdue fines, which undermine the reliability of the library's records and result in lost revenue. Additionally, the lack of proper record-keeping increases the risk of losing track of borrowed books or misplacing physical records like book cards, further complicating operations.

Given that the root issue lies in My RDA's lack of user-friendliness and essential features, the group has proposed implementing a more robust and efficient system. A new system will not only streamline the library's processes but also improve user experience and encourage both staff and students to make better use of the library's resources. This upgrade

aims to enhance operational standards, ensure accuracy, and keep the library adaptable to future technological advancements, ultimately supporting its continued relevance.

Value

The library system plays a crucial role in supporting the library's operations and user interactions. Updating and automating this system will greatly benefit the school library by streamlining the book circulation process and improving record-keeping. These enhancements will encourage greater use of library resources by making them more accessible and efficient.

The benefits of implementing an updated and automated system include the following:

- **Faster Data Retrieval**: Users can quickly find necessary information, such as borrow details, due dates, and book information like titles and authors.
- **Improved Accuracy**: Automation minimizes errors in recording borrow and return dates, ensuring reliable and precise data management.
- **Reduced Administrative Workload**: Streamlining processes will free up staff to focus on more critical tasks, improving operational efficiency.
- **Cost Savings**: Reduced reliance on paper and printing materials will lower operational costs.
- **Enhanced System Reliability**: The updated system will be more stable and better documented, ensuring consistent performance.
- **Improved User Satisfaction**: A more user-friendly system will lead to a better experience for students and faculty.
- **Higher Staff Morale**: By reducing frustration and streamlining tasks, staff will experience greater job satisfaction.
- **Enhanced Library Reputation**: A modernized and efficient system will positively reflect on the library within the school community.

Design and Implementation

Functional requirements

The functional requirements outline the core capabilities of the proposed library system, and are designed to meet the client's primary need for a more efficient and automated solution. Currently, the DPA library manages a collection of 500 books, and the system is tailored to handle data at this scale. The primary user of the system is the librarian, who will utilize its features to streamline library operations and enhance overall efficiency. The following below provides a breakdown of the functional requirements critical to streamlining clinic operations:

1. **Automated Book Borrowing**: To streamline operations, the system includes automated book borrowing features. The borrow_book function validates ID numbers and accession numbers, enforces borrow limits depending whether the borrower is a student or a faculty member, and updates book statuses dynamically. Once the book is officially borrowed with an associated borrower, the status automatically becomes borrowed in the system. These functionalities are supported by a user-friendly interface through the borrow_book.html template, which allows users to input book accession numbers and ID numbers for borrowing, and prompts appropriate error messages for invalid inputs or borrow limit exceedances.
2. **Borrow Limits:** The library's business rules (students are limited to 1 book, and faculty to 2 books for borrowing) are implemented in the system, with checks implemented in the borrow_book function.
3. **Efficient Searching and Sorting through Algorithms**: The system includes a Linear Search algorithm implemented in the view_books function, which allows users to search with partial matches by title, author, publisher, or borrower name. The system also utilizes efficient algorithms like MergeSort and Binary Search to ensure fast borrowing of books given large datasets of 1000 books and 1000 users. Furthermore, MergeSort is used for quick sorting of the borrowed books by various criteria (e.g., title, author, publisher, days remaining, accession number, status). The view_books.html interface enhances this functionality with a sorting interface which enables users to sort books by the various criteria mentioned.
4. **Dynamic Handling of Book Statuses**: Book statuses are updated dynamically, reflecting their current availability, borrow, or overdue status based on user actions across the system.
5. **Automated Tracking of Days Remaining**: The system also allows the tracking of days remaining until the book should be returned by the customer which is derived from the day the book was borrowed, with a fixed deadline of return of five days.
6. **Return of Overdue Books and Overdue Fine Calculation**: Overdue fines are displayed when the user returns overdue books, wherein which user is prompted for confirmation. The system calculates overdue fines based on the number of days overdue with a fine rate of 5 pesos per day and displays these calculations on the confirm_return.html page, ensuring transparent and accurate fine management.

7. **Error Handling**: User-friendly error messages for invalid user IDs, exceeding borrow limits, and unavailable books are shown on the borrow page.

Collectively, these enhancements address the client's requirements for a reliable and efficient library system, reducing manual workload, minimizing errors, and ensuring a better user experience for the librarian.

System requirements

The system requirements describe the needed hardware and software requirements needed to run the system. For the system, a Python Django framework was chosen for the technical stack, using HTML, CSS, & Bootstrap for the UI.

There aren't necessarily system requirements to be able to run Django, but when installing Django, Python (including Pip) has its own system requirements. To be able to run the latest version of Python, version 3.13 (as of Dec 2024), and a Django framework listed below are the minimum requirements and recommended requirements.

Minimum system requirements:

1. OS: Windows 7/8/10, macOS 10.13 and later, Linux
2. Storage: 10 GB
3. RAM: 2 GB
4. CPU: 1 GHz single-core processor or any simple processor

Recommended system requirements:

1. OS: Any latest version of Windows (10 or 11), macOS, and Linux
2. Storage: 50 GB HDD or SSD
3. RAM: 4 GB
4. CPU: Dual-core processor (2.0 GHz or higher)
5. Other applications that can be used: Pycharm, Visual Studio Code

Proposed solution approach

Two types of algorithms that were chosen for the system are sorting and searching. For the sorting algorithm of the system, **Merge Sort** was chosen as, compared to other algorithms, its performance remains consistent in all possible cases with a Big-Notation of O(n log n). It is used in the system to efficiently sort large datasets, such as the lists of users and books, ensuring optimal performance even as the data grows. Additionally, MergeSort's efficiency allows for quick binary search operations on the sorted lists, further enhancing the overall speed of the borrow_book and view_books functions. Lastly, MergeSort is also used in the view_books function, which allows users to sort the books by various criteria (e.g., title, author, publisher, days remaining, accession number, status).

As for the searching algorithm, both **Binary Search** and **Linear Search** were chosen. The choice is both for practical reasons and performance. Binary Search will be used when

searching the accession number and user ID as this algorithm relies on the whole value/query to get a match and is faster and more consistent as a search algorithm for sorted, larger dataset. As for Linear Search, it will be used for searching in the borrowed table for fields such as Title, Author, Publisher, and Name. There will be cases when a portion of a value/query (ex: one word from the title, only the first or last name of the author, etc.) is used when searching for books. With Linear Search, partial matches are valid values, making searching easier for the user.

Project timeline

Due Date	Task
October 20, 2024	**Project Implementation & Proposal** ● Project Description ○ Scope ○ Features ● Algorithms to be Used ● Rough Project Timeline
November 25 to December 1, 2024	**Design & Implementation** ● Create system ○ Models and Classes ○ Views and Templates ■ Algorithm ■ UI ○ Dataset **Analysis & Documentation** ● Analysis of loading speed ○ Time space complexity on 3 datasets ● Create paper
December 2, 2024	● **Pass Paper & Presentation** ● **Defense Presentation**

Actual Implementation

Model implementation

Django was used to create the models for the books and users. These models are named Book and User. The columns for the attributes are included in the models and these attributes store the specific types of data necessary for the software. Meanwhile, methods are also utilized because they provide additional functionality and encapsulate key business logic related to managing library books.

The **Book model** encapsulates all necessary attributes and methods to manage book records in a library system, offering built-in logic for tracking borrowing status and return deadlines.

```
class Book(models.Model):
    ACCESSION_NUMBER_CHOICES = [
        ('borrowed', 'Borrowed'),
        ('available', 'Available'),
        ('returned', 'Returned'),
    ]

    accession_number = models.CharField(max_length=20, unique=True)
    author = models.CharField(max_length=100)
    title = models.CharField(max_length=200)
    publisher = models.CharField(max_length=100)

    date_borrowed = models.DateTimeField(null=True, blank=True)
    status = models.CharField(max_length=10, choices=ACCESSION_NUMBER_CHOICES, default='available')
    # Associate the book with a user
    user = models.ForeignKey('User', null=True, blank=True, on_delete=models.SET_NULL)
```

Book model Fields:

1. accession_number
    a. The unique identifier for each book
2. author, title, publisher
    a. These fields store the book's author, title, and publisher respectively.
3. date_borrowed
    a. A DateTimeField that records the date and time a book was borrowed.
4. status
    a. A CharField that tracks the book’s status (e.g., borrowed, available, or returned).
    b. It uses predefined choices (ACCESSION_NUMBER_CHOICES) and defaults to 'available'.
5. user
    a. A ForeignKey relationship links the book to a User model, indicating who borrowed it.

```python
@property
def expected_return_date(self):
    if self.date_borrowed:
        return self.date_borrowed + timedelta(days=5)
    return None # if not borrowed
```

```python
@property
def days_remaining(self):
    if self.date_borrowed:
        due_date = (self.date_borrowed + timedelta(days=5)).date()
        today = now().date()
        return (due_date - today).days
    return 0
```

Book model Methods:

1. expected_return_date (Property):
    a. Calculates the expected return date for a borrowed book.
    b. Adds 5 days to the date_borrowed using timedelta.
    c. Returns None if the book hasn't been borrowed.
2. days_remaining (Property):
    a. Calculates how many days remain until the book is due.

The **User model** represents users (students or staff) of the library system. This model allows the library system to store and differentiate between its users, assigning each a unique ID number and specifying whether they are a student or a staff member. Moreover, it maintains essential details like names, unique IDs, and user roles while enforcing data integrity through constraints like unique IDs and predefined user types.

```python
class User(models.Model):
    USER_TYPE_CHOICES = [
        ('Student', 'Student'),
        ('Staff', 'Staff'),
    ]

    id_number = models.CharField(max_length=6, unique=True)  # 6-digit unique ID
    name = models.CharField(max_length=100)
    user_type = models.CharField(max_length=10, choices=USER_TYPE_CHOICES)  # Either Student or Staff

    def __str__(self):
        return f"{self.name} ({self.user_type})"
```

User model fields:

1. id_number
    a. The unique identification number for each user.
2. name
    a. Used to store the user's full name.
3. user_type
    a. Used to classify users as either "Student" or "Staff."
    b. The field is constrained by predefined choices in the USER_TYPE_CHOICES list, ensuring only valid values are stored.
4. USER_TYPE_CHOICES
    a. A list of tuples that defines the valid options for the user_type field.
    b. Each tuple contains a database value (e.g., 'Student') and a human-readable label (e.g., 'Student').

Linear Search Algorithm

As mentioned in our proposed solution approach, Linear Search was chosen as one of the searching algorithms for the system, for the purpose of the algorithm being able to take partial match values. This algorithm is implemented in the view_books page of the system, where users can search for borrowed books and their associated text-based values, such as Title, Author, Publisher, and Borrower Name.

The function linear_search executes the Linear Search algorithm. First, an empty list (matched_books) is initialized (which will hold the books that match the search query). The code then goes through the whole list of books. Then, it extracts and appends all matches to matched_books and returns the results.

```python
def linear_search(books, query, key_func):
    matched_books = []

    for book in books:
        if query.lower() in key_func(book).lower():
            matched_books.append(book)

    return matched_books
```

This function is then used in the function def view_books and is applied to a search form in the HTML template of the view_books page. In the code below, when a query is sent into the search form matches the book's title, author, or publisher in the borrowed_books list, the page will display the books that matches the query.

```python
    # Apply linear search if a query is present
    search_query = request.GET.get('search', '')
    if search_query:
        borrowed_books = linear_search(
        borrowed_books,
        search_query,
        key_func=lambda book: f"{book.title} {book.author} {book.publisher}"
    )
```

```html
<div class="search-container">
    <label for="search" style="padding-right: 3px;">Search Books:</label>
    <input type="text" id="search" name="search" placeholder="Search by Title" value="{{ search_query }}">
    <button type="submit" class="btn btn-light btn-block">Search</button>
    <a class="btn btn-light btn-block" role="button" href="{% url 'view_books' %}">Reset</a>
</div>
```

Binary Search Algorithm

In the system, Binary Search is an iterative and recursive algorithm implemented to efficiently locate specific records from sorted lists of User and Book objects in the borrow_book view function. Binary Search is a search algorithm used to find the position of a target value within a sorted array. This works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. With a logarithmic time complexity of O(log n), it allows the system to handle large numbers of users or books efficiently, especially since the search space is halved with every iteration. Moreover, Binary Search is a natural fit here because the datasets are pre-sorted using the merge_sort function, enabling the binary search to operate correctly and efficiently.

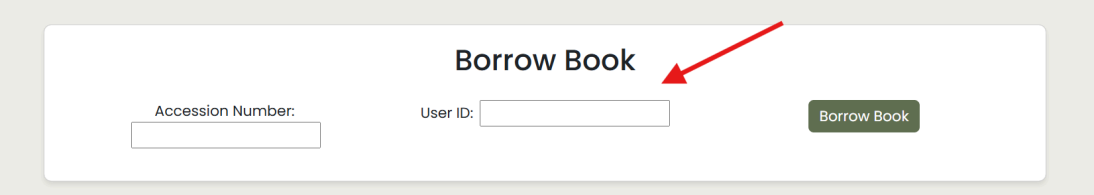
The binary_search function implements the binary search algorithm that searches for a specific item in a sorted list of objects. The function takes three parameters: arr, the list of objects to search; key, the target value to find; and key_attr, the attribute of the objects in the list that should be compared with the key. The algorithm uses two pointers, low and high, to define the search range, which initially covers the entire list. At each step, it calculates the middle index, mid, of the current range and retrieves the value of the specified attribute (key_attr) for the object at that index using getattr. The value is then converted to lowercase to ensure case-insensitive comparison.

```
def binary_search(arr, key, key_attr):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        mid_key = getattr(arr[mid], key_attr).lower()
```

If the middle value matches the target key, the corresponding object is returned. If the middle value is less than the key, the search continues in the upper half of the list by adjusting the low pointer to mid + 1. Conversely, if the middle value is greater than the key, the search narrows to the lower half by setting the high pointer to mid - 1. This process repeats until the target key is found or the low pointer exceeds the high pointer, indicating that the key is not in the list. In the latter case, the function returns None.

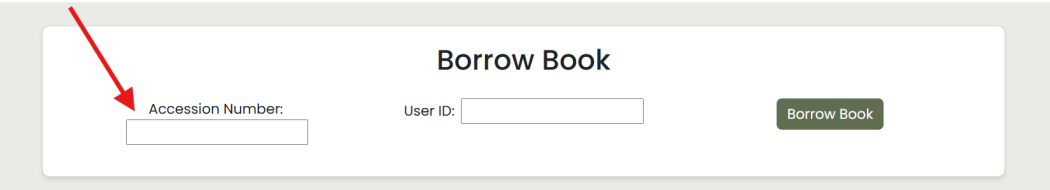
```
        if mid_key == key.lower():
            return arr[mid]
        elif mid_key < key.lower():
            low = mid + 1
        else:
            high = mid - 1
    return None
```

The borrow_book function handles the process of borrowing books. When a user submits their ID number (user_id) during the borrowing process, the system performs a binary search on the sorted all_users list, using the id_number attribute as the search key. This process divides the list into halves repeatedly, narrowing the search space until the user is either found or determined to be absent. If the user is not found, the function returns None, and an error message is displayed, indicating that the ID number is invalid.




```
if request.method == "POST":
    accession_number = request.POST.get('accession_number')
    user_id = request.POST.get('user_id')

    try:
        # Perform binary search for user
        user = binary_search(all_users, user_id, 'id_number')
        if user is None:
            return render(request, 'libapp/borrow_book.html', {
                'book_list': book_list,
                'error': 'ID number not found.'
            })
```

At the same time, when the user provides a book's accession number (accession_number), a binary search is performed on the sorted available_books list, using the accession_number attribute to locate the book. If the book is not found, the system responds with an error message stating that the book is either unavailable or already borrowed.




```
    # Perform binary search for books
    book = binary_search(available_books, accession_number, 'accession_number')
    if book is None:
        return render(request, 'libapp/borrow_book.html', {
            'book_list': book_list,
            'error': 'Book not found or currently borrowed.'
        })
```

Sorting Algorithm

MergeSort is a recursive, comparison-based algorithm that sorts data with a time complexity of O(n log n) and a space complexity of O(n) due to the auxiliary space required during the merge step. It will be used on the “currently borrowed books” screen to sort books by criteria such as accession number, title, author, publisher, remaining days, and status. Compared to quadratic algorithms like BubbleSort or InsertionSort, MergeSort is more efficient, particularly for larger datasets, making it suitable for the library's book collection. Additionally, MergeSort is a stable sort, which is essential when handling complex records like books.

The merge_sort function implements the MergeSort algorithm. It recursively divides an array arr into two halves, sorts each half, and then merges them back together in sorted order. The key_func parameter allows for custom sorting based on specific criteria (e.g., sorting books by title or author). It returns a sorted list of elements.

```
def merge_sort(arr, key_func):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key_func)
    right_half = merge_sort(arr[mid:], key_func)

    return merge(left_half, right_half, key_func)
```

The merge function is used to combine two sorted lists (left and right) into a single sorted list. It compares elements from both lists using a specified key_func (which could be attributes like title or accession number) and adds the smaller element to the sorted_list. It then appends any remaining elements from either list.

```
def merge(left, right, key_func):
    sorted_list = []

    while left and right:
        left_key = key_func(left[0]).lower() if isinstance(key_func(left[0]), str) else key_func(left[0])
        right_key = key_func(right[0]).lower() if isinstance(key_func(right[0]), str) else key_func(right[0])

        if left_key <= right_key:
            sorted_list.append(left.pop(0))
        else:
            sorted_list.append(right.pop(0))

    sorted_list.extend(left)
    sorted_list.extend(right)

    return sorted_list
```

As previously mentioned, the borrow_book function handles the process of borrowing books. It retrieves all users and available books from the database and uses MergeSort to sort both lists—users by their ID numbers and books by their accession numbers. After sorting, binary search is performed on the sorted lists to efficiently find the specific user and book based on their unique identifiers (user ID and book accession number). Once the user and book are located, the function checks whether the user has reached their borrowing limit and if the book is available. If all conditions are met, the book's status is updated to 'borrowed', and the transaction is saved.

```
def borrow_book(request):
    # Fetch users and available books from the database
    all_users = list(User.objects.all())
    available_books = list(Book.objects.filter(status='available'))

    # Sort users and books on every request
    all_users = merge_sort(all_users, key_func=lambda user: user.id_number)
    available_books = merge_sort(available_books, key_func=lambda book: book.accession_number)
```

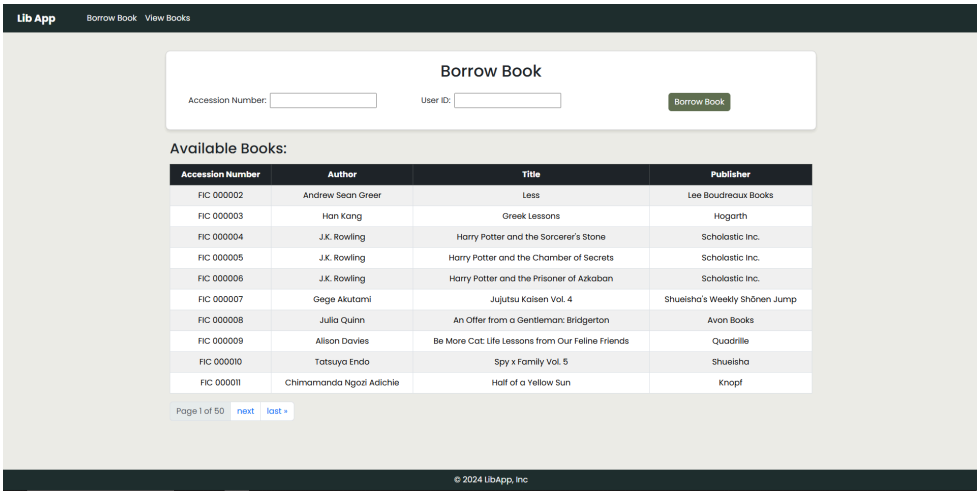
MergeSort is also used in the view_books function, which allows users to sort the books by various criteria (e.g., title, author, publisher, days remaining, accession number, status). Depending on the selected sorting criterion, the function applies the appropriate sorting by passing the corresponding key_func to the merge_sort function.

```
def view_books(request):
    borrowed_books = Book.objects.filter(status__in=['borrowed', 'overdue'])
    current_date = timezone.now().date()

    # Sorting criteria
    sort_field = request.GET.get('sort', 'title')

    # Apply sorting based on selected field
    if sort_field == 'title':
        borrowed_books = merge_sort(borrowed_books, key_func=lambda book: book.title)
    elif sort_field == 'author':
        borrowed_books = merge_sort(borrowed_books, key_func=lambda book: book.author)
    elif sort_field == 'publisher':
        borrowed_books = merge_sort(borrowed_books, key_func=lambda book: book.publisher)
    elif sort_field == 'days_remaining':
        borrowed_books = merge_sort(borrowed_books, key_func=lambda book: book.days_remaining)
    elif sort_field == 'accession_number':
        borrowed_books = merge_sort(borrowed_books, key_func=lambda book: book.accession_number)
```

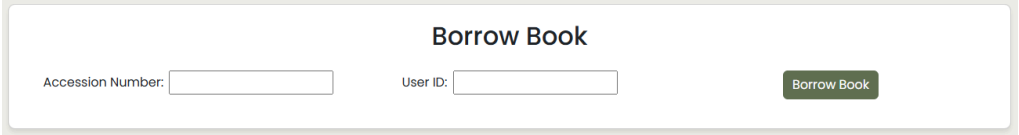
```
return render(request, 'libapp/view_books.html', {
    'books': borrowed_books,
    'search_query': search_query,
    'sort_field': sort_field,
})
```

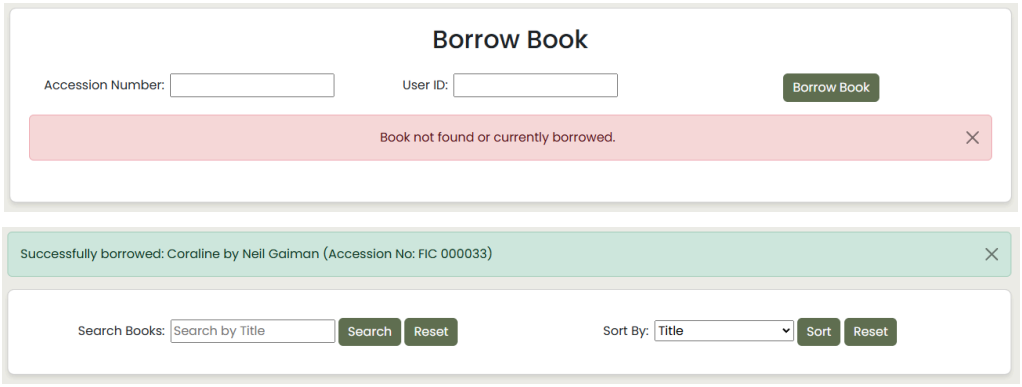
User Interface

The user interface of the system should be user friendly. When you open the system, the user will be greeted by the main page of the system, the Borrow Book page. The page contains a table of the currently available books. The table only contains 10 per page for the user to easily view the books by flipping through the pages.

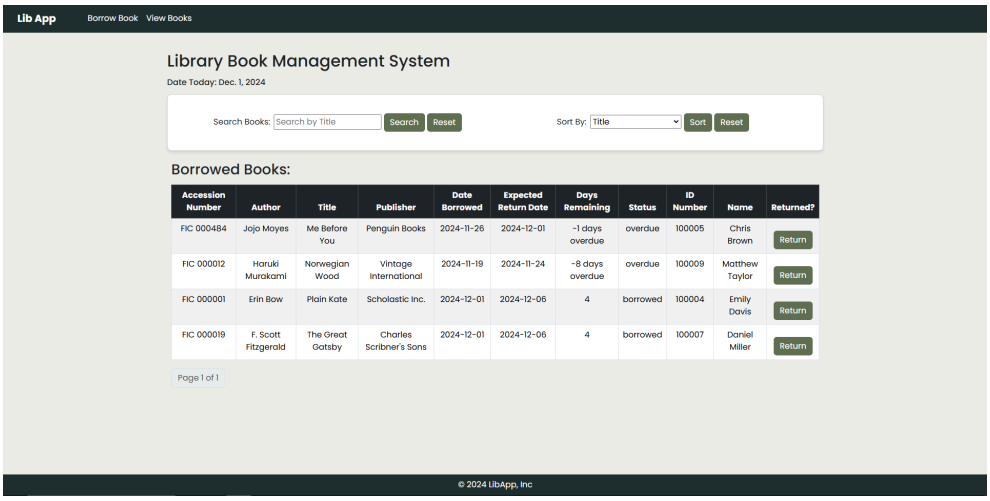


The page also contains a “Borrow Book” form where the user will input the Accession Number of their chosen book and their User ID. If the book is currently borrowed or the user has reached maximum book borrows (or if there are other contingencies), an alert will appear below the form to inform the user of the error. If the book has been successfully borrowed, the user will be led to the View Books page and a success alert will appear above the search and sorting form of the page.

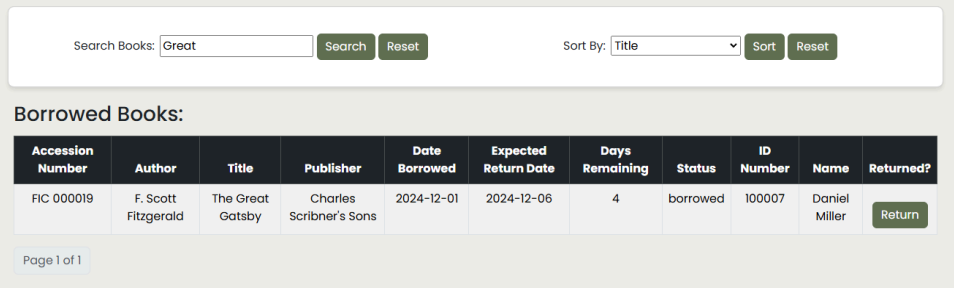


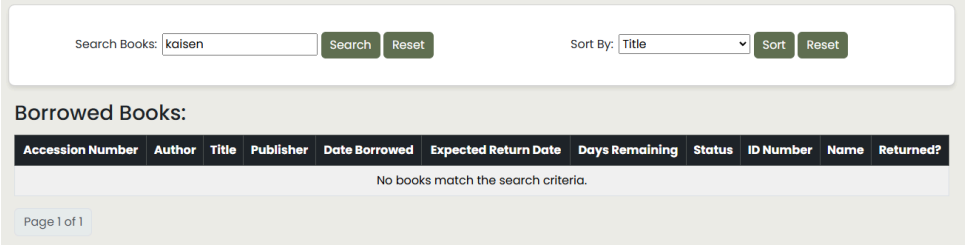


The View Books or the Library Book Management System page is where the user will be able to view the currently borrowed books. The user will also be able to search and sort the books using the form above the table.

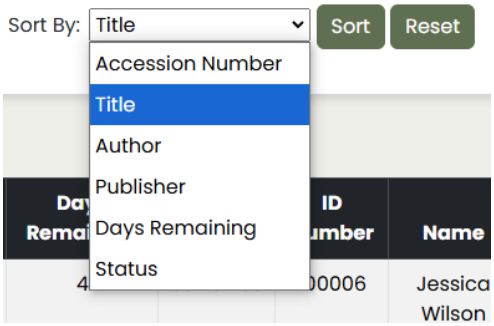


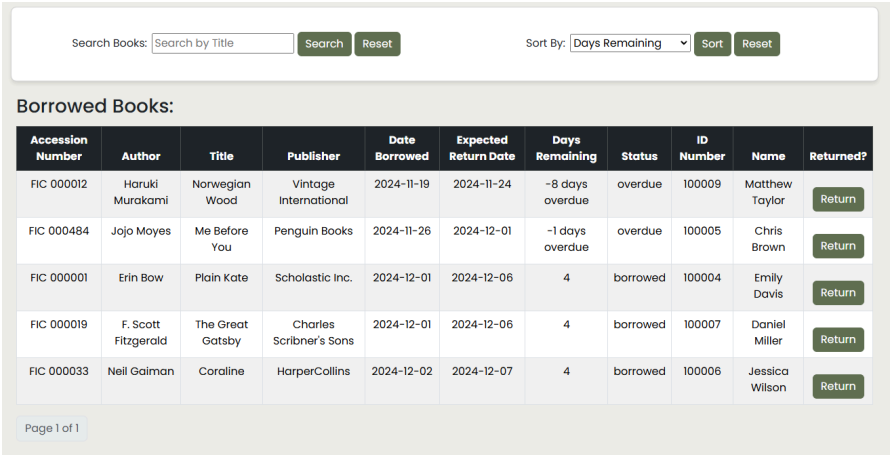
To search, the user will input a query into the search form. If the query matches any of the books in the table, the book will be posted on the page. Otherwise, the page will not post an alert saying that no books match the search criteria.



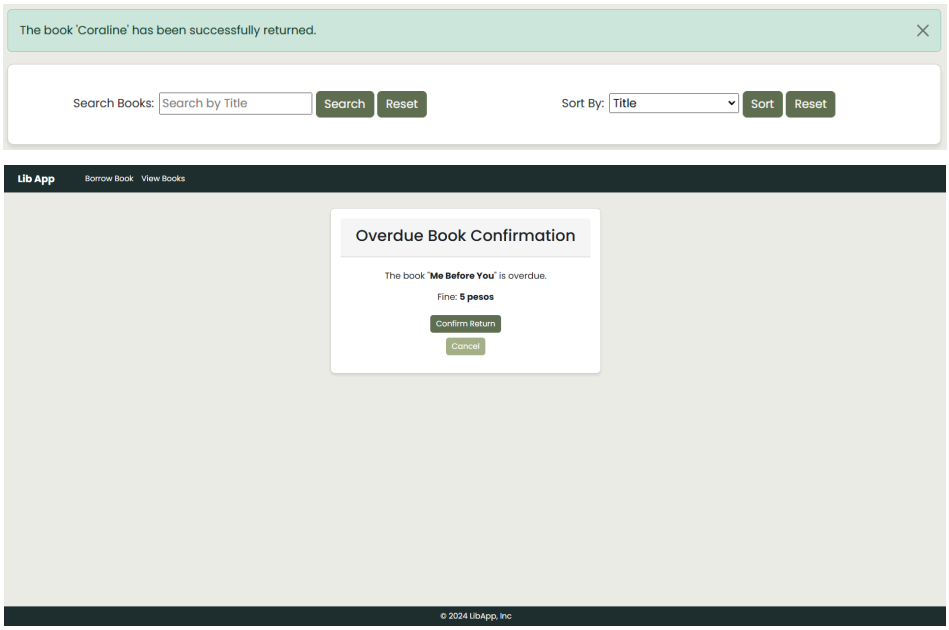


To sort, the user will click on the ‘Sort By’ drop down and choose which of the options they want to sort the table by. Once the option has been selected and the sort button has been clicked, the table will be sorted according to the option selected.





When returning a book, the user will click on the return button on the ‘Returned?’ column. If the book is not overdue, a successful alert will appear above the search and sort form. If the book is overdue, the ‘Overdue Book Confirmation’ page will appear. The page contains the title of the book that is overdue along with the fee that needs to be paid. The fee is determined by how many days the book is overdue (5 pesos per day).



Data Structures Used

The system makes use of Lists, Models, & Classes for its implementation. Lists were used throughout the system to store values. Below is an example of its use. Here, an empty list is implemented. Once the while loop has been satisfied, the values extracted from the loop will be appended to the list.

```
def merge(left, right, key_func):
    sorted_list = []

    while left and right:
        left_key = key_func(left[0]).lower() if isinstance(key_func(left[0]), str) else key_func(left[0])
        right_key = key_func(right[0]).lower() if isinstance(key_func(right[0]), str) else key_func(right[0])

        if left_key <= right_key:
            sorted_list.append(left.pop(0))
        else:
            sorted_list.append(right.pop(0))

    sorted_list.extend(left)
    sorted_list.extend(right)

    return sorted_list
```

As for models, it is used as the main values throughout the system. This includes the details of the books like accession number, author, title, etc.

```
accession_number = models.CharField(max_length=20, unique=True)
author = models.CharField(max_length=100)
title = models.CharField(max_length=200)
publisher = models.CharField(max_length=100)

date_borrowed = models.DateTimeField(null=True, blank=True)
status = models.CharField(max_length=10, choices=ACCESSION_NUMBER_CHOICES, default='available')
# Associate the book with a user
user = models.ForeignKey('User', null=True, blank=True, on_delete=models.SET_NULL)
```

Lastly for classes, it is used as the classification of models. This includes our classes Book and User.

```
class Book(models.Model):
    ACCESSION_NUMBER_CHOICES = [
        ('borrowed', 'Borrowed'),
        ('available', 'Available'),
        ('returned', 'Returned'),
    ]

class User(models.Model):
    USER_TYPE_CHOICES = [
        ('Student', 'Student'),
        ('Staff', 'Staff'),
    ]
```

Testing Procedures and Results

Binary Search vs. Linear Search

To evaluate the time complexity of the Binary Search algorithm for searching books, the group conducted experiments with datasets of varying sizes. Additionally, the group compared Binary Search with Linear Search, which is widely known for its simplicity and versatility as a search algorithm. The tests were performed using datasets of 100, 500, and 1,000 books. For each dataset size, both sorting methods were executed 10 times, and the time taken for each execution was recorded in seconds. The following steps were followed in the experiment:

- **Data Generation**: The group loaded datasets of 100, 500, and 1,000 books using the load_books command, where each book contained various details and unique accession numbers.
- **Searching**: The Binary Search algorithm was applied to search a specific book to be borrowed by accession number. While Binary Search was also used on searching the ID Number in the book borrowing process, it is excluded because the general goal of this experiment is only to simply measure the sorting performance of a specific implementation of the algorithm, and searching the accession number was chosen by the group. This same approach was used for Linear Search.
- **Time Measurement**: The execution time for each search operation was recorded using Python's time.perf_counter() method. This method is used instead of time.time() because it provides a higher-resolution timer that is more suitable for measuring the performance of short durations with greater precision, which Binary Search requires. The start time was captured before applying the search function, and the end time was recorded after the search was completed. The time taken to sort the books was then printed with a precision of 10 decimal places, with 10 executions per dataset size. To implement Linear Search, the search function in the borrow_book view function was simply changed to Linear Search, with a few adjustments on how the book was retrieved.
- **Analysis**: The average execution time for each search method and dataset size was calculated, and the results were analyzed to observe how the execution time grew relative to the input size.

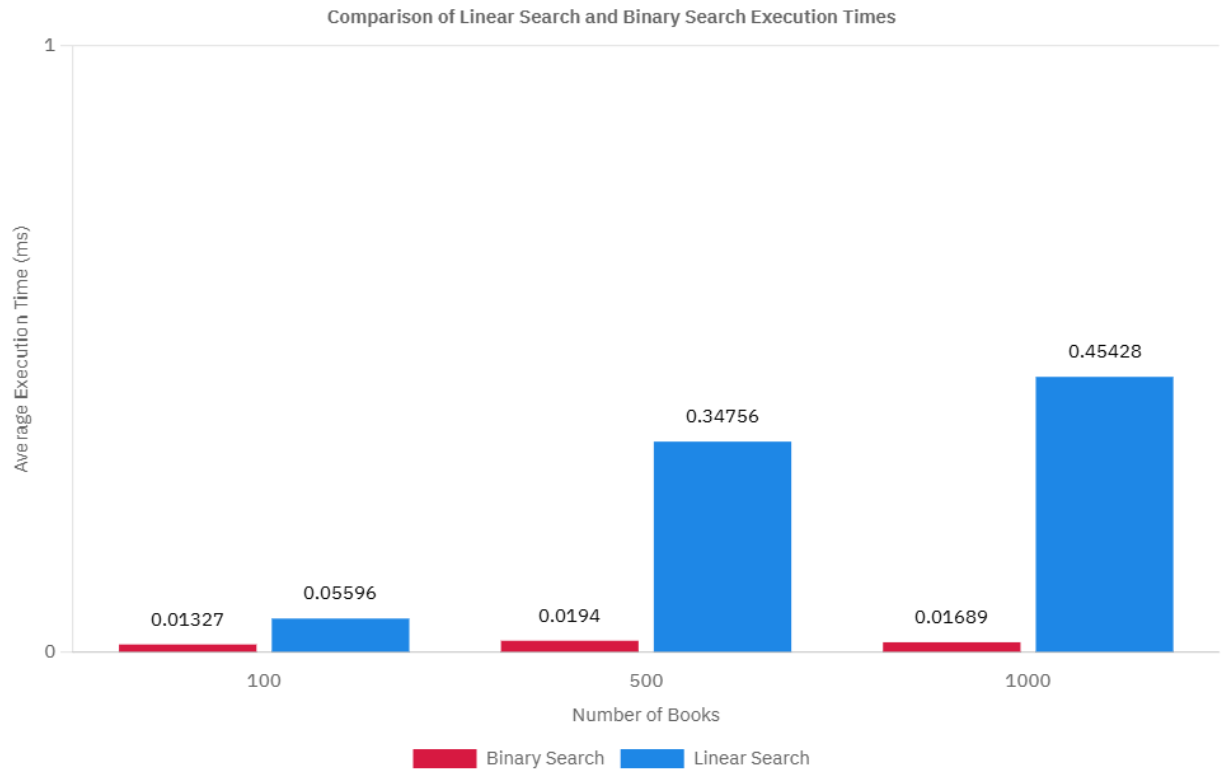


Figure 1. Time Complexity of Binary Search vs. Linear Search

The graph illustrates the execution times of binary search and linear search algorithms as the size of the dataset increases. The y-axis represents the average execution time (in milliseconds), while the x-axis shows the number of books (for dataset size). The red bars correspond to binary search, and the blue bars represent linear search.

Binary Search maintains a consistently low execution time across all dataset sizes, demonstrating its efficiency. This is due to its logarithmic time complexity, O(log n), which allows it to quickly narrow down the search space by halving it with each step. The execution time for Binary Search stays almost constant even when the dataset size increases from 100 to 1000.

On the other hand, when the amount of dataset increases, linear search shows a noticeable rise in execution time. There is a direct relationship between dataset size and execution time because of its linear time complexity O(n), which requires it to analyze every item in the list in the worst scenario. For smaller datasets, Linear Search works well, but as the dataset size increases, its inefficiency becomes more noticeable, as execution times increase sharply.

All-in-all, this experiment illustrates the superior efficiency of binary search, especially for large, sorted datasets, where its logarithmic scaling provides a clear advantage over the slower, linear growth of execution time observed with linear search.

Mergesort vs. Quicksort

To evaluate the time complexity of the MergeSort algorithm for sorting books, the group conducted experiments with datasets of varying sizes. Additionally, the group compared MergeSort with QuickSort, which is widely used and well-known for its efficiency in sorting large datasets. The tests were performed using datasets of 100, 500, and 1,000 books. For each dataset size, both sorting methods were executed 10 times, and the time taken for each execution was recorded in seconds. The following steps were followed in the experiment:

- **Data Generation**: The group loaded datasets of 100, 500, and 1,000 books using the load_books command, where each book contained various details and unique accession numbers.
- **Sorting**: The MergeSort algorithm was applied to sort the books by accession number, as the specific sorting criterion was not critical for the experiment, since the goal was to observe sorting performance in general. The same approach was used for QuickSort.
- **Time Measurement**: The execution time for each sorting operation was recorded using Python's time.time() method. The start time was captured before applying the sorting function, and the end time was recorded after the sorting was completed. The time taken to sort the books was then printed with a precision of 4 decimal places, with 10 executions per dataset size. To implement QuickSort, the sorting function in the view_books method was simply changed to QuickSort.
- **Analysis**: The average execution time for each sorting method and dataset size was calculated, and the results were analyzed to observe how the execution time grew relative to the input size.

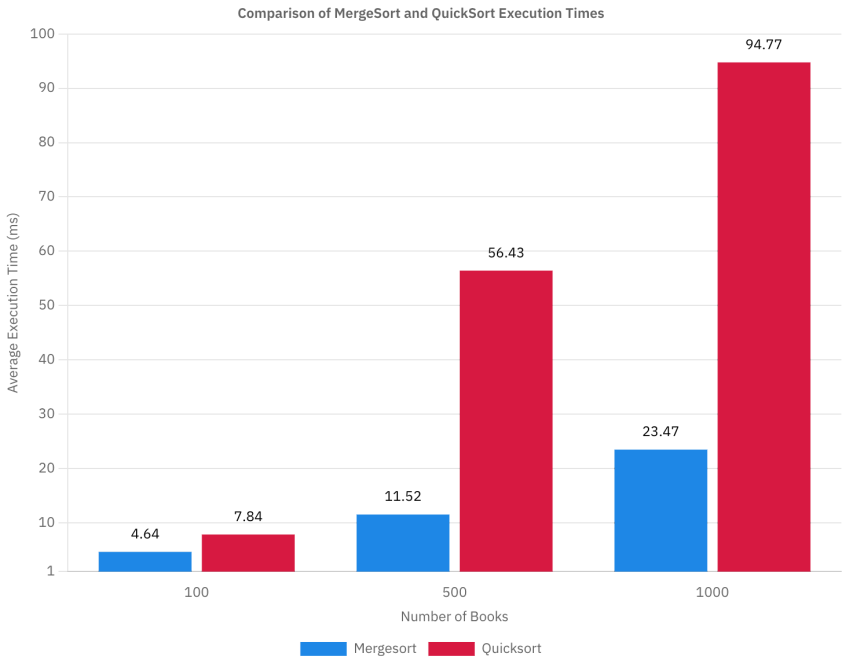


Figure 2. Time Complexity of Mergesort vs. Quicksort

The graph above compares the execution times of MergeSort and QuickSort for datasets of 100, 500, and 1,000 books. The execution times are converted to milliseconds for clearer comparison, with the y-axis representing execution time and the x-axis representing the number of books in the dataset.

For 100 books, the difference in performance between MergeSort and QuickSort is minimal. However, for 500 and 1,000 books, the gap becomes more noticeable. While both algorithms show a similar growth pattern, QuickSort slows down more significantly as the dataset increases, particularly for 1,000 books. MergeSort, on the other hand, maintains consistent performance regardless of the dataset size.

MergeSort has a time complexity of **O(n log n)**, meaning the time taken to sort the books increases proportionally to the product of the number of books and the logarithm of that number. Its space complexity is **O(n)** due to the auxiliary space required during the merging process. As expected, execution time increases with the dataset size, confirming the O(n log n) time complexity. The growth is not linear but follows a slightly faster increase due to the logarithmic factor.

QuickSort, which also has an average time complexity of **O(n log n)**, was tested for comparison. However, QuickSort's execution time grows more significantly with larger datasets. This difference can be explained by QuickSort's worst-case time complexity of **O(n^2)**, which occurs in specific scenarios, particularly with poor pivot selection. This highlights MergeSort's greater stability and predictability in performance when sorting book datasets. The consistent execution time of MergeSort across different dataset sizes demonstrates its reliability for efficiently handling larger datasets.

Potential Improvements

The group suggests testing more algorithms to know more efficient algorithms that can be a perfect fit for the system. It is given that the client's book list will grow in the future so it is in the best interest of the group to test more algorithms that can efficiently manage larger datasets.

As for the system, though it is out of the scope of the current project, it is suggested for the system to have an archive or a history of the book's borrowings, including a history of the user's book borrows. Its main purpose is to be able to have more information about the borrowings of both books and users. It is also suggested for the system to have a Log-In method for both user determination and data integrity. The group suggests this for when the project's scope is bigger and improvement of the system is requested.

References

GeeksforGeeks. (2024, September 4). *Binary Search Algorithm – Iterative and Recursive Implementation*. GeeksforGeeks. https://www.geeksforgeeks.org/binary-search/

GeeksforGeeks. (2024, August 19). *What are the minimum hardware requirements for python programming?* GeeksforGeeks. https://www.geeksforgeeks.org/what-are-the-minimum-hardware-requirements-for-python-programming/

Coding Snow. (2020, January 31). *Simply Create a Glowing Shadow Hover Button- for Beginners - using CSS, HTML*. YouTube. https://www.youtube.com/watch?v=Pcf4F5xa1xs

Mark Otto, J. T. (n.d.). *Pagination*. Bootstrap. https://getbootstrap.com/docs/4.0/components/pagination/