**Advanced Data Structures**

**COP 5536 Fall 2023**

# Programming Assignment #1

**Nam**e: Manikumar Honnenahalli Lakshminarayana Swamy

**UFID:** 24369711

**Email**: [manikuma.honnena@ufl.edu](mailto:manikuma.honnena@ufl.edu)

## Program Overview:

The GatorLibrary Management System is a software application designed to efficiently manage the library's books, patrons, and borrowing operations. The system utilizes Red-Black tree to ensure the efficient organization and retrieval of book-related information. Additionally, a priority-queue mechanism is implemented using Binary Min-heaps to manage book reservations when a particular book is not currently available for borrowing.

## Compilation and Execution:

### Windows, MacOS and Linux

The project has been tested with the latest stable versions of Java, specifically versions 17 and 11, on both Windows and MacOS. Given that Java commands are operating system agnostic, these commands are expected to function similarly in a Linux environment. Additionally, the project has been tested on the thunder.cise.ufl.edu server.

**Instruction to execute the program:**

1. Unzip the folder "HonnenahalliLakshminarayanaSwamy\_Manikumar.zip" and navigate to the extracted directory.
2. Open the terminal in the current directory and execute the 'make' command to compile the source files, generating executable files.
3. Run the command 'java gatorLibrary {FILE\_NAME.txt}', replacing '{FILE\_NAME.txt}' with the actual input file name.
4. After the execution completes, observe the message '{FILE\_NAME.txt} is created successfully!!'. Additionally, check for the generated '{FILE\_NAME\_output\_file.txt}' created by the program.

## Key Features/Objective:

1. **Book Management:**

* The system maintains a Red-Black tree data structure for organizing and managing books.
* Each book record contains essential information such as book Id, book name, author name, borrowed by, availability status and reservation heap to maintains the reservation of patrons.

1. **Red-Black Tree Implementation:**

* Red-Black trees are utilized for the effective insertion and deletion of book-related information.
* The Red-Black tree structure guarantees a balanced and efficient data structure, ensuring the maintenance of balance after each insertion or deletion operation, optimizing the search and update operations for book records.

1. **Borrowing and Reservation Operations:**

* Patrons are registered in the system with relevant details such as patron id, patrons’ priority, and time of reservation to break the tie if the patrons have same priority.
* Patrons can borrow available books, and the system updates the book's availability status accordingly.
* In cases where a book is not available, patrons can make reservations.
* Each book has its own Binary Min-heap to manage reservations based on priority.

1. **Binary Min-Heap Implementation:**

* Binary Min-heaps are utilized to implement a priority-queue mechanism for book reservations.
* Patrons making reservations are prioritized based on patron’s priority, if there is a same priority then will break the tie with reservation timestamp.

1. **Input/Output Handling:**

* The system accepts the input file name as a command-line argument and the system incorporates `InputStreamReader` and `BufferedReader` for reading input data, such as book records and patron details, etc. from input file.
* Output data, including system responses such as printing the book information, color flip count information, borrowing and reservation book information and removing the book information, is written to file using `OutputStream and `FileOutputStream`.

Function Prototypes and Structure:

The GatorLibrary Management System is structured in a modular and organized manner to facilitate efficient development, and maintenance. The following classes make up the program structure:

1. Book.java
2. gatorLibrary.java
3. RedBlackTree.java
4. Reservation.java
5. Utility.java
6. **Book.java:**

* The Book class serves as a fundamental component in the library management system, encapsulating essential attributes like book ID, title, author, and availability status. With a dedicated list for reservations and references for Red-Black Tree structure. The class incorporates a customized toString() method which converts the book object to string .

1. **gatorLibrary.java:**

* The class is the main program serves as the entry point for the application.
* **Command Patterns:** class defines regular expression patterns to match different input commands, such as inserting a book, printing book information, borrowing and returning books, deleting books, finding the closest book, and checking color flip count.
* **Input Handling:** The main method reads the input file name from the command line arguments. If no arguments are provided, it prompts the user to supply the input file name.
* **File Output:** The class creates an output file "{InputFile.txt} \_output\_file.txt" to store the results of the library operations.
* **Library Operations:** The class utilizes a RedBlackTree named library to manage the books and patrons in the library. It processes various input commands using regular expressions, matches patterns, and performs corresponding actions, such as inserting, printing, borrowing, returning, deleting, and finding books.
* **Utility Methods:** The class relies on a Utility object to perform auxiliary tasks, such as writing to the output file, adding new lines, and providing error messages.
* **Program Termination:** The program terminates upon encountering the "Quit ()" command, ensuring proper closure and output file completion.
* **Exception Handling:** The class includes exception handling to manage potential errors during file reading and other operations.

1. **RedBlackTree.java**

* This section encapsulates the core logic for the Red-Black tree, handling the insertion and deletion of books. Each node in the Red-Black tree corresponds to a book, which contains the details of books and a reservation heap for managing patron reservations. And has a space complexity O(N), where n is number of books in the library.

This file consists of following functions:

1. **getBookFromLibrary**(Book books): The getBookFromLibrary method navigates the library's binary search tree recursively, with a time complexity of O (log N), where N is the number of books and has a space complexity of O (log N) due to its recursive nature, where N represents the number of books in the binary search tree). It efficiently searches for a book based on the provided book ID, comparing it with the current node's ID. If a match is found, the current Book object is returned; otherwise, it continues the search in the left or right subtree.
2. **deleteBookFromLibrary**(bookId, outputStream, utility): The deleteBookFromLibrary method orchestrates the removal of a book, identified by its ID, from the Red-Black Tree in the library. It begins by searching for the book in the balanced tree structure, taking O (log N) time. If the book is found, the method generates a status message indicating its unavailability. Additionally, if there are reservations for the book, it cancels them and updates the message accordingly.

Subsequently, the method removes the book from the dictionary, executes the deleteBook method to perform the deletion while addressing Red-Black Tree property violations. Post-deletion, it checks for potential color changes in the tree nodes, and the worst-case time complexity for rotations and tree rebalancing operations is O (log N). The results, including unavailability status and reservation cancellations, are logged to an output stream using the Utility class.

1. **insertBook**(bookId, bookName, authorName, availabilityStatus): The insertBook method facilitates the seamless addition of a new book to the library's binary search tree, employing Red-Black Tree balancing mechanisms. It creates a new book node based on the provided details and performs a standard binary search tree insert, taking O (log N) time in the worst case. Subsequently, the method applies rotations and color adjustments to uphold the Red-Black Tree properties, ensuring the structural balance of the tree. Moreover, the update of a dictionary to monitor the color flip count for each book ID enhances the efficiency of book management operations. This method contributes significantly to the expansion of the library's collection while maintaining the integrity of the underlying Red-Black Tree.

1. **updateColorFlipCount**(): The updateColorFlipCount method efficiently tracks and updates the color flip count for each book ID in the library's Red-Black Tree. It iterates through the keys in the colorChanged map, which represents the books with modified colors. For each book ID, if the color change in the map differs from the current color in the dictionary, it increments the colorFlipCount and updates the dictionary with the new color. Finally, the colorChanged map is cleared. This method operates in O(K) time complexity, where K is the number of books with color changes, as it only processes the nodes that underwent color modifications, not the entire tree.
2. returnBook(patronId, bookId, access, utility): The returnBook method efficiently manages book return requests in the library system. It first searches for the specified book in the Red-Black Tree, taking advantage of the tree's organized structure to achieve a time complexity of O (log n), where n is the number of books in the library. The method checks whether the book was borrowed by the designated patron and write the logs to the file using the provided Utility instance using Output stream. If the returned book has reservations, it assigns the book to the patron with the highest priority from the reservation list, involving a heap operation with a time complexity of O (log n). Overall, the method ensures accurate and streamlined book return processing while optimizing search and reservation assignment operations.
3. **borrowBook**(bookId, patronId, outputStream, Utility: The borrowBook method facilitates the borrowing of a book by a patron, considering various scenarios and efficiently managing the library's records. It begins by searching for the book with the provided ID in the library's binary search tree, ensuring its availability. If the book is available, it is assigned to the patron, and its status is updated. The method handles cases where the book is already borrowed by the patron, reserved, or if the reservation limit is reached. If the book is not available, the patron is added to the reservation list, with a limit of 20 patrons per book.

The time complexity of this method is primarily determined by the search operation in the binary search tree (getBookFromLibrary method), which has a time complexity of O (log n), where n is the number of nodes in the tree. Additionally, the operation to fix the property of the reservation heap involves inserting a new reservation, which has a time complexity of O (log n), where n is the number of reservations in the heap. Overall, the dominant time complexity for the borrowBook method is O (log n).

1. **findClosestBook(bookid, outputStream)**: The findClosestBooks method navigates a binary tree of books to identify the closest books to a given book ID. It populates a map with the details of these closest books and prints the results using the provided Utility and OutputStream instances. The method uses a loop to traverse the tree, updating the closest book based on the absolute difference of book IDs. If an exact match is found, the details are stored in the TreeMap. In cases of ties, where there are two closest books, both are included in the result. The method then calls printTheBooksInOrder to display the book details in the desired order. Parameters include bookId for the target book, utility for book printing, and access for the output stream. It may throw an IOException in case of printing errors. This operation enhances the functionality of locating and displaying closest books in the binary tree. And the time complexity id O (log n) where n is the number of books on the library.

1. **printBooks**(start, end, utility, outputStream): The printBooks method retrieves and prints book details within a specified ID range in the library's binary search tree. Using the getBooksInRange method, it populates a TreeMap with the relevant book details. If no books are found in the range, it outputs a corresponding message; otherwise, it prints the books in order. The time and space complexity is O (n) for the getBooksInRange method, where n is the number of nodes in the tree, influencing the overall time complexity of the printBooks method.
2. **findPredcessor**(books): The findPredessor method efficiently locates the predecessor of a given book in the binary search tree by traversing the left subtree and finding the node with the maximum key value. This operation, crucial for maintaining Red-Black Tree properties during book deletion, has a time complexity of O (log n), where n is the number of nodes in the tree. The method ensures effective retrieval of the predecessor, contributing to the overall balance of the data structure.
3. **rotateLeft**(node): The rotateLeft method performs a left rotation on a specified node in a binary search tree. It updates parent and grandparent references, adjusts child relationships, and ensures the tree's balance. The method is crucial for correcting right-heavy imbalances, ultimately preserving the integrity of the Red Black tree. And this operation performs in O (1) time and space complexity.
4. **rotateRight**(node): The rotateRight method executes a constant time O (1) right rotation on a Red Black tree, essential for addressing left-heavy imbalances. It involves updating parent and grandparent references and adjusting child relationships. The space complexity is O (1) as it uses a fixed amount of memory for temporary variables, independent of the tree's size.
5. **rotateLeftRight**(node): The rotateLeftRight method performs a left-right rotation on a specified node in a Red Black tree. This operation involves updating parent and grandparent references and adjusting child relationships. Like single rotations, the time complexity is O (1) since it consists of a fixed number of pointer updates and comparisons. The space complexity is also O (1) as the method uses a constant amount of memory for temporary variables, independent of the tree's size. The method is essential for maintaining the balance and integrity of the Red Black tree, especially in scenarios where a left-right imbalance needs correction.
6. **rotateRightLeft**(node): The rotateRightLeft method performs a right-left rotation on a specified node in a Red Black tree, involving updates to parent and grandparent references as well as adjustments to child relationships. Like single rotations, the time complexity is O (1) as it consists of a fixed number of pointer updates and comparisons. The space complexity is also O (1) since the method utilizes a constant amount of memory for temporary variables, regardless of the tree's size. This method is crucial for maintaining the balance and integrity of the Red Black tree, particularly when correcting right-left imbalances.
7. **getSibling**(node): The getSibling method in the Red-Black Tree retrieves the sibling of a specified node. If the node has a parent, it examines whether the node is the left or right child of its parent. The sibling is then determined based on the sibling's position relative to the node. This method is essential in Red-Black Tree operations, particularly during rotations and color adjustments, to maintain the tree's balance and adherence to Red-Black Tree properties. The time complexity is O (1), and the space complexity is O (1), making it an efficient and integral part of Red-Black Tree algorithms.
8. **getBooksInRange** (root): The getBooksInRange method recursively traverses Red Black tree. It adds Book objects to a TreeMap if their book IDs fall within a given range. This method is useful for efficiently retrieving books within a specific ID range and relies on the TreeMap's natural ordering of keys.
9. **heapify** (reservationList, size, firstIndex): The heapify method is responsible for maintaining the min-heap property within a list of reservations. It identifies the indices of the left and right children, compares their priority and reservation times with the current smallest element, and recursively swaps elements if necessary. The method ensures that the smallest priority reservation is at the top, allowing efficient retrieval of the highest-priority reservation. The time complexity is O (log n), where n is the number of reservations in the heap, as it traverses the height of the heap. The space complexity is O (log n) due to the recursive nature of the method.
10. **Reservation.java:** This class represents a reservation made by a patron for borrowing a book. It encapsulates information such as the patron's ID, priority number, and the timestamp of the reservation. This class facilitates efficient management of reservations in the library system, allowing for prioritized book allocation based on patron priority levels and reservation times.

1. **Utility.java:** The Utility class in the book management system provides essential methods for writing data to files, handling book availability information, processing reservations, managing reservation timestamps, and ensuring integration with a RedBlackTree. Its functionality simplifies file operations, supports reservation handling, and contributes to the overall efficiency of the book management system.
2. **write** (data, outputStream): The write method writes the provided data as bytes to the specified output stream, followed by adding a new line. It utilizes the getBytes method for conversion and includes error handling for IOException to handle potential I/O issues during the write operation.
3. **writeTheBookData** (book, outputStream): The writeTheBookData method writes the string representation of the provided Book object to the specified output stream, followed by adding a new line. It utilizes the toString method of the Book class and includes error handling for IOException to handle potential I/O issues during the write operation.
4. getpatronIds (reservationList): The getpatronIds method retrieves and returns a list of patron IDs from a provided list of reservations, utilizing Java Stream API.
5. getTimeStampForReservation(): The getTimestampForReservation method increments and returns a timestamp variable, providing a unique value for reservation timestamps with each invocation. This timestamp used as tie breaker when two patrons have same priority.