COP 5536 Fall 2023 Programming Project

REPORT

GatorLibrary MANAGEMENT SYSTEM

Name: Jyothi Anvitha Ambati

UFID: 83307174

UF email ID: j.ambati@ufl.edu

Programming Language: Java

Compile - makefile

Execute - $ java gatorLibrary file_name

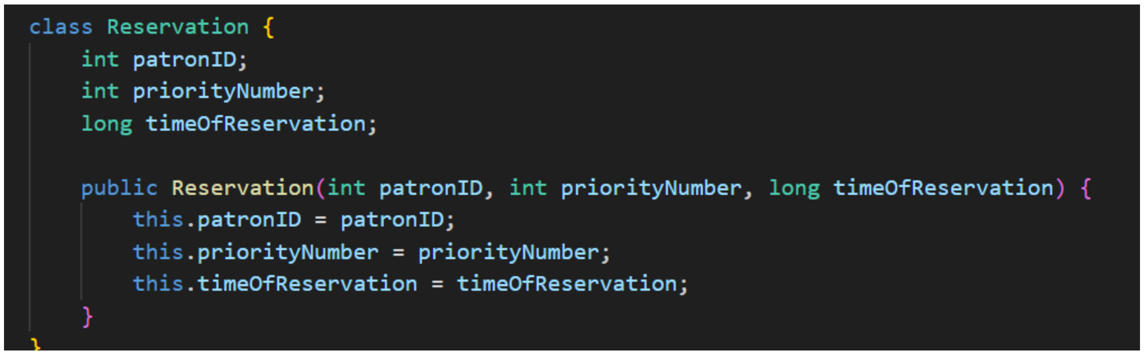
Test files - test1.txt , test2.txt, test3.txt, test4.txt

Output file - <inputfilename>_output.txt

This report presents the design and implementation details of a Library Management System developed in Java. The system encompasses classes for managing books, reservations, borrowing and patrons in a library setting.

The program consists of 5 classes.

1. **Reservation class:**
   a. patronID: An integer representing the ID of the patron who made the reservation.
   b. priorityNumber: An integer indicating the priority level of the reservation.
   c. timeOfReservation: A long value representing the timestamp of when the reservation was made.



```
class Reservation {
    int patronID;
    int priorityNumber;
    long timeOfReservation;

    public Reservation(int patronID, int priorityNumber, long timeOfReservation) {
        this.patronID = patronID;
        this.priorityNumber = priorityNumber;
        this.timeOfReservation = timeOfReservation;
    }
```

d. The constructor initializes the Reservation object with the provided values for patronID, priorityNumber, and timeOfReservation. It allows creating instances of reservations with specific details at the time of instantiation.

The space complexity per instance is O(1).

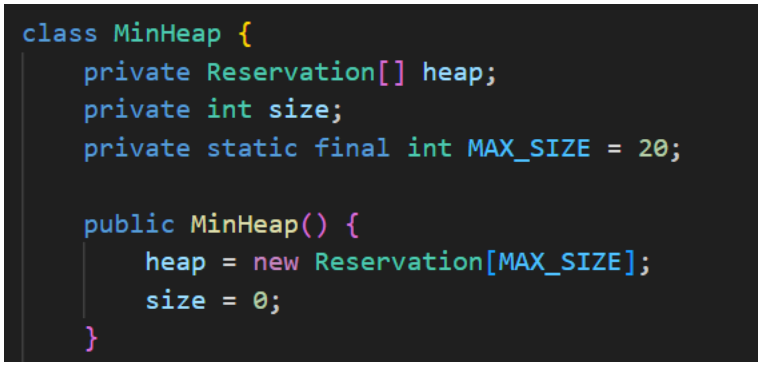
2. MinHeap Class

This MinHeap class implements a minimum heap data structure to manage a collection of Reservation objects.

```
class MinHeap {
  public MinHeap();
  public boolean isEmpty();
  public void insert(int, int, long);
  public Reservation extractMin();
  public java.lang.String toString();
}
```

Fields:

a. heap: An array of Reservation objects used to represent the heap.
b. size: An integer keeping track of the current number of elements in the heap.
c. MAX_SIZE: A constant defining the maximum capacity of the heap (set to 20 in this case).
d. Constructor: Initializes the heap array with the specified MAX_SIZE and sets the initial size to 0.

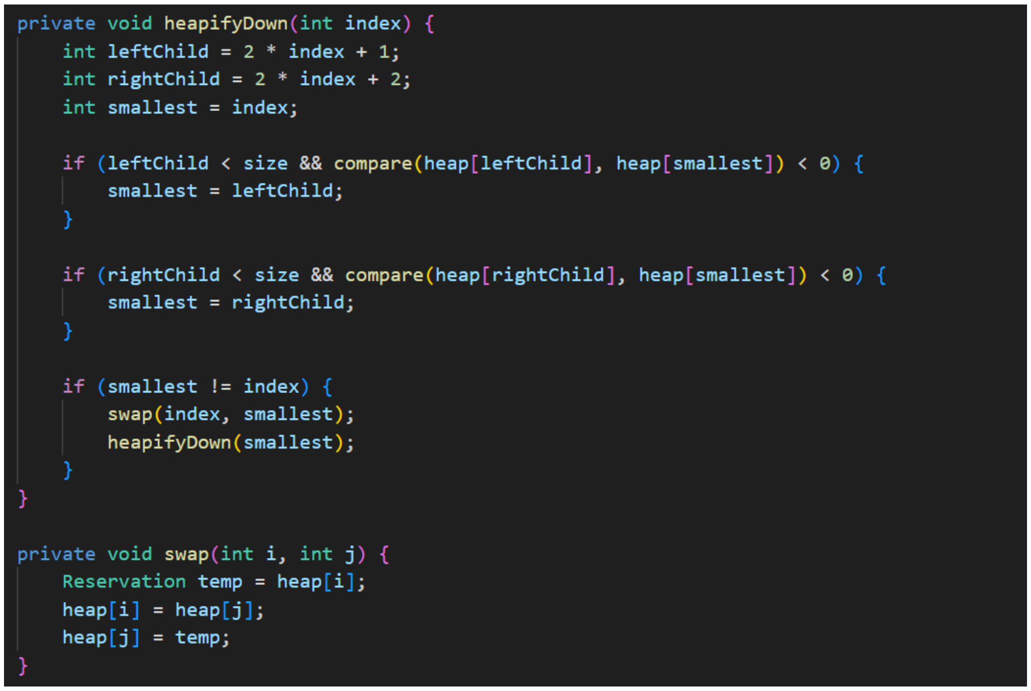


```
class MinHeap {
    private Reservation[] heap;
    private int size;
    private static final int MAX_SIZE = 20;

    public MinHeap() {
        heap = new Reservation[MAX_SIZE];
        size = 0;
    }
```

e. isEmpty(): Checks if the heap is empty by verifying if size is 0.
f. insert(int patronID, int priorityNumber, long timeOfReservation): Adds a new Reservation to the heap.
   Time Complexity : Time Complexity: O(log N), where N is the number of elements in the heap.
g. extractMin(): Removes and returns the minimum element (based on priority and reservation time) from the heap.
   Time Complexity: O(log N), where N is the number of elements in the heap.

The MinHeap maintains the order of reservations based on their priority and reservation time. When inserting a new reservation, it uses heapifyUp to maintain the heap property by moving the newly inserted reservation up the heap if needed. Similarly, heapifyDown is used after extracting the minimum element to maintain the heap property.

The compare method is responsible for comparing two reservations based on priority number and time of reservation to determine their order within the heap. The toString method generates a string representation of the heap by displaying the patron IDs of reservations in the heap array.



```java
private void heapifyDown(int index) {
    int leftChild = 2 * index + 1;
    int rightChild = 2 * index + 2;
    int smallest = index;

    if (leftChild < size && compare(heap[leftChild], heap[smallest]) < 0) {
        smallest = leftChild;
    }

    if (rightChild < size && compare(heap[rightChild], heap[smallest]) < 0) {
        smallest = rightChild;
    }

    if (smallest != index) {
        swap(index, smallest);
        heapifyDown(smallest);
    }
}

private void swap(int i, int j) {
    Reservation temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}
```

The space complexity is O(1).

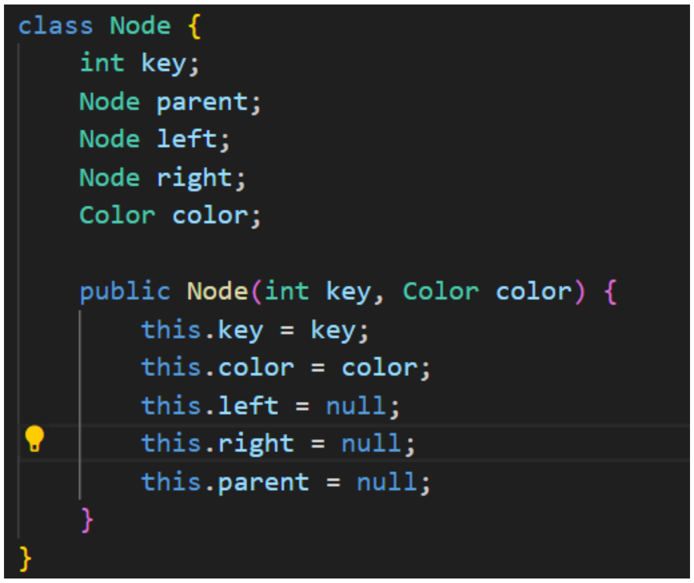
3. **NodeClass:**
   This class represents a node used in a Red-Black Tree, a self-balancing binary search tree. Let's break down the attributes and constructor:

   Attributes:

   a. int key: Represents the key value stored in the node.
   b. Node parent: Points to the parent node.
   c. Node left: Points to the left child node.
   d. Node right: Points to the right child node.
   e. Color color: Represents the color of the node in the Red-Black Tree (could be either RED or BLACK).

Constructor:

a. public Node(int key, Color color): Initializes a node with the given key value and color.
b. Sets the key value and color to the provided parameters.
c. Initializes the left, right, and parent references to null.



```
class Node {
    int key;
    Node parent;
    Node left;
    Node right;
    Color color;

    public Node(int key, Color color) {
        this.key = key;
        this.color = color;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}
```

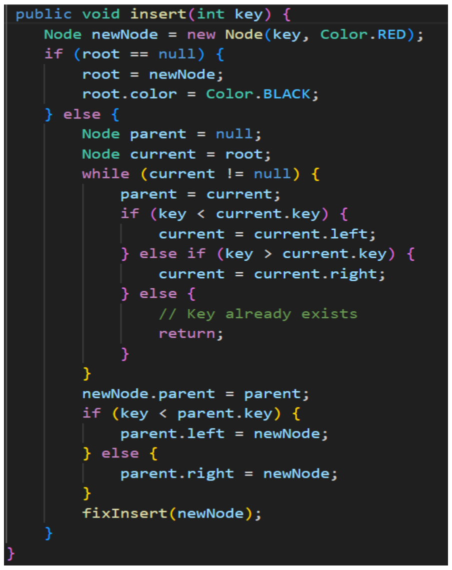
4. **RedBlackTree Class :**

```
class RedBlackTree {
  public RedBlackTree();
  public void insert(int);
  public void delete(int);
  public Node findClosestNode(int);
  public int getColorFlipCount();
  public java.lang.Object getRoot();
}
```

Manages the Red-Black Tree structure, with methods to insert, delete, and find nodes. Uses a private Node attribute root to keep track of the root of the tree. Maintains a colorFlipCount variable to track the number of color changes during rotations or deletions.

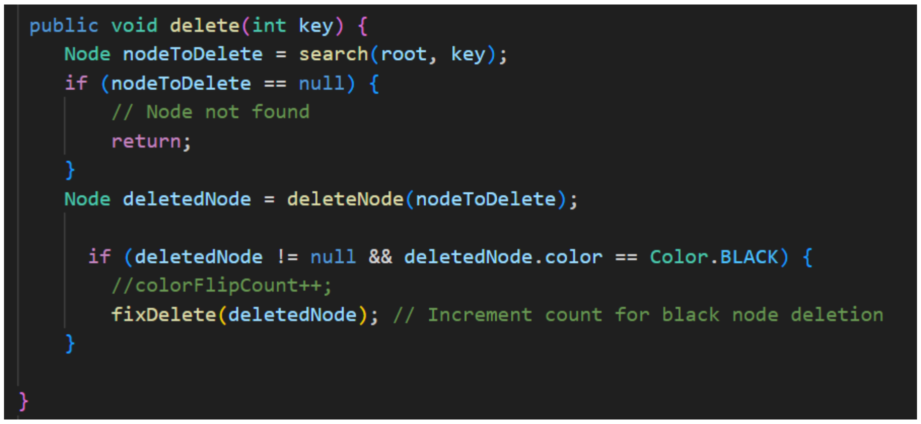
a. insert(int key): Inserts a new node with the given key into the Red-Black Tree and performs necessary rotations to maintain Red-Black Tree properties.

Time Complexity: O(log N), where N is the number of elements in the tree.



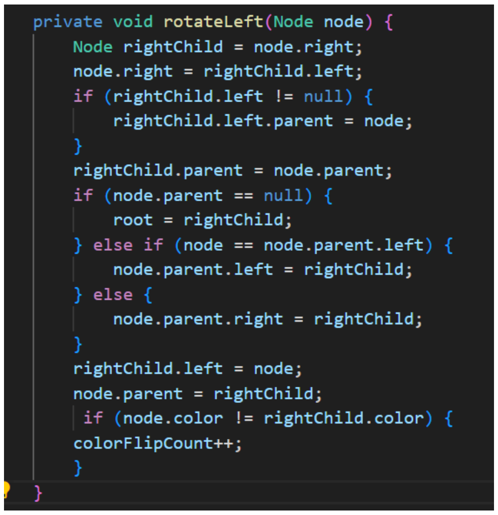
```java
public void insert(int key) {
    Node newNode = new Node(key, Color.RED);
    if (root == null) {
        root = newNode;
        root.color = Color.BLACK;
    } else {
        Node parent = null;
        Node current = root;
        while (current != null) {
            parent = current;
            if (key < current.key) {
                current = current.left;
            } else if (key > current.key) {
                current = current.right;
            } else {
                // Key already exists
                return;
            }
        }
        newNode.parent = parent;
        if (key < parent.key) {
            parent.left = newNode;
        } else {
            parent.right = newNode;
        }
        fixInsert(newNode);
    }
}
```

b. delete(int key): Deletes the node with the given key from the Red-Black Tree and adjusts the tree structure while maintaining the Red-Black properties.
Time Complexity: O(log N), where N is the number of elements in the tree.

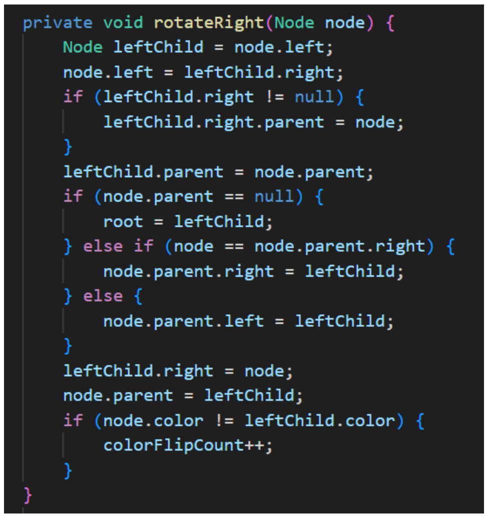


```java
public void delete(int key) {
    Node nodeToDelete = search(root, key);
    if (nodeToDelete == null) {
        // Node not found
        return;
    }
    Node deletedNode = deleteNode(nodeToDelete);

      if (deletedNode != null && deletedNode.color == Color.BLACK) {
        //colorFlipCount++;
        fixDelete(deletedNode); // Increment count for black node deletion
    }

}
```

c. fixInsert(Node node) and fixDelete(Node node): These methods handle the adjustments needed to maintain the Red-Black Tree properties after insertion or deletion, which might cause violations.
d. rotateLeft(Node node) and rotateRight(Node node): Perform left and right rotations, respectively, to maintain the Red-Black Tree balance during insertion and deletion operations.



```java
private void rotateLeft(Node node) {
    Node rightChild = node.right;
    node.right = rightChild.left;
    if (rightChild.left != null) {
        rightChild.left.parent = node;
    }
    rightChild.parent = node.parent;
    if (node.parent == null) {
        root = rightChild;
    } else if (node == node.parent.left) {
        node.parent.left = rightChild;
    } else {
        node.parent.right = rightChild;
    }
    rightChild.left = node;
    node.parent = rightChild;
     if (node.color != rightChild.color) {
    colorFlipCount++;
    }
}
```



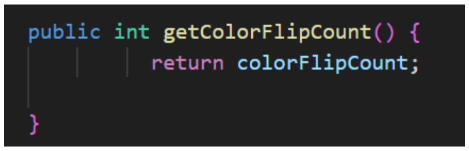
```java
private void rotateRight(Node node) {
    Node leftChild = node.left;
    node.left = leftChild.right;
    if (leftChild.right != null) {
        leftChild.right.parent = node;
    }
    leftChild.parent = node.parent;
    if (node.parent == null) {
        root = leftChild;
    } else if (node == node.parent.right) {
        node.parent.right = leftChild;
    } else {
        node.parent.left = leftChild;
    }
    leftChild.right = node;
    node.parent = leftChild;
    if (node.color != leftChild.color) {
        colorFlipCount++;
    }
}
```

e. findClosestNode(int targetID) and findClosestNode(Node root, int targetID):

   Locates the node in the tree closest to the specified target key.

   Time Complexity: O(log N), where N is the number of elements in the tree.

f. getColorFlipCount():Returns the count of color flips that occurred during tree modifications.



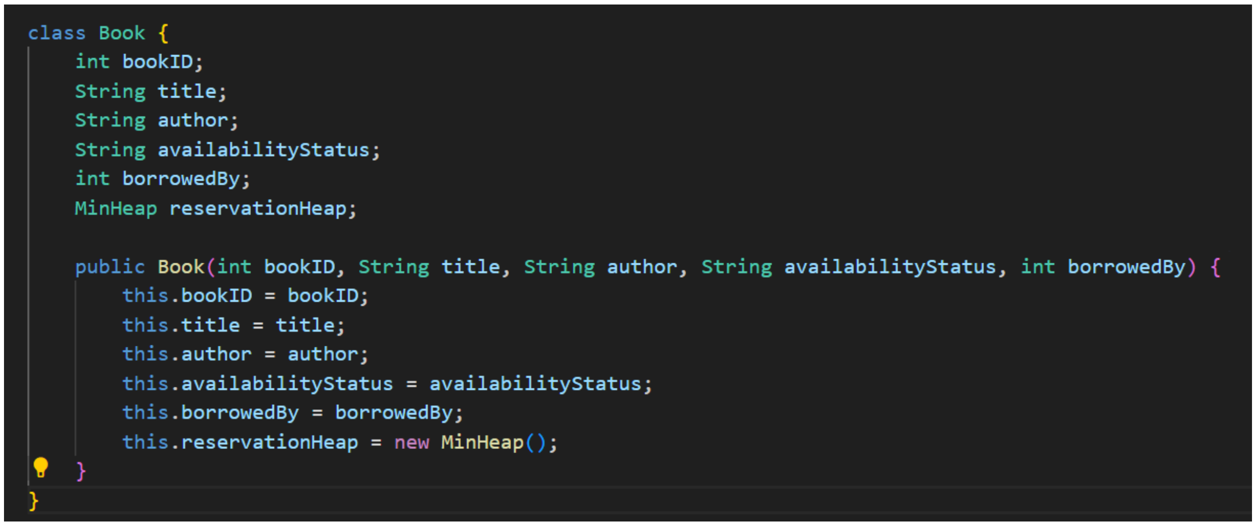
```java
public int getColorFlipCount() {
            return colorFlipCount;

}
```

The RedBlackTree class maintains the core functionalities for insertion, deletion, and node retrieval in a Red-Black Tree while adhering to the necessary rules to ensure a balanced structure. It utilizes helper methods like search, successor, and findClosestNode for navigation and search operations. The tree adjustments are performed through fixInsert and fixDelete methods to satisfy the Red-Black Tree properties, ensuring logarithmic time complexity for insertion, deletion, and search operations.

The space complexity is O(1).

5. **Book Class:**
   The Book class serves as a blueprint for individual books in the library, storing essential information about each book such as its unique ID, title, author, availability status, borrower details, and a reservation heap.
   The availabilityStatus attribute helps track whether a book is available for borrowing, already borrowed, or any other relevant status.

The borrowedBy attribute stores the ID of the patron who has borrowed the book. If the book is available, this might be set to a default value, indicating it's not currently borrowed.
The reservationHeap allows for managing reservations made by patrons who want to borrow the book when it becomes available. The MinHeap data structure is used, presumably to prioritize reservations based on certain criteria (like priority number or reservation time).



```java
class Book {
    int bookID;
    String title;
    String author;
    String availabilityStatus;
    int borrowedBy;
    MinHeap reservationHeap;

    public Book(int bookID, String title, String author, String availabilityStatus, int borrowedBy) {
        this.bookID = bookID;
        this.title = title;
        this.author = author;
        this.availabilityStatus = availabilityStatus;
        this.borrowedBy = borrowedBy;
        this.reservationHeap = new MinHeap();
    }
}
```

This class has a space complexity of O(1).

6. GatorLib Class

This GatorLib class seems to manage a library system by incorporating various functionalities using a Red-Black Tree for book management along with other data structures like HashMap and MinHeap.
class GatorLib {
 public GatorLib();
 public int findClosestBook(int, java.io.BufferedWriter) throws java.io.IOException;
 public void processOperation(java.lang.String);
 public void executeOperationsFromFile(java.lang.String);
 public void processOperation(java.lang.String, java.io.BufferedWriter) throws java.io.IOException;
}

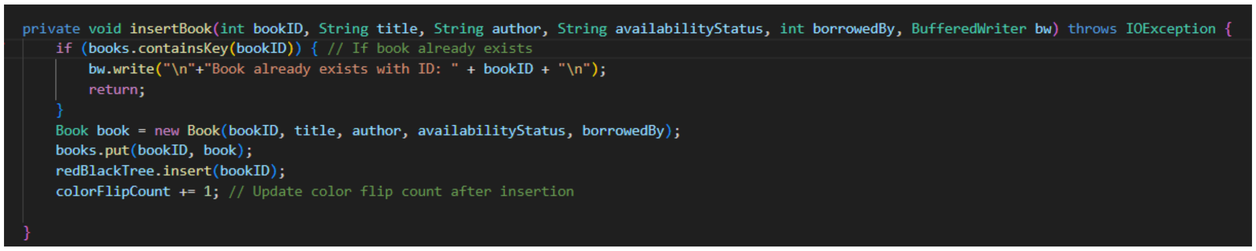
Attributes:

a) redBlackTree (RedBlackTree): Utilizes a Red-Black Tree data structure to manage the book IDs for efficient retrieval and maintenance.

b) books (Map<Integer, Book>): Stores book information with book IDs as keys and corresponding Book objects as values.
c) colorFlipCount (int):Tracks the number of color flips in the Red-Black Tree.
d) Inner Class Node: Represents the nodes used within the Red-Black Tree. Holds an integer key and attributes for left, right pointers, and color.
e) Constructor: Initializes the Red-Black Tree (redBlackTree), the book storage (books), and sets the colorFlipCount to zero.

Methods:

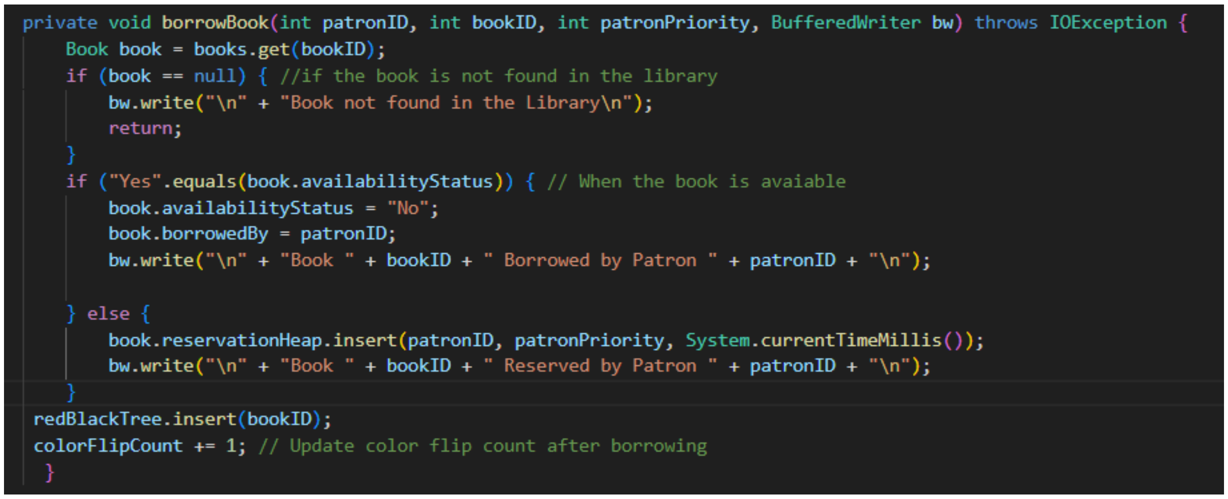
a) insertBook(int bookID, String title, String author, String availabilityStatus, int borrowedBy, BufferedWriter bw):Inserts a new book into the library along with its details. Updates the Red-Black Tree with the book ID and increments the color flip count.



```java
private void insertBook(int bookID, String title, String author, String availabilityStatus, int borrowedBy, BufferedWriter bw) throws IOException {
    if (books.containsKey(bookID)) { // If book already exists
        bw.write("\n"+"Book already exists with ID: " + bookID + "\n");
        return;
    }
    Book book = new Book(bookID, title, author, availabilityStatus, borrowedBy);
    books.put(bookID, book);
    redBlackTree.insert(bookID);
    colorFlipCount += 1; // Update color flip count after insertion

}
```

Time Complexity: O(log N), due to the Red-Black Tree insertion.

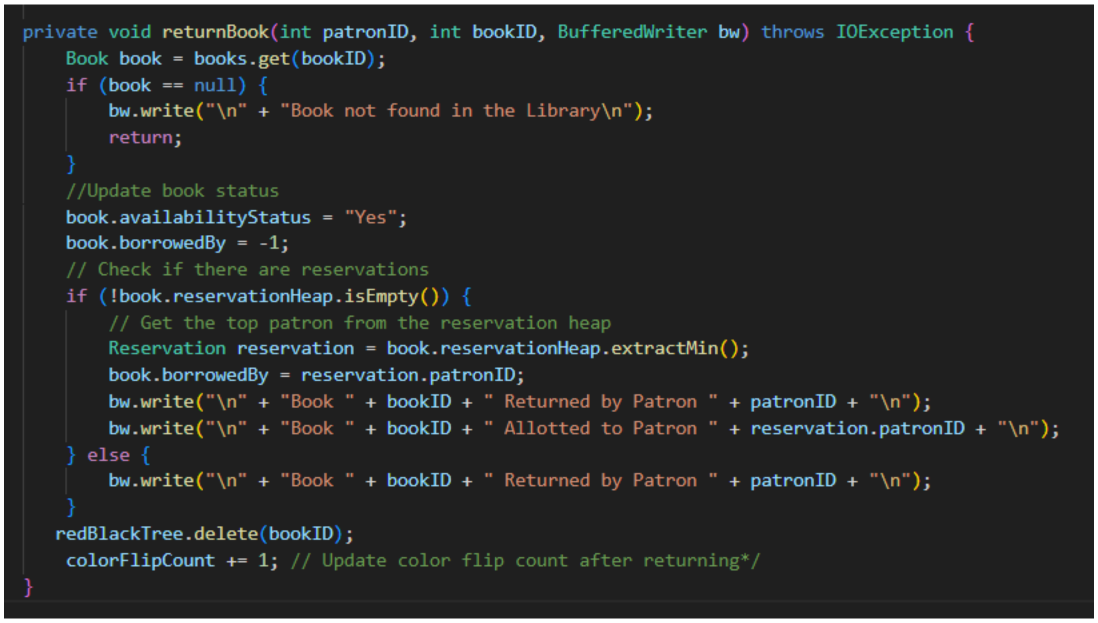
b) borrowBook(int patronID, int bookID, int patronPriority, BufferedWriter bw):Handles the borrowing process for a book: Updates book availability status or adds a reservation to the book's reservation heap if it's not available. Updates the Red-Black Tree and increments the color flip count.



```java
private void borrowBook(int patronID, int bookID, int patronPriority, BufferedWriter bw) throws IOException {
    Book book = books.get(bookID);
    if (book == null) { //if the book is not found in the library
        bw.write("\n" + "Book not found in the Library\n");
        return;
    }
    if ("Yes".equals(book.availabilityStatus)) { // When the book is avaiable
        book.availabilityStatus = "No";
        book.borrowedBy = patronID;
        bw.write("\n" + "Book " + bookID + " Borrowed by Patron " + patronID + "\n");

    } else {
        book.reservationHeap.insert(patronID, patronPriority, System.currentTimeMillis());
        bw.write("\n" + "Book " + bookID + " Reserved by Patron " + patronID + "\n");
    }
 redBlackTree.insert(bookID);
 colorFlipCount += 1; // Update color flip count after borrowing
  }
```

Time Complexity: O(log N), due to the Red-Black Tree insertion.

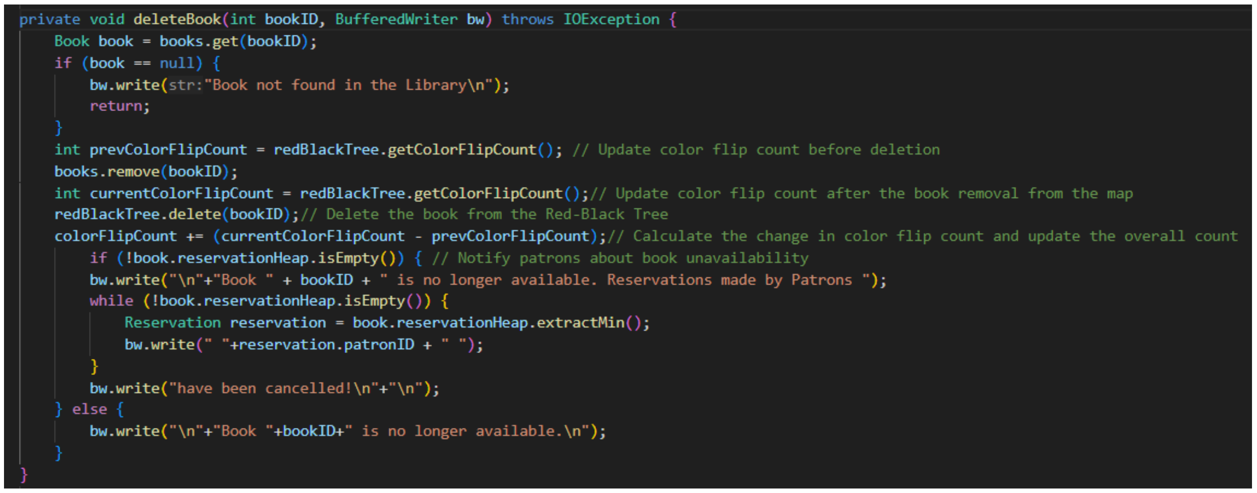
c) returnBook(int patronID, int bookID, BufferedWriter bw):Manages the returning process of a book. Updates book status, assigns the book to the next reserved patron if available. Adjusts the Red-Black Tree and color flip count accordingly.



```java
private void returnBook(int patronID, int bookID, BufferedWriter bw) throws IOException {
    Book book = books.get(bookID);
    if (book == null) {
        bw.write("\n" + "Book not found in the Library\n");
        return;
    }
    //Update book status
    book.availabilityStatus = "Yes";
    book.borrowedBy = -1;
    // Check if there are reservations
    if (!book.reservationHeap.isEmpty()) {
        // Get the top patron from the reservation heap
        Reservation reservation = book.reservationHeap.extractMin();
        book.borrowedBy = reservation.patronID;
        bw.write("\n" + "Book " + bookID + " Returned by Patron " + patronID + "\n");
        bw.write("\n" + "Book " + bookID + " Allotted to Patron " + reservation.patronID + "\n");
    } else {
        bw.write("\n" + "Book " + bookID + " Returned by Patron " + patronID + "\n");
    }
   redBlackTree.delete(bookID);
    colorFlipCount += 1; // Update color flip count after returning*/
}
```

Time Complexity: O(log N), due to the Red-Black Tree deletion.

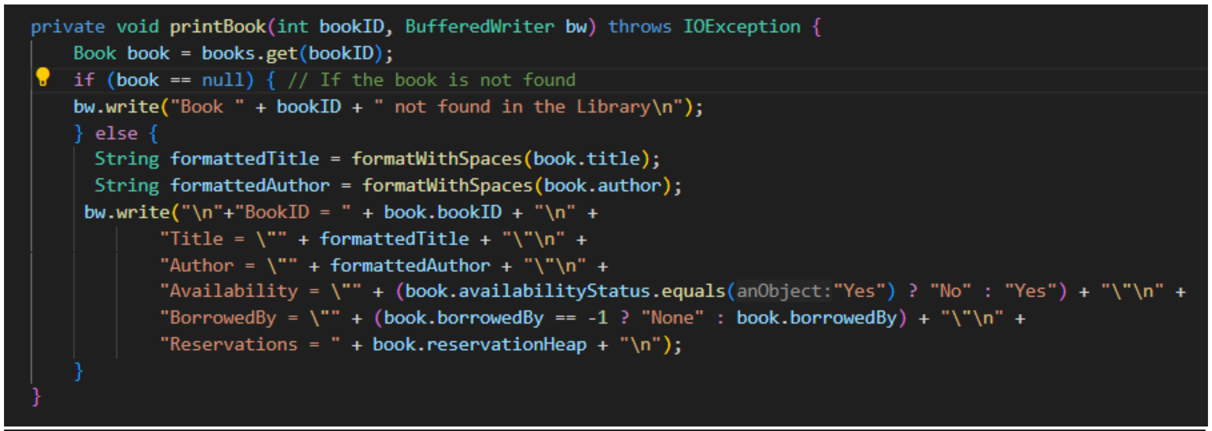
d) deleteBook(int bookID, BufferedWriter bw): Deletes a book from the library, updating the Red-Black Tree and color flip count. Cancels reservations and notifies patrons if applicable.



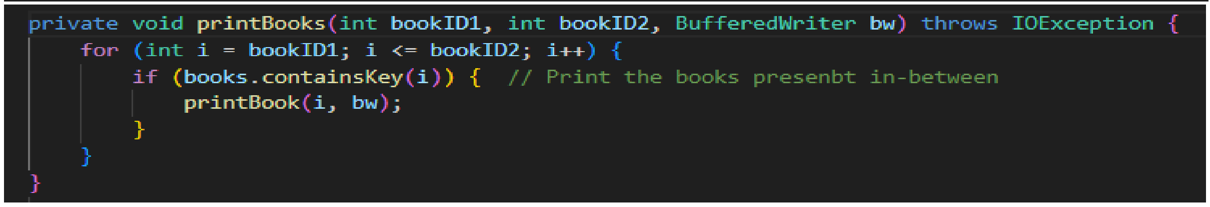
```java
private void deleteBook(int bookID, BufferedWriter bw) throws IOException {
    Book book = books.get(bookID);
    if (book == null) {
        bw.write(str:"Book not found in the Library\n");
        return;
    }
    int prevColorFlipCount = redBlackTree.getColorFlipCount(); // Update color flip count before deletion
    books.remove(bookID);
    int currentColorFlipCount = redBlackTree.getColorFlipCount();// Update color flip count after the book removal from the map
    redBlackTree.delete(bookID);// Delete the book from the Red-Black Tree
    colorFlipCount += (currentColorFlipCount - prevColorFlipCount);// Calculate the change in color flip count and update the overall count
        if (!book.reservationHeap.isEmpty()) { // Notify patrons about book unavailability
        bw.write("\n"+"Book " + bookID + " is no longer available. Reservations made by Patrons ");
        while (!book.reservationHeap.isEmpty()) {
            Reservation reservation = book.reservationHeap.extractMin();
            bw.write(" "+reservation.patronID + " ");
        }
        bw.write("have been cancelled!\n"+"\n");
    } else {
        bw.write("\n"+"Book "+bookID+" is no longer available.\n");
    }
}
```

Time Complexity: O(log N), due to the Red-Black Tree deletion.

e) printBook(int bookID, BufferedWriter bw) and printBooks(int bookID1, int bookID2, BufferedWriter bw): Outputs book details or a range of book details to a BufferedWriter.



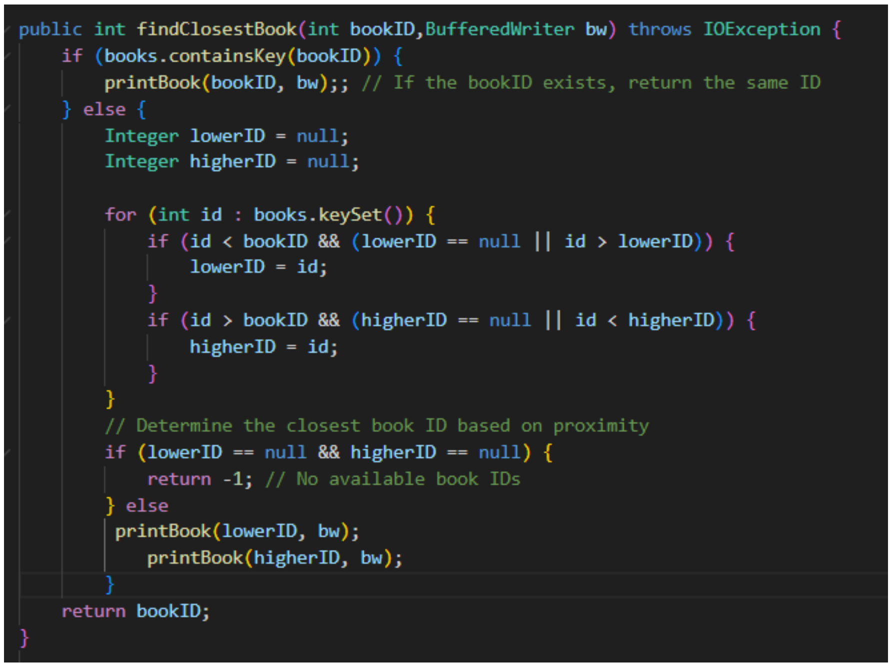
```java
private void printBook(int bookID, BufferedWriter bw) throws IOException {
    Book book = books.get(bookID);
    if (book == null) { // If the book is not found
    bw.write("Book " + bookID + " not found in the Library\n");
    } else {
      String formattedTitle = formatWithSpaces(book.title);
      String formattedAuthor = formatWithSpaces(book.author);
     bw.write("\n"+"BookID = " + book.bookID + "\n" +
            "Title = \"" + formattedTitle + "\"\n" +
            "Author = \"" + formattedAuthor + "\"\n" +
            "Availability = \"" + (book.availabilityStatus.equals(anObject:"Yes") ? "No" : "Yes") + "\"\n" +
            "BorrowedBy = \"" + (book.borrowedBy == -1 ? "None" : book.borrowedBy) + "\"\n" +
            "Reservations = " + book.reservationHeap + "\n");
    }
}
```



```java
private void printBooks(int bookID1, int bookID2, BufferedWriter bw) throws IOException {
    for (int i = bookID1; i <= bookID2; i++) {
        if (books.containsKey(i)) {  // Print the books presenbt in-between
            printBook(i, bw);
        }
    }
}
```

Time Complexity of printbook(…): O(1), accessing book details from the HashMap.
Time Complexity of printbooks(…): O(K), where K is the number of books between bookID1 and bookID2.

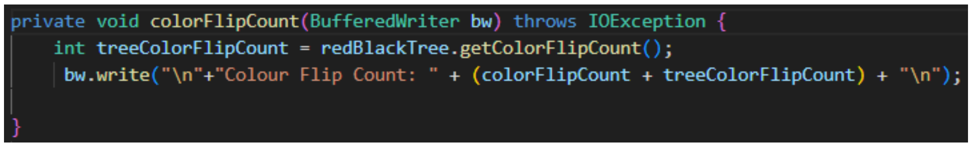
f) findClosestBook(int bookID, BufferedWriter bw):Searches for the closest book ID based on proximity and prints its details.



```java
public int findClosestBook(int bookID,BufferedWriter bw) throws IOException {
    if (books.containsKey(bookID)) {
        printBook(bookID, bw);; // If the bookID exists, return the same ID
    } else {
        Integer lowerID = null;
        Integer higherID = null;

        for (int id : books.keySet()) {
            if (id < bookID && (lowerID == null || id > lowerID)) {
                lowerID = id;
            }
            if (id > bookID && (higherID == null || id < higherID)) {
                higherID = id;
            }
        }
        // Determine the closest book ID based on proximity
        if (lowerID == null && higherID == null) {
            return -1; // No available book IDs
        } else
         printBook(lowerID, bw);
            printBook(higherID, bw);
        }
    return bookID;
}
```

Time Complexity: O(log N), due to finding the closest book in the Red-Black Tree.

f) colorFlipCount(BufferedWriter bw):Outputs the total color flip count from the Red-Black Tree and book deletions.



```java
private void colorFlipCount(BufferedWriter bw) throws IOException {
    int treeColorFlipCount = redBlackTree.getColorFlipCount();
     bw.write("\n"+"Colour Flip Count: " + (colorFlipCount + treeColorFlipCount) + "\n");

}
```

ColorFlipCount:

The colorFlipCount variable is incremented whenever a color change occurs during RBT insertion, deletion, or balancing to track these modifications.

1. In the RBT's fixInsert and fixDelete methods, the code increments colorFlipCount whenever a color change (color flip) is made to restore or maintain the RBT properties and color changes occur after identifying and addressing violations of RBT properties.
2. These increments are done at specific points when nodes' colors are changed during otations or rebalancing to ensure adherence to RBT rules.
3. The colorFlipCount incrementations are directly tied to cases where the RBT structure requires a color change to maintain its properties.
4. After rotations (left or right), colors of nodes are adjusted to maintain the RBT rules, and colorFlipCount is incremented accordingly.
5. Insertion Operation: colorFlipCount is incremented in the insertBook method of GatorLib whenever a new book is inserted into the Red-Black Tree.
6. Borrowing Operation: In the borrowBook method, colorFlipCount is incremented after inserting a book into the Red-Black Tree.
7. Returning Operation: When a book is returned in the returnBook method, colorFlipCount is incremented after the book is deleted from the Red-Black Tree.
8. Deletion Operation: In the deleteBook method, colorFlipCount is updated based on the color flip count before and after the book deletion from the Red-Black Tree.

Given these operations, the colorFlipCount is incremented at the appropriate points in the code in response to Red-Black Tree modifications (insertion, deletion). These increments are done to keep track of the color flips specifically within the Red-Black Tree operations and ensure the adherence to RBT properties after each modification.

Time Complexity: O(1), as it calculates the color flip count directly.

Manages book insertions, borrowing, returns, deletions, and searching for book details based on IDs. Utilizes Red-Black Tree for efficient book ID management and reservation heaps within Book objects. Keeps track of color flips in the Red-Black Tree and provides functionality to display this count and the functions are being called using switch command.

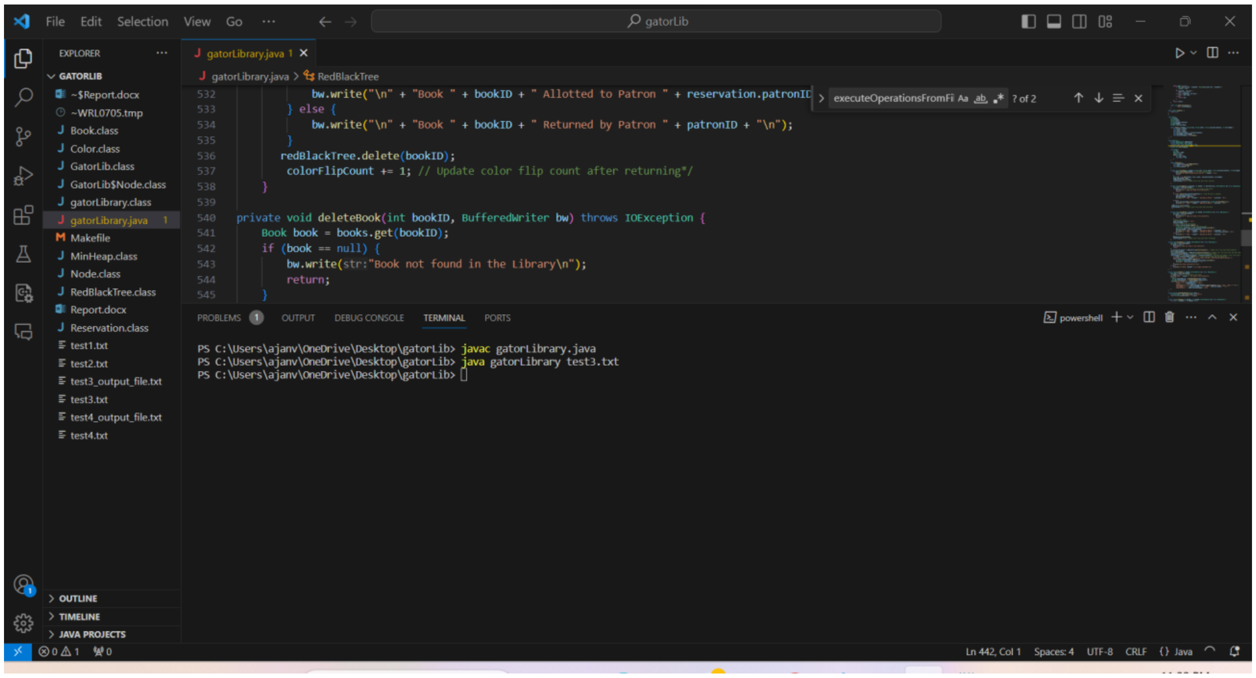
GatorLib contributes O(1) space for each instance.

Overall Time Complexities and Space Complexities :

The operations involving Red-Black Tree (insert, delete, findClosestNode) have a time complexity of O(log N) where N is the number of elements in the tree. Other operations involving MinHeap (insert, extractMin) also have a time complexity of O(log N) where N is the number of elements in the heap. Accessing book details (printBook) from the HashMap is O(1) and printing a range of books (printBooks) is O(K) where K is the number of books in the range. The complexity of finding the closest book (findClosestBook) is O(log N) due to traversal in the Red-Black Tree.

Overall, most of the classes and their members seem to have constant space complexities or space complexities that are determined by the number of instances created or the number of operations performed. The space complexity increases based on the number of books and nodes in the Red-Black tree. The space complexity of the entire program is contingent on the number of books stored in the library and the operations executed.

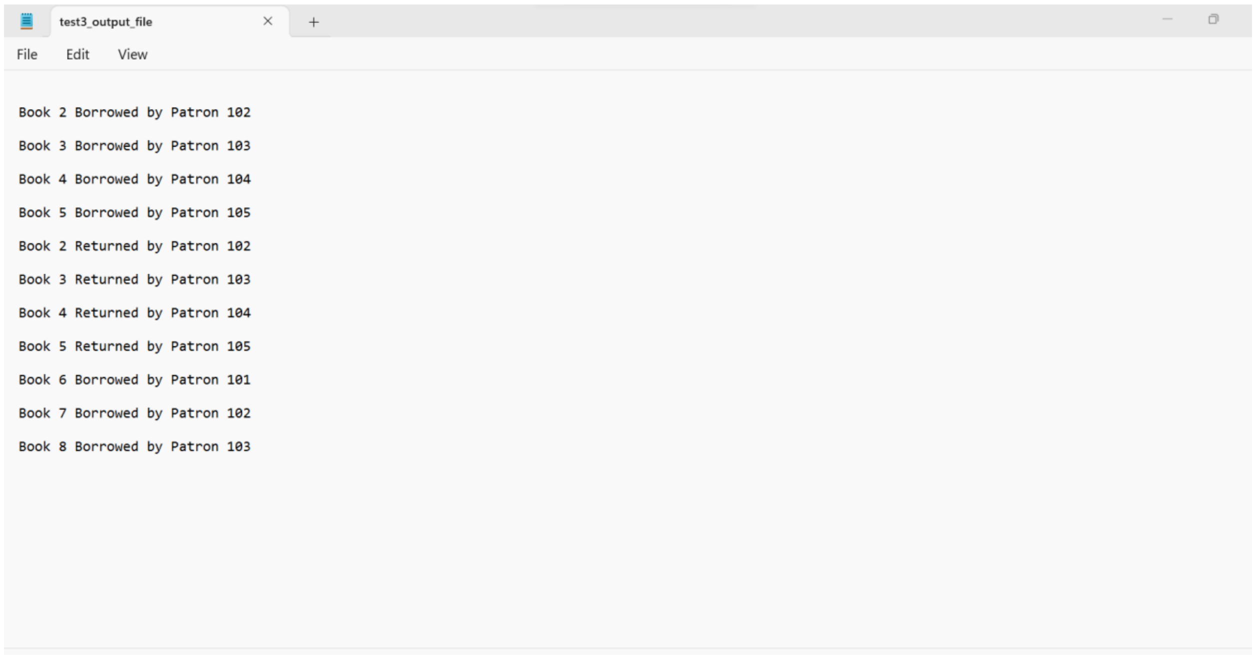
Output:



```
            bw.write("\n" + "Book " + bookID + " Allotted to Patron " + reservation.patronID
        } else {
            bw.write("\n" + "Book " + bookID + " Returned by Patron " + patronID + "\n");
        }
      redBlackTree.delete(bookID);
        colorFlipCount += 1; // Update color flip count after returning*/
    }

private void deleteBook(int bookID, BufferedWriter bw) throws IOException {
    Book book = books.get(bookID);
    if (book == null) {
        bw.write(str:"Book not found in the Library\n");
        return;
    }
```

```
PS C:\Users\ajanv\OneDrive\Desktop\gatorLib> javac gatorLibrary.java
PS C:\Users\ajanv\OneDrive\Desktop\gatorLib> java gatorLibrary test3.txt
PS C:\Users\ajanv\OneDrive\Desktop\gatorLib> 
```

test4_outout.txt:



```
Book 2 Borrowed by Patron 102
Book 3 Borrowed by Patron 103
Book 4 Borrowed by Patron 104
Book 5 Borrowed by Patron 105
Book 2 Returned by Patron 102
Book 3 Returned by Patron 103
Book 4 Returned by Patron 104
Book 5 Returned by Patron 105
Book 6 Borrowed by Patron 101
Book 7 Borrowed by Patron 102
Book 8 Borrowed by Patron 103
```