Fall 2023 COP 5536 Advanced Data Structures
Programming Project - GatorLibrary Management System

Name: Rishabh Srivastav **UFID**: 7659 9488 **Email**: rsrivastav@ufl.edu

Project description:

GatorLibrary is an imaginary library that requires a software system to effectively oversee its books, patrons, and borrowing operations. In order to build such a system, a Red-Black Tree data structure is used to ensure efficient book management using book ID and using a min-priority binary heap mechanism to manage book reservations based on patron IDs and their priority. The code for the application has been written in JAVA and the instructions to run the code and description of classes used, major functions, and their prototypes have been included in this report.

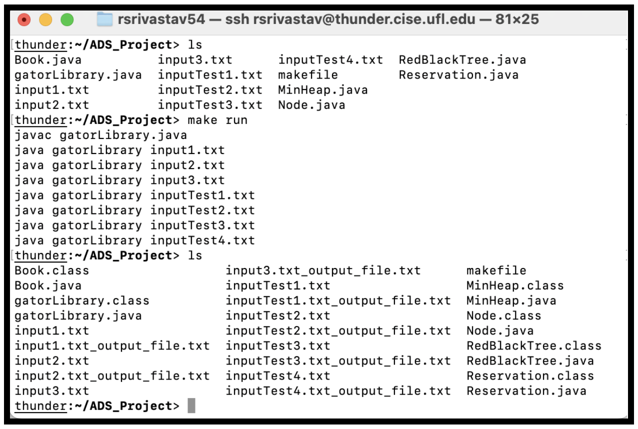
Instructions to run the code:

1. Unpack the project file using any unzip software like Winrar or 7zip.
2. The project folder consists of the following files:

- **i.** gatorLibrary.java
- **ii.** Book.java
- **iii.** Reservation.java
- **iv.** MinHeap.java
- **v.** Node.java
- **vi.** RedBlackTree.java
- **vii.** Makefile
- **viii.** Input files - (Test cases in project PDF - Input1.txt, Input2.txt, Input3.txt, Additional test cases - InputTest1.txt, InputTest2.txt, InputTest3.txt, InputTest4.txt)

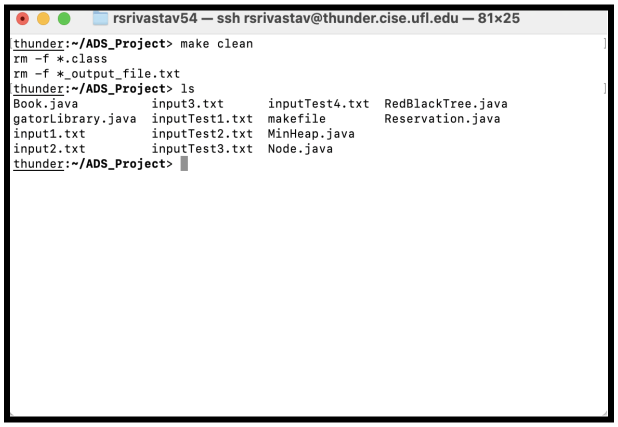
3. Once inside the project folder, run the command '**make**' to execute the make file.

- **i.** This will run '**javac gatorLibrary.java**' that compiles all associated java files and **'java gatorLibrary input.txt'** to produce the output file '**input.txt_output_file.txt**'.
- **ii.** After this you can run the command **'make run'** to automatically pass all input.txt files (3 in the project description and 4 provided as additional test cases) and generate the required output.txt files
- **iii.** You can open all the generated output.txt files, for example, output file for '**input2.txt**' will be '**input2.txt_output_file.txt**' etc.
- **iv.** Execute the command **'make clean'** to remove all compiled .class files and remove all output.txt files to start afresh.



rsrivastav54 — ssh rsrivastav@thunder.cise.ufl.edu — 81×25

```
thunder:~/ADS_Project> ls
Book.java          input3.txt       inputTest4.txt  RedBlackTree.java
gatorLibrary.java  inputTest1.txt   makefile        Reservation.java
input1.txt         inputTest2.txt   MinHeap.java
input2.txt         inputTest3.txt   Node.java
thunder:~/ADS_Project> make run
javac gatorLibrary.java
java gatorLibrary input1.txt
java gatorLibrary input2.txt
java gatorLibrary input3.txt
java gatorLibrary inputTest1.txt
java gatorLibrary inputTest2.txt
java gatorLibrary inputTest3.txt
java gatorLibrary inputTest4.txt
thunder:~/ADS_Project> ls
Book.class                 input3.txt_output_file.txt      makefile
Book.java                  inputTest1.txt                  MinHeap.class
gatorLibrary.class         inputTest1.txt_output_file.txt  MinHeap.java
gatorLibrary.java          inputTest2.txt                  Node.class
input1.txt                 inputTest2.txt_output_file.txt  Node.java
input1.txt_output_file.txt inputTest3.txt                  RedBlackTree.class
input2.txt                 inputTest3.txt_output_file.txt  RedBlackTree.java
input2.txt_output_file.txt inputTest4.txt                  Reservation.class
input3.txt                 inputTest4.txt_output_file.txt  Reservation.java
thunder:~/ADS_Project> 
```



rsrivastav54 — ssh rsrivastav@thunder.cise.ufl.edu — 81×25

```
thunder:~/ADS_Project> make clean
rm -f *.class
rm -f *_output_file.txt
thunder:~/ADS_Project> ls
Book.java          input3.txt       inputTest4.txt  RedBlackTree.java
gatorLibrary.java  inputTest1.txt   makefile        Reservation.java
input1.txt         inputTest2.txt   MinHeap.java
input2.txt         inputTest3.txt   Node.java
thunder:~/ADS_Project> 
```

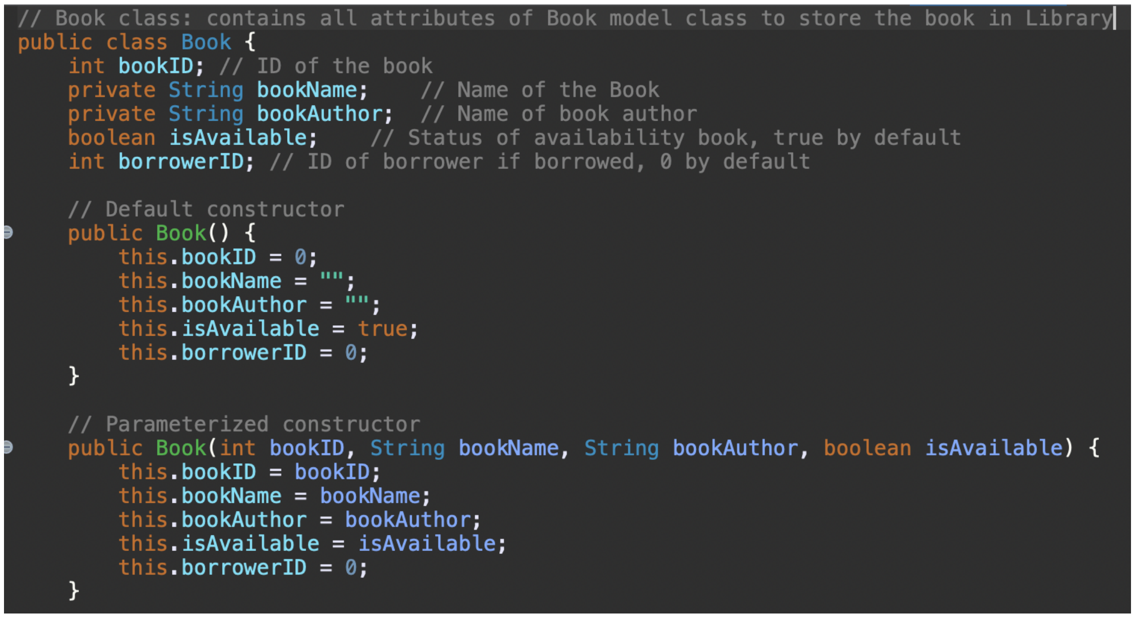
Application structure:

1. gatorLibrary.java (Execution begins here):

i. This is the main file from where the program execution begins and reads in input from the text file.
ii. This code encapsulates the gatorLibrary class, which oversees the library functionalities (insert a book, borrow a book, delete a book, find a book in range etc.) and engages with other classes.
iii. The **'main'** method in the class reads input from a file, validates if the number of arguments given are sufficient or not and then passes the input text to the function **'processCommand'** to process each operation from the input text file. It also creates the output text file which returns the results of the given operations.

2. Book.java:

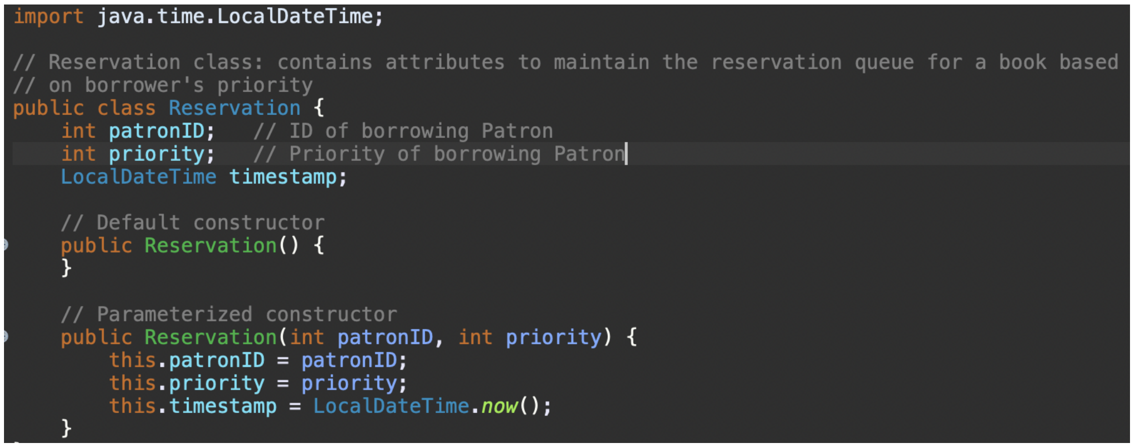
i. This class consists of the **'Book'** class, depicting individual books within the library.
ii. Each book object has attributes like book ID, name of book, author, its availability and any borrower ID as well if it is borrowed by any patron.
iii. The class facilitates the modeling and administration of data related to books within the library.



```java
// Book class: contains all attributes of Book model class to store the book in Library
public class Book {
    int bookID; // ID of the book
    private String bookName;    // Name of the Book
    private String bookAuthor;  // Name of book author
    boolean isAvailable;    // Status of availability book, true by default
    int borrowerID; // ID of borrower if borrowed, 0 by default

    // Default constructor
    public Book() {
        this.bookID = 0;
        this.bookName = "";
        this.bookAuthor = "";
        this.isAvailable = true;
        this.borrowerID = 0;
    }

    // Parameterized constructor
    public Book(int bookID, String bookName, String bookAuthor, boolean isAvailable) {
        this.bookID = bookID;
        this.bookName = bookName;
        this.bookAuthor = bookAuthor;
        this.isAvailable = isAvailable;
        this.borrowerID = 0;
    }
```

3. Reservation.java:

i. This file encompasses the **'Reservation'** class, which serves as a representation of reservations initiated by patrons for borrowing books.
ii. It consists of attributes like patron ID, their priority. Additionally, it includes a timestamp, indicating the moment when the reservation was created.



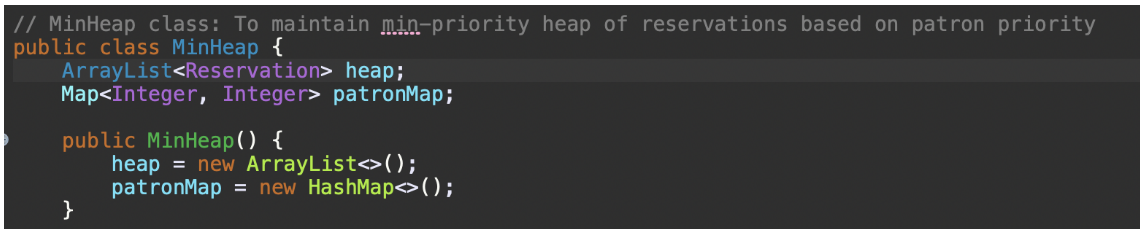
```java
import java.time.LocalDateTime;

// Reservation class: contains attributes to maintain the reservation queue for a book based
// on borrower's priority
public class Reservation {
    int patronID;   // ID of borrowing Patron
    int priority;   // Priority of borrowing Patron
    LocalDateTime timestamp;

    // Default constructor
    public Reservation() {
    }

    // Parameterized constructor
    public Reservation(int patronID, int priority) {
        this.patronID = patronID;
        this.priority = priority;
        this.timestamp = LocalDateTime.now();
    }
```

4. MinHeap.java (Implementation of a priority queue using a binary min-heap):

i. Within this document, you'll find the Min Heap implementation, serving as a priority queue for overseeing reservations. This Min Heap guarantees that the reservation possessing the highest priority or in case priority is not given then earliest timestamp is consistently positioned at the root.

ii. A list of 'Reservation' class is used within the MinHeap class to denote individual reservations along with a patronMap hashmap to keep mapping of patron ID with their associated priority.

iii. The document encompasses functions dedicated to inserting reservations, deleting reservations, and preserving the heap property.



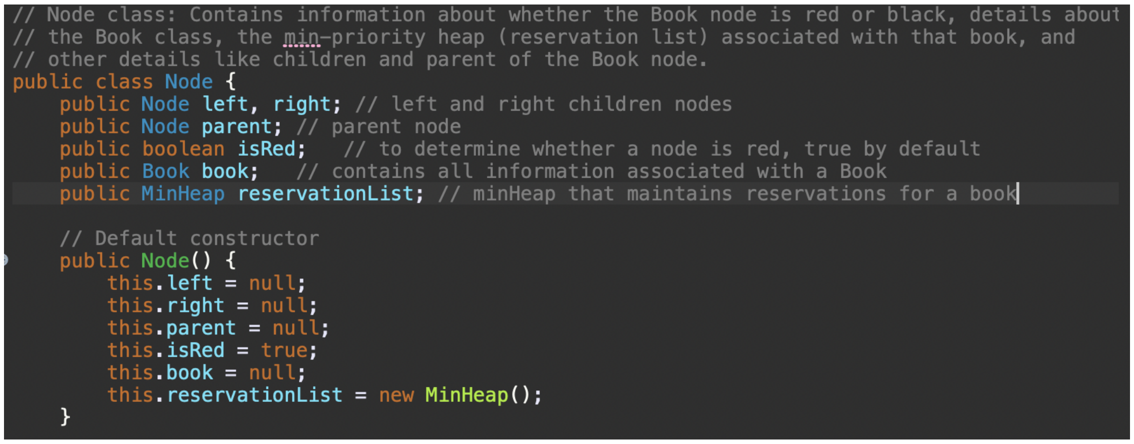
```
// MinHeap class: To maintain min-priority heap of reservations based on patron priority
public class MinHeap {
    ArrayList<Reservation> heap;
    Map<Integer, Integer> patronMap;

    public MinHeap() {
        heap = new ArrayList<>();
        patronMap = new HashMap<>();
    }
```

5. Node.java:

i. This class encompasses the **'Node'** class, responsible for embodying book class nodes in the Red-Black Tree.

ii. Every node carries details pertaining to a book, including the book ID, book name, author name, availability, and borrower ID.

iii. The Node class is designed with references to its left and right children, parent, and indications regarding whether it is a red or black node.



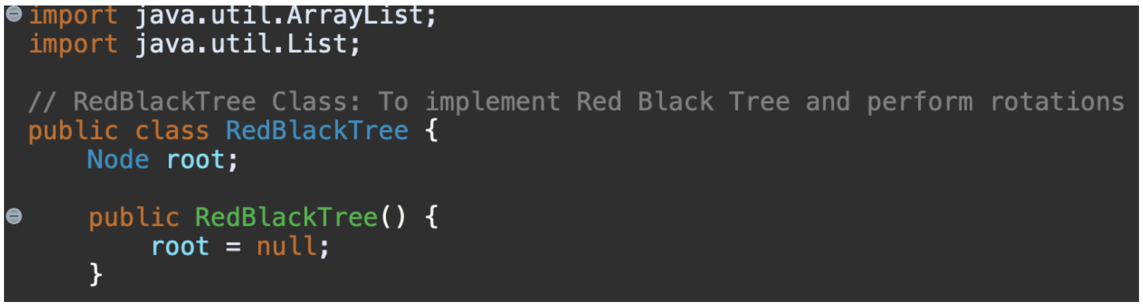
```
// Node class: Contains information about whether the Book node is red or black, details about
// the Book class, the min-priority heap (reservation list) associated with that book, and
// other details like children and parent of the Book node.
public class Node {
    public Node left, right; // left and right children nodes
    public Node parent; // parent node
    public boolean isRed;    // to determine whether a node is red, true by default
    public Book book;    // contains all information associated with a Book
    public MinHeap reservationList; // minHeap that maintains reservations for a book

    // Default constructor
    public Node() {
        this.left = null;
        this.right = null;
        this.parent = null;
        this.isRed = true;
        this.book = null;
        this.reservationList = new MinHeap();
    }
```

6. RedBlackTree.java (Red-Black Tree Implementation):

i. This class presents the implementation of a Red-Black Tree, a binary search tree that self-balances. The Red-Black Tree serves as an efficient organizational structure for searching books in the library by their book IDs.

ii. The file incorporates methods for inserting, deleting, searching, and traversing nodes, along with left and right rotation operations within the tree.

iii. The file also contains the **'Node'** class, which represents individual book nodes in the Red-Black Tree.



```
import java.util.ArrayList;
import java.util.List;

// RedBlackTree Class: To implement Red Black Tree and perform rotations
public class RedBlackTree {
    Node root;

    public RedBlackTree() {
        root = null;
    }
```

<u>Function prototypes:</u>

The major functions implemented in the application are explained here along with their complexity analysis:

1. printBook(bookId):

i. This function is tasked with displaying details regarding the book identified by the specified bookID.
ii. The function searches the bookID in the RedBlackTree class and if it is found, writes the details to the output file.
iii. The time complexity is logarithmic, expressed as O(log N), as it involves traversing the height of the tree to locate the book with the specified bookID.

2. printBooks((bookID1, bookID2):

i. This function accepts two integer parameters, bookID1 and bookID2, which denote the range of book IDs for which information should be printed.
ii. It begins by checking if bookID1 is greater than bookID2. If this condition is true, a "Range is invalid" message is printed, indicating that the provided range is not valid.
iii. If the range is valid, the function invokes the printBooks method of the RedBlackTree class, passing the specified book ID range (bookID1 and bookID2) as parameters.
iv. All books withing the range are added to an arrayList and by using the above printBook() function, each individual book is printed.
v. Linear time complexity, denoted as O(N), is incurred due to storing all books in an arrayList and subsequently traversing them within the specified range.

3. insertBook(bookID, bookName, authorName, availability)

i. This function is equipped with four parameters:
=> bookID: An integer that serves as a unique identifier for the book.
=> bookName: A string denoting the name of the book.
=> authorName: A string specifying the name of the author of the book.
=> availability: A string indicating whether the book is available or not, formatted as either "Yes" or "No."
ii. It conducts a check through the RedBlackTree.search method to verify if a book with the provided bookID already exists in the library.
iii. In the event that a book with the same bookID is discovered, a message is printed, notifying that the bookID already exists.
iv. If there is no existing book with the given bookID, the function proceeds to instantiate a new Node, representing the book.
v. A new Book object is generated with the supplied information and assigned to the Node.
vi. The overall color flip count is updated based on the outcome of the Red-Black Tree insertion, executed by invoking the RedBlackTree.insertBook() method.
vii. The process of inserting into a red-black tree requires O(log N) time, and subsequent actions for balancing and color correction can also take O(log N) in the worst-case scenario.

4. borrowBook(patronID, bookID, patronPriority)

i. It accepts three parameters:
=> patronID: an integer denoting the unique identifier for the patron.
=> bookID: an integer denoting the unique identifier for the book intended for borrowing.
=> patronPriority: an integer representing the patron's priority level.
ii. The function looks for the book with the specified bookID in the Red-Black Tree using the RedBlackTree.search() method.

iii. If the book is not found, it outputs a message stating that the book is not found.
iv. If the book is presently not available:
=> It verifies whether the patron is on the reservation wait list. If true, a message is delivered denoting that the patron is already on the waitlist.
=> In the event that the customer is not already on the waitlist, the function adds them while taking their priority into account via the minHeap.addReservation() method.
=> A message verifying the customer's reservation of the book is printed.
v. If the book can be obtained, it puts out a message stating that the patron has borrowed the book, sets the availability of the book to false, and links the patron's ID to the borrower ID of the book.
vi. To initiate a book borrowing, the addition of a reservation node into the min Heap may lead to a heapify operation, resulting in a time complexity of O(log K) in the worst-case scenario.

5. returnBook(patronID, bookID):
i. This method manages the procedure for returning a book and requires 2 parameters:
=> patronID: an integer representing the unique patron ID who borrowed the book
=> bookID : an integer representing the book ID of the book to be returned
ii. If book is not found using RedBlackTree.search() method, then it is an invalid operation.
iii. If the availability of book is true or borrowerID doesn't match patronID then also it is invalid opearaion.
iv. Otherwise, make the book available, print a message stating that the borrower has returned the book and allot the book to the next person in the reservation wait list (if any).

6. deleteBook(bookID):
i. This method manages the procedure for removing a book.
ii. It requires one parameter:
=> bookID: an integer that serves as specific identification for the book that has to be removed.
iii. The RedBlackTree.search() method is used to look for the book with the given bookID in the tree.
iv. It prints an error message saying that the book cannot be found if it cannot be located in the tree.
v. In the event that the book is located in the tree:
i. A message stating that the book is no longer available is printed to the output file.
ii. In the event that reservations are made for the book that is being deleted, it prints details regarding the canceled reservations, together with the patron IDs in the proper order according to their positions in the minHeap.
vi. By utilizing the RedBlackTree.deleteBook() method to remove the book with the given bookID from the Red-Black Tree, it modifies the overall color flip count.
vii. Deleting a book requires O(log N) operations to reach the book; additionally, O(log N) steps are needed to appropriately color and re-balance the red-black tree when a node is eliminated.

7. findClosestBook(bookID):
i. This procedure is in charge of locating the books in the tree that are closest to a particular bookID and printing details about them.
ii. It required one input:
=> bookID: an integer that serves as the book's special identification.
iii. It uses the RedBlackTree.search method to determine if the specified bookID is already in the library.
iv. If the book is located, the RedBlackTree.printBook() method is used to output book details.
v. If the given bookID is not found in the tree:
a. It initializes a list of book Nodes and uses an inorder traversal to fill it with book information.
b. Using the getLeastBigger() function, it determines the index of the least largest bookID which is greater than the specified bookID.
c. It manages many scenarios to identify the most relevant books:

=> Details regarding the smallest book based on bookID is printed if the index is -1, meaning that the specified bookID is lesser than all bookIDs in the tree.
=> Details about the largest book based on bookID is printed if the index and list size are equal and the specified bookID is greater than all bookIDs in the tree.
=> IThe nearest books are found by comparing the bookIDs if the index is inside the list's bounds:
* Details about both books is printed if there is no difference.
* Details regarding the previous book are printed if it is closer.
* Details about the book later is printed if it is closer.

vi. Books are compared against one another after being copied into a list, which results in a linear time O(N) complexity.

8. colorFlipCount() (Defense regarding different number of flips):

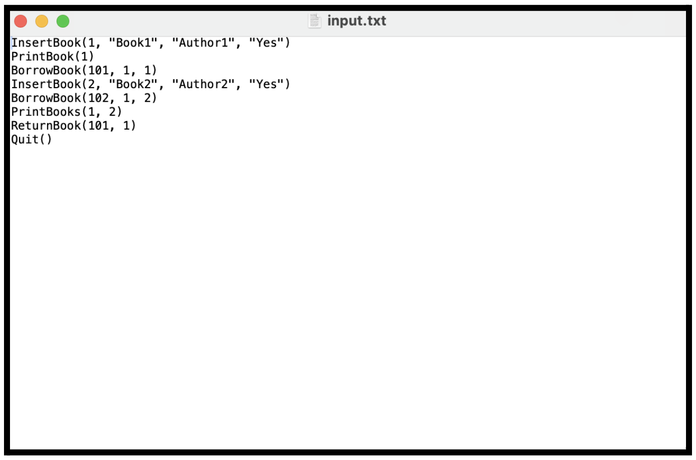
i. The Red-Black Tree's overall color flip count is printed using this method.
ii. The Flip Count is modified following each color flip, as can be seen in the aforementioned routines.
iii. With regard to the methods I've used, there are variations in the overall number of flip counts in my case. Every time a flip is carried out with this implementation, the flip count is updated. Therefore, I still interpret the flip count even if there is a flip count at the intermediary stage but no flip at the end. As a result, in both situations, my total number of recorded flips exceeds the actual count given by the test cases.
iv. The call to this function results in an O(1) access to Colorflip since it is constantly modified and stored in a variable inside the gatorLibrary main() method

Space Complexity:

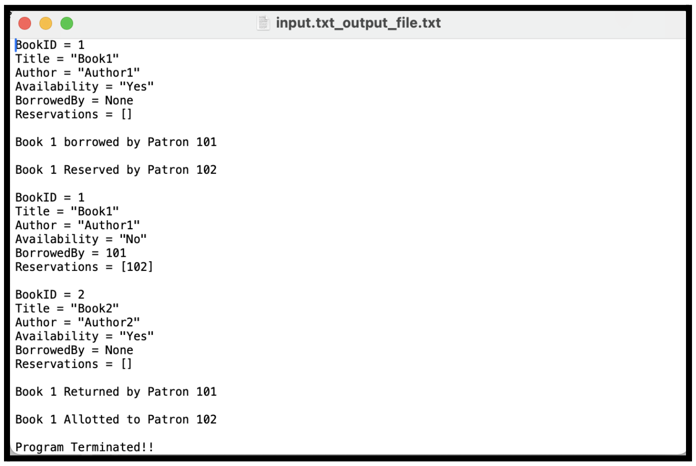
Binary Min-Heap: O(K) Considering I'm employing an arrayList execution of the min heap, it has a space of O(K), where K is number of borrowers in wait list and in the worst scenario, the list could accommodate all of the reservations at once.

Red Black Tree: O(N) as N book nodes will be present in the Red-Black tree at any given time.

Output Screenshots:


input.txt

```
InsertBook(1, "Book1", "Author1", "Yes")
PrintBook(1)
BorrowBook(101, 1, 1)
InsertBook(2, "Book2", "Author2", "Yes")
BorrowBook(102, 1, 2)
PrintBooks(1, 2)
ReturnBook(101, 1)
Quit()
```


input.txt_output_file.txt

```
BookID = 1
Title = "Book1"
Author = "Author1"
Availability = "Yes"
BorrowedBy = None
Reservations = []

Book 1 borrowed by Patron 101

Book 1 Reserved by Patron 102

BookID = 1
Title = "Book1"
Author = "Author1"
Availability = "No"
BorrowedBy = 101
Reservations = [102]

BookID = 2
Title = "Book2"
Author = "Author2"
Availability = "Yes"
BorrowedBy = None
Reservations = []

Book 1 Returned by Patron 101

Book 1 Allotted to Patron 102

Program Terminated!!
```