# COP 5536 Fall 2023 Programming Project

# REPORT


## UF ID: 12427601

## EMAIL: sa.ekbote@ufl.edu

## NAME: Sai Aakash Ekbote


Make-File for the Project

```
JAVAC = javac
JAVA = java
SOURCE = gatorLibrary.java
TARGET = gatorLibrary.class

default: $(TARGET)

$(TARGET): $(SOURCE)
    $(JAVAC) $(SOURCE)

run1: $(TARGET)
    $(JAVA) gatorLibrary inputFile1.txt

run2: $(TARGET)
    $(JAVA) gatorLibrary inputFile2.txt

run3: $(TARGET)
    $(JAVA) gatorLibrary inputFile3.txt

run4: $(TARGET)
    $(JAVA) gatorLibrary inputFile4.txt

run5: $(TARGET)
    $(JAVA) gatorLibrary inputFile5.txt

run6: $(TARGET)
    $(JAVA) gatorLibrary inputFile6.txt

run7: $(TARGET)
    $(JAVA) gatorLibrary inputFile7.txt

clean:
    rm -f *.class
```

Fig 1

Step1: run “make” command in the linux terminal

Step2: run “make run1” command in the linux terminal similarly run “make run2” till “make run7”

Step3: run “make clean” to clean up the .class executable files (run this command before running step2)

Project Structure:

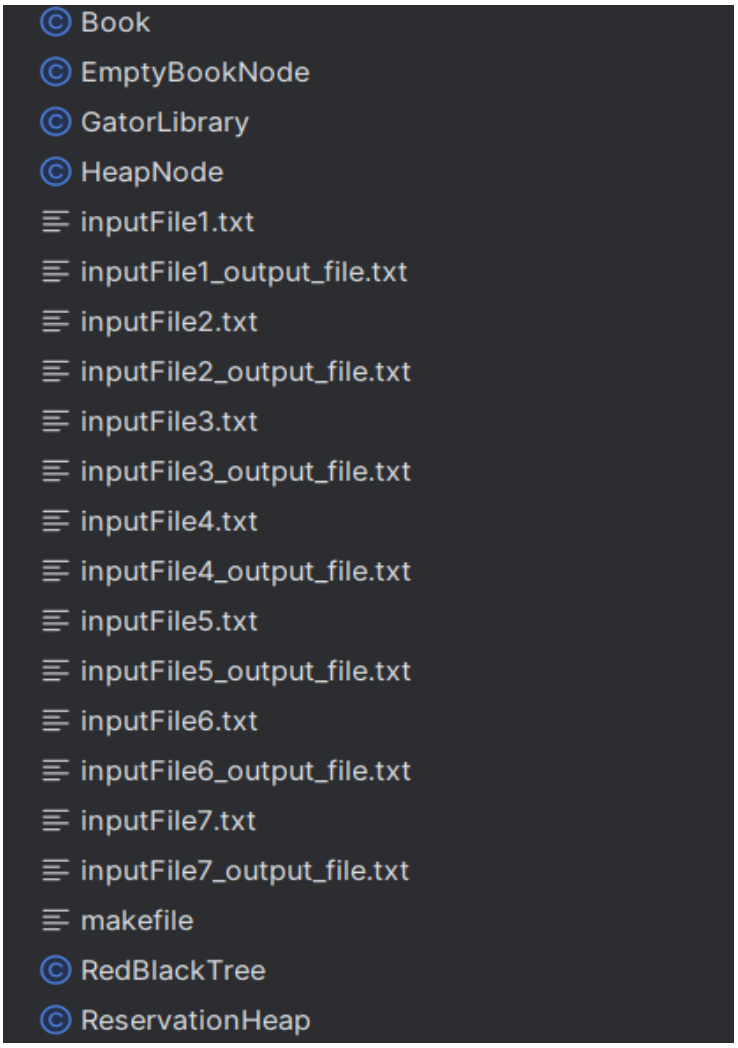
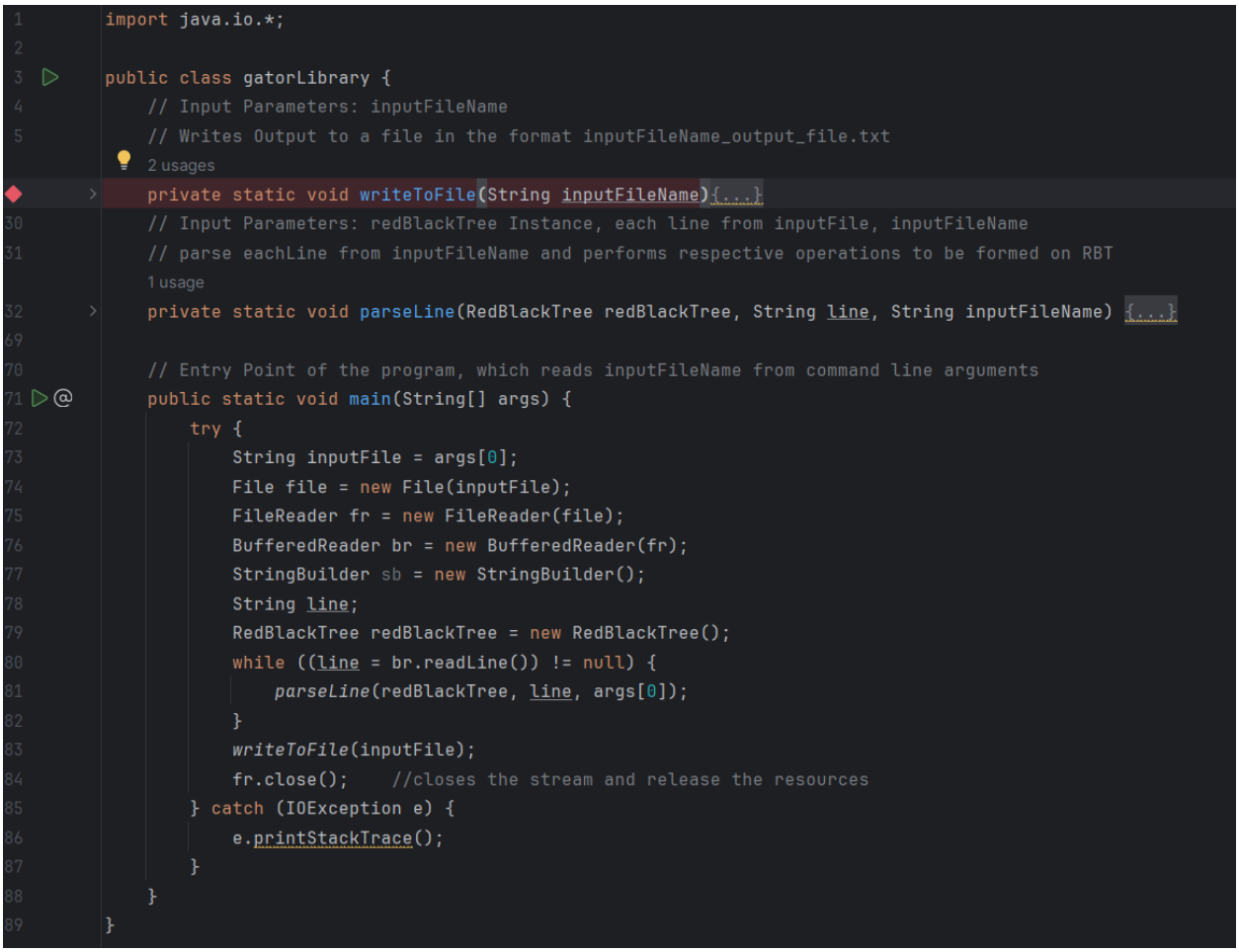



Fig 2

Main class is gatorLibrary.java

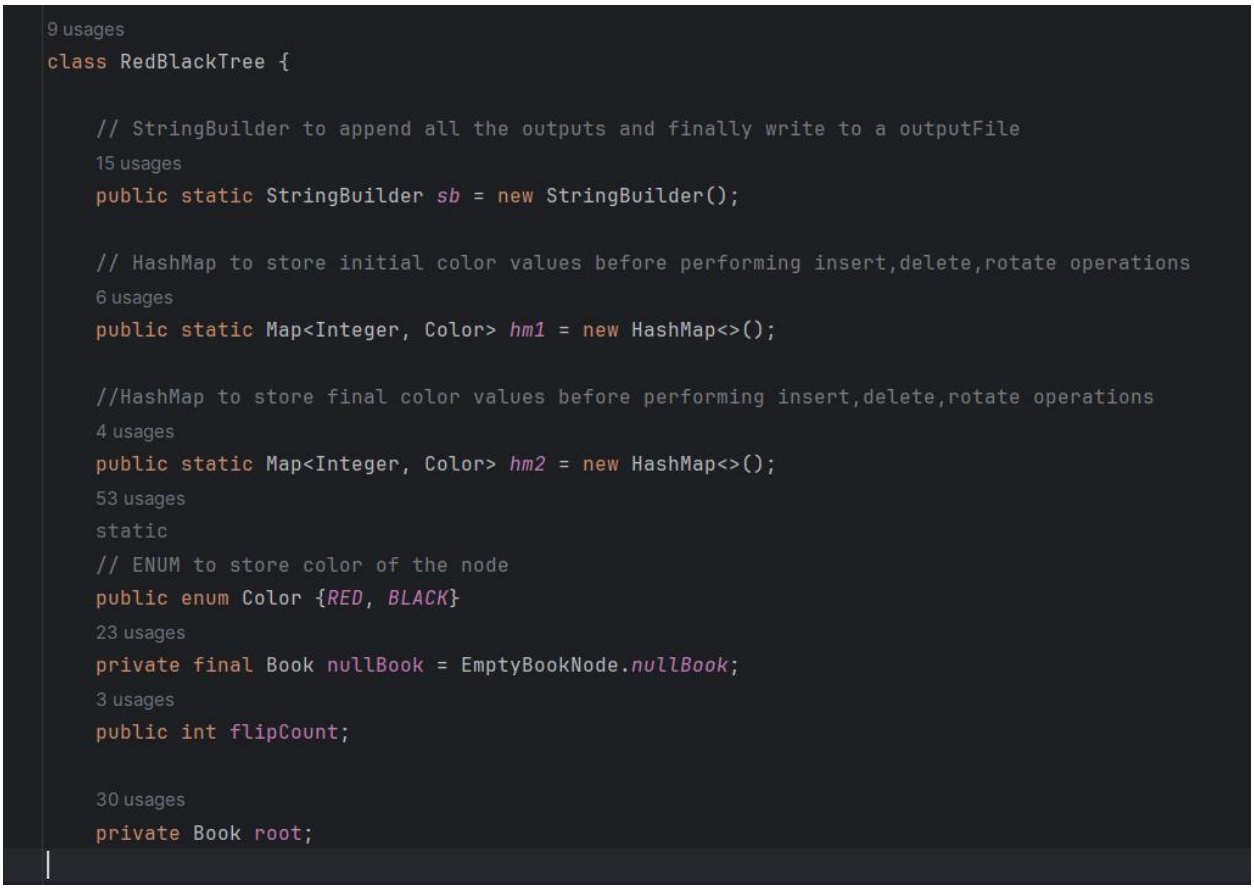


```java
import java.io.*;

public class gatorLibrary {
    // Input Parameters: inputFileName
    // Writes Output to a file in the format inputFileName_output_file.txt
    2 usages
    private static void writeToFile(String inputFileName){...}
    // Input Parameters: redBlackTree Instance, each line from inputFile, inputFileName
    // parse eachLine from inputFileName and performs respective operations to be formed on RBT
    1 usage
    private static void parseLine(RedBlackTree redBlackTree, String line, String inputFileName) {...}

    // Entry Point of the program, which reads inputFileName from command line arguments
    public static void main(String[] args) {
        try {
            String inputFile = args[0];
            File file = new File(inputFile);
            FileReader fr = new FileReader(file);
            BufferedReader br = new BufferedReader(fr);
            StringBuilder sb = new StringBuilder();
            String line;
            RedBlackTree redBlackTree = new RedBlackTree();
            while ((line = br.readLine()) != null) {
                parseLine(redBlackTree, line, args[0]);
            }
            writeToFile(inputFile);
            fr.close();    //closes the stream and release the resources
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Fig3

➔ Main method reads inputFile from command line arguments and calls parseLine function which parses each line from inputFile and executes Respective operation

➔ We call writeToFile when we reached endOfInput or encountered Quit() operation



```
9 usages
class RedBlackTree {

    // StringBuilder to append all the outputs and finally write to a outputFile
    15 usages
    public static StringBuilder sb = new StringBuilder();

    // HashMap to store initial color values before performing insert,delete,rotate operations
    6 usages
    public static Map<Integer, Color> hm1 = new HashMap<>();

    //HashMap to store final color values before performing insert,delete,rotate operations
    4 usages
    public static Map<Integer, Color> hm2 = new HashMap<>();
    53 usages
    static
    // ENUM to store color of the node
    public enum Color {RED, BLACK}
    23 usages
    private final Book nullBook = EmptyBookNode.nullBook;
    3 usages
    public int flipCount;

    30 usages
    private Book root;
```

Fig4

The above image contains structure of RedBlackTree Node

```java
public class Book {
    16 usages
    public int bookId;
    54 usages
    public RedBlackTree.Color color;
    54 usages
    public Book left, right, parent;
    2 usages
    public String bookName;
    2 usages
    public String authorName;
    7 usages
    public boolean availabilityStatus;
    7 usages
    public int borrowedBy;
    8 usages
    public ReservationHeap reservationHeap;

    @Override
    public String toString() {
        return
                "BookID = " + bookId +
                        "\nTitle = \"" + bookName +
                        "\"\nAuthor = \"" + authorName +
                        "\"\nAvailability = \"" + (availabilityStatus?"Yes":"No") +
                        "\"\nBorrowedBy = " + (borrowedBy==-1?"None":borrowedBy) +
                        "\nReservations = [" + this.reservationHeap+"]\n";
    }
```

Fig5: Book structure

```java
14 usages
public class HeapNode implements Comparable<HeapNode> {
    4 usages
    public int patronId;
    7 usages
    public int priorityNumber;
    5 usages
    public Date timeOfReservation;

    1 usage
    public HeapNode(int patronId, int priorityNumber, Date timeOfReservation) {
        this.patronId = patronId;
        this.priorityNumber = priorityNumber;
        this.timeOfReservation = timeOfReservation;
    }

    // Overriding compareTo method , comparing by priorityNumber first and breaking ties with timeOfReserva
    @Override
    public int compareTo(HeapNode other){
        if (this.priorityNumber != other.priorityNumber)
            return this.priorityNumber - other.priorityNumber;
        else
            return this.timeOfReservation.compareTo(other.timeOfReservation);
    }
```

Fig6: HeapNode structure

```java
public class ReservationHeap {
    // Set a fixed capacity for the heap. Given 20 as per the problem statement
    2 usages
    public static final int CAPACITY = 20;

    // Array to store heap elements.
    18 usages
    public HeapNode[] heap;

    // Current number of elements in the heap.
    13 usages
    public int size;

    // Constructor initializes the heap array and size.
    1 usage
    public ReservationHeap() {...}
```

Fig7: Reservation Heap Structure

## Function Prototype:

## insertBook():

```java
public void insertBook(int bookId,String bookName,String authorName,String availabilityStatus,int borrowedBy){
    Book book = new Book(bookId,bookName,authorName,availabilityStatus.equals("Yes")?true:false,borrowedBy);
    initializeHashMaps();
    if(root == nullBook){
        hm1.put(bookId, Color.BLACK);
    }
    else{
        hm1.put(bookId,Color.RED);
    }
    insert(book);
    populateHm2();
    calculateFlipCounts();
}
```

Fig8: InsertBook() structure

We pass borrowedBy as “-1” for initial case and Create a new Book to be inserted in the red black tree and pass book as argument to insert(book) .

## borrowBook():

```java
// Borrow a book from GatorLibrary
1 usage
public void borrowBook(int patronId,int bookId,int patronPriority){
    Book book=printBook(bookId);
    if(book==null)
        return;
    if(book.availabilityStatus){
        book.borrowedBy=patronId;
        book.availabilityStatus=false;
        sb.append("\nBook "+bookId+" Borrowed by Patron "+patronId +"\n");
    }  else if (alreadyReservedByPatron(patronId, book)) {
        sb.append("Book " + bookId + " Already Reserved by Patron " + patronId + "\n");
    } else {
        sb.append("\nBook " + bookId + " Reserved by Patron " + patronId+"\n");
        book.reservationHeap.addToHeap(new HeapNode(patronId, patronPriority, new Date()));
    }
}

1 usage
public boolean alreadyReservedByPatron(int patronId, Book book) {
    for (HeapNode reservation : book.reservationHeap.heap) {
        if (reservation != null && reservation.getPatronId() == patronId) {
            return true;
        }
    }
    return false;
}
```

Fig9: borrowBook() structure

- ➔ We do inorder search in the redblack tree to fetch book by patronId , and then check availability status for that book
- ➔ If book is available then we allot the book to patron who requested the borrow
- ➔ if book is not available we add the patron to the reservation heap, with the help of patronID, patronPriority and Current TimeofReservation

## DeleteBook():

```
// Delete a book from RB Tree
1 usage
public void deleteBook(int bookId) {
    Book delBook = printBook(bookId);
    if(delBook == null) {
        sb.append("\nBook "+bookId+" is no longer available."+"\n");
        return;
    }
    initializeHashMaps();
    hm1.remove(bookId);
    delete(delBook);
    populateHm2();
    calculateFlipCounts();
    if(delBook.reservationHeap.isEmpty()){
        sb.append("\nBook "+bookId+" is no longer available."+"\n");
    } else{
    sb.append("\nBook "+bookId+" is no longer available. Reservations made by Patrons "+ delBook.reservationHeap+" have been cancelled!" +"\n");
    }
}
```

Fig10: deleteBook() structure

- ➔ we first do inorder search in the redblack tree to find if the bookId requested to be deleted is present or not,
- ➔ if present we delete from redblack Tree calling delete(book) and also compute flipCounts
- ➔ if not present we display appropriate message

## findClosestBook()

```java
// findClosestBook function based on bookId
1 usage
public void findClosestBook(int targetId){
    int minDiff=Integer.MAX_VALUE;
    List<Book> listOfBooks=new ArrayList<>();
    inorder(this.root, lower: -1, upper: -1,listOfBooks, flag: false);
    List<Book> res=new ArrayList<>();
    for(Book book:listOfBooks){
        int diff=Math.abs(targetId-book.bookId);
        if(minDiff>diff){
            minDiff=diff;
            res=new ArrayList<>();
            res.add(book);
        }else if(minDiff==diff)
            res.add(book);
    }
    res.sort(new sortByBookId());
    for(Book book:res) {
       // System .out.println(book);
        sb.append(book + "\n");
    }
}

1 usage
public class sortByBookId implements Comparator<Book>{
    public int compare(Book a, Book b) { return a.bookId - b.bookId; }
}
```

Fig11: findClosestBook() structure

➔ we do inorder Traversal and get all the books in an arraylist
➔ we calculate the difference between targetBookId and for each bookId from inorder result and compute closestBooks.
➔ We then sort the result based on bookId

## PrintBook()

```java
// PrintBook based on bookId
7 usages
public Book printBook(int bookId) {
    Book temp = root;
    if(root.bookId == -1)
        return null;
    while (true) {
        if (bookId < temp.bookId) {
            if (temp.left == nullBook) {
                return null;
            } else {
                temp = temp.left;
            }
        }else if (bookId == temp.bookId) {
            return temp;
        } else {
            if (temp.right == nullBook) {
                return null;
            } else {
                temp = temp.right;
            }
        }
    }
}
```

```java
} else if (operation.equals("PrintBook")) {
    int bookId = Integer.parseInt(ar[0].trim());
    Book book = redBlackTree.printBook(bookId);
    if (book != null) {
        RedBlackTree.sb.append(book +"\n");
    }
    else {
        RedBlackTree.sb.append("Book " + bookId + " not found in the library" +"\n");
    }
```

Fig 12: printBook() structure

## printBooks():

```java
// printBooks based on bookId1 and bookId2
1 usage
public void printBooks(int bookId1,int bookId2){
    List<Book> listOfBooks=new ArrayList<>();
    inorder(this.root,bookId1,bookId2,listOfBooks, flag: true);
    for(Book book: listOfBooks)
        sb.append(book+"\n");
}
```

```java
// Inorder Traversal for printBooks
4 usages
private void inorder(Book book, int lower, int upper, List<Book> listOfBooks, boolean flag){
    if(book== nullBook)
        return;
    inorder(book.left,lower,upper,listOfBooks,flag);
    if(flag) {
        if (book.bookId >= lower && book.bookId <= upper)
            listOfBooks.add(book);
    }
    else {
        listOfBooks.add(book);
    }
    inorder(book.right,lower,upper,listOfBooks,flag);
}
```

Fig 13: printBooks() structure

colorFlipCount()

```java
// Print the colorFlipCount
1 usage
public void colorFlipCount(){
    sb.append("\nColor Flip Count : "+this.flipCount +"\n");
}
```

Fig 14: colorFlipCount() structure

## ReturnBook()

```
// Return a book to the GatorLibrary
1 usage
public void returnBook(int patronId,int bookId){
    Book book=printBook(bookId);
    if(book==null)
        return;
    if(book.borrowedBy!=patronId)
        return;
    if(book.availabilityStatus)
        return;
    book.borrowedBy=-1;
    book.availabilityStatus=true;
    sb.append("Book "+bookId+" Returned by Patron "+patronId+"\n");
    if(!book.reservationHeap.isEmpty()){
        int reservedPatronId=book.reservationHeap.poll().patronId;
        if(reservedPatronId==-1)
            return;
        book.borrowedBy=reservedPatronId;
        book.availabilityStatus=false;
        sb.append("\nBook "+bookId+" Allotted to Patron "+reservedPatronId+"\n");
    }
}
```

Fig14: ReturnBook() structure

- ➔ We do inorder search in the redblack tree and fetch the book associated with the book id
- ➔ We set borrowedBy parameter to -1 and set availability status of the book to true and print appropriate message
- ➔ We then, check if reservation heap for that book is empty or not,
- ➔ If it is Empty , we exit from the function
- ➔ Else, we allote the book to reserved Patron from the reservationHeap, and set borrowedBy to reservedPatronId and availabiltyStatus to false and print appriopriate message and exit from the function

## Quit()

```
// Terminate the program
1 usage
public void quit(){
    sb.append("\nProgram Terminated!!" +"\n");
    this.root=null;
}
```

```
// Input Parameters: redBlackTree Instance, each line from inputFile, inputFileName
// parse eachLine from inputFileName and performs respective operations to be formed on RBT
1 usage
private static void parseLine(RedBlackTree redBlackTree, String line, String inputFileName) {
    line = line.replaceAll( regex: "\"", replacement: "");
    if (line.contains("Quit")) {
        redBlackTree.quit();
        writeToFile(inputFileName);
        System.exit( status: 0);
    }
```

Fig: 15: Quit() structure

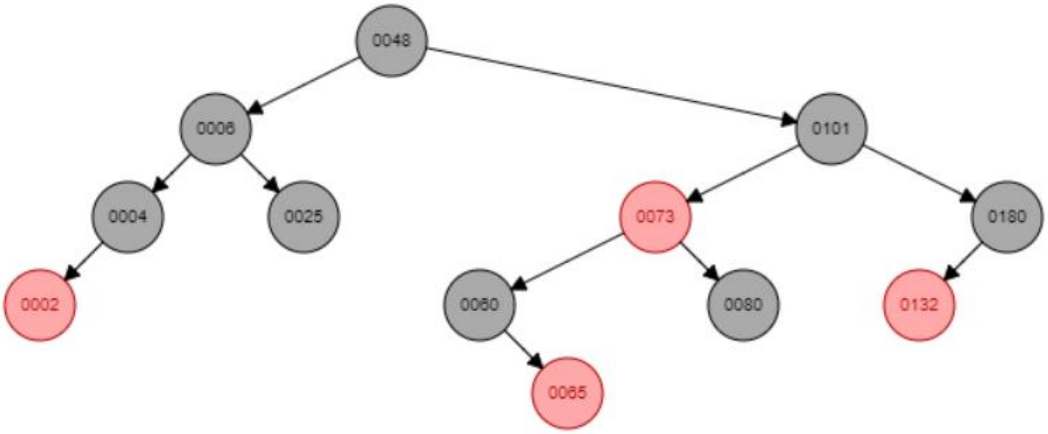
## Input and Outputs:

All inputs and Outputs are matching with sample Input and Output provided except InputFile4.txt

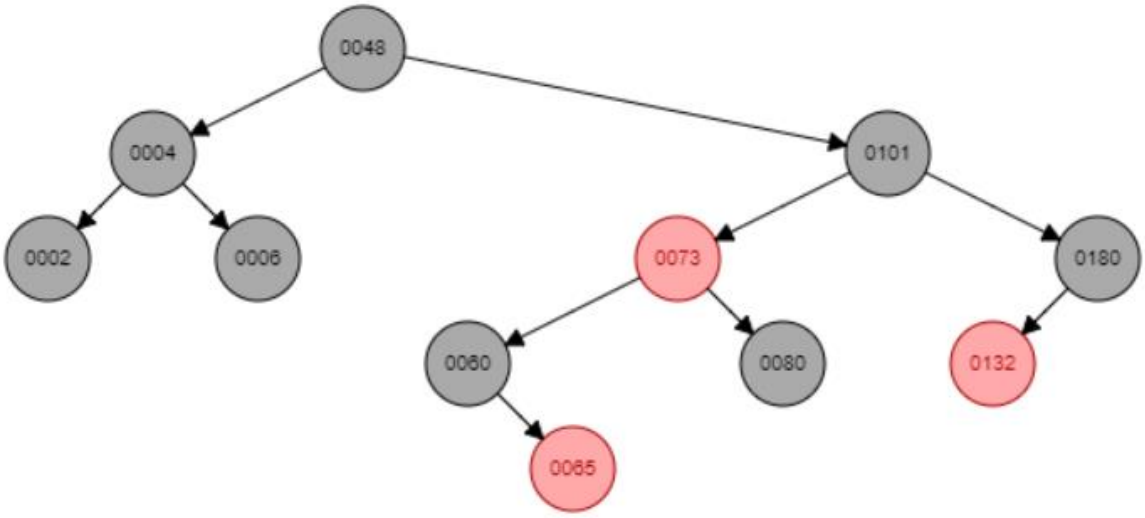
**For InputFile4.txt**

**Before DeleteBook(25) is encountered tree looks like**

**ColorFlipCount is 23**



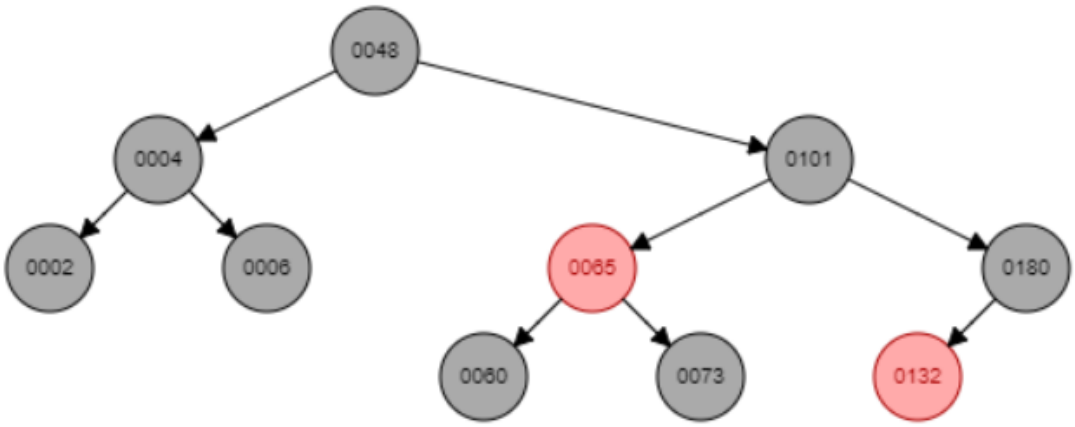

**After DeleteBook(25)**




**ColorFlipCount is 24 as red node 2 is changed to black**

**DeleteBook(80)**

**Tree Becomes:**




**Node 73 color is changed from red to black and rest of the color remains same**

**So, ColorFlipCount is 25**

**Therefore, The final ColorFlipCount is 25 in my case instead of 27 given in the sample output.**

## How to calculate FlipCount():

```
// HashMap to store initial color values before performing insert,delete,rotate operations
6 usages
public static Map<Integer, Color> hm1 = new HashMap<>();

//HashMap to store final color values before performing insert,delete,rotate operations
4 usages
public static Map<Integer, Color> hm2 = new HashMap<>();
```

Initialize two hashmaps , hm1 and hm2

```java
// Initialize hashmaps used for flipCount
2 usages
private void initializeHashMaps(){
    hm1.clear();
    hm1= new HashMap<>(hm2);
    hm2.clear();
}
```

Before each insert and delete we invoke initializeHashMaps() which copies all the key value pairs from hm2 to hm1

```java
public void insertBook(int bookId,String bookName,String authorName,String availabilityStatus,int borrowedBy){
    Book book = new Book(bookId,bookName,authorName,availabilityStatus.equals("Yes")?true:false,borrowedBy);
    initializeHashMaps();
    if(root == nullBook){
        hm1.put(bookId, Color.BLACK);
    }
    else{
        hm1.put(bookId,Color.RED);
    }
    insert(book);
    populateHm2();
    calculateFlipCounts();
}
```

We insert bookId and Color as black if root is empty or else

We insert bookId and color as red

```
2 usages
private void populateHm2(){
    inorderTraversal(root);
}

// Inorder Traversal into RB Tree
3 usages
private void inorderTraversal(Book root){
    if(root == nullBook || root == null){
        return;
    }
    inorderTraversal(root.left);
    hm2.put(root.bookId, root.color);
    inorderTraversal(root.right);
}
```

After Insert operation is perfomed, we perform inorder traversal on red black Tree and update hashmap with bookId and color of each Node

```
// calculateFlip Counts in RB Tree
2 usages
private void calculateFlipCounts(){
    for(Map.Entry<Integer, Color> entry: hm1.entrySet()){
        if(entry.getValue() != hm2.get(entry.getKey())){
            this.flipCount++;
        }
    }
}
```

We then calculate the FlipCounts comparing key,value pairs from both the hashmaps.