**COP 5536 – Advanced Data Structures**

**Fall 2023**

**Programming Project**

**GatorLibrary Management System**

**PROJECT REPORT**

**Name: Charithesh Puppireddy**

**Programming Environment:**

GatorLibrary Management System is implemented in java programming language. All .java files are in the same directory. Input file should be in the same directory as the .java files. A makefile is present whose default function is to create .class files of all java classes. Output will be written to input\_file\_name\_output\_file.txt which is present in the same directory as the input file. Input file name should be given as the first command line argument. The program can be executed in the following order.

1. make  
2. java gatorLibrary input\_file\_name.txt

**Implementation**:

To implement gatorLibrary we make use of a min heap and red-black tree (RBT). The min heap priorities book reservations for the patrons with lower priority. If two or more patrons have the same priority, then the patron with minimum timestamp is given more priority. The RBT is implemented with book id number as the search key.

1. To implement **PrintBook(bookID)** method.

To implement PrintBook method, we traverse through the RBT tree by comparing at each node, its bookID with the given id number and either traverse through the left subtree or the right subtree until we find the node with required id. If we don’t find the node with the required bookID we print *“*BookID not found in the Library*”* or else print the book details.

2. To implement **PrintBooks(bookID1, bookID2)** method.

To implement PrintBooks method between a given two book id numbers, we perform a recursive in-order traversal of the RBT tree to find and print the details of the books whose id numbers lies between the bookID1 and bookID2 (both included).

3. To implement **InsertBook(bookID, bookName, authorName, avaialabilityStatus, borrowedBy, reservationHeap)** method.

To implement insert method, we first check if the book with given id is already present in the RBT, if yes then we throw exception saying that the bookID is already present in the RBT. If the book with given ID is not present, then we initialize a book object with the given id and we assign a min heap for the book object to store reservations for that book. We then link the node with its parent node and call fixRBTAfterInsert function which then does necessary changes to the tree such that RBT properties hold.

4. To implement **BorrowBook(patronID, bookID, patronPriority)** method.

To implement BorrowBook method we first search the book with the given id. Then we check if the book is already borrowed, if yes, we add the current patron to the minheap with the given priority. If the book is not borrowed by anyone then we mark the book as borrowed by the given patron and set the availability of the book to false.

5. To implement **ReturnBook(patronID, bookID)** method.

To implement ReturnBook method we search the book with given id and get the node. We check if there are any other patrons in the minheap i.e, reservation heap. If there are patrons in the heap, then we assign the book to the next patron in the minheap which gives the patron with high priority. If there are no patrons waiting in the reservation heap, we set the book availability as true and borrowed field to 0 ( denoting no one borrowed the book).

6. To implement **DeleteBook(bookID)** method.

To implement the DeleteBook(bookID), first we search the node. If the node is not present, we do nothing, else we delete the node if it has one child and it has two children we replace it with maximum book id in left subtree of the node and delete the maximum on left subtree. After deleting the node if the node color is black then we adjust the RBT in order to hold its properties.

7. To implement **FindClosestBook(targetID)** method.

To implement FindClosestBook(targetID), first we search the node with the targetID. If the node is present, then we print the book details. If it is not present then, we find the ceil and floor of the given targetID in the RBT. After getting the ceil and floor of the given targetID, we check the difference between the ceil and target and floor and target. If the ceil is closer then we print the book with ceil as id, if floor is closer then we print the book with floor as id, if both are equal then we print both the books.

8. To implement **ColorFlipCount()** method

To implement ColorFlipCount() method, we declare a global integer variable to store the number of color flips happened in the tree. At any time where we change the color we check if the previous color and final color of the node is same or not, if it is not the same then we increment the variable. When the color flip count method is called, we just print the value in the integer variable.

The input file name is read as a command line argument and we loop across each statement in the input file, perform the required operation and write the result to the output file named as input\_file\_name\_output\_file.txt.

**Project Structure:**

The project consists of 5 class files and a make file.

1. makefile
2. gatorLibrary.java
3. GatorLibraryService.java
4. RedBlackTree.java
5. Reservation.java
6. ReservationHeap.java

**Makefile:**

The make file is used to create .class files of the above mentioned five classes. It has a default argument which complies all the classes.

CLASSES = $(wildcard \*.class)

SOURCES = $(wildcard \*.java)

default:

javac $(SOURCES)

clean:

rm -rf $(CLASSES)

rm -rf \*output\_file\*

**Class Definitions and function prototypes:**

**1. gatorLibrary.java**

This class is used to execute the program. It reads the file passed as command line argument line by line, extract the arguments and then call the respective function in the GatorLibraryService class by creating an object for the class.

* **main(String[] args)**

This method reads the lines from the file passed as command line argument and passes the line input to processCommand function.

* **processCommand(operation,arguments)**

This method takes in operation and arguments as input. Based on the operation it calls the respective functions using switch case passing arguments.

* **processInsertBook(String arguments)**

This method calls insertBook function in GatorLibraryService class.

* **processPrintBook(String arguments)**

This method calls printBook function in GatorLibraryService class.

* **processPrintBooks(String arguments)**

This method calls printBooks function in GatorLibraryService class.

* **processBorrowBook(String arguments)**

This method calls borrowBook function in GatorLibraryService class.

* **processReturn(String arguments)**

This method calls returnBook function in GatorLibraryService class.

* **processDelete(String arguments)**

This method calls deleteBook function in GatorLibraryService class.

* **processColorFlip()**

This method calls printColorFlips function in GatorLibraryService class.

* **processFindClosest(String arguments)**

This method calls findClosest function in GatorLibraryService class.

**2. GatorLibraryService.java**

This class is responsible for executing the functions that are called from the gatorLibrary class, creates an output file and writes the output to the output file. It calls the insert, delete and search functions on RedBlackTree.

**Instance variables**

* RedBlackTree redBlackTree - reference to the RedBlackTree.java class
* BufferedWriter bufferedWriter – used to write output to the file.

**Functions**

* **insertBook(int id,String name,String authorName,boolean isAvailable)**

Creates an object for the Book class and calls insertNode function in RedBlackTree.java passing the book as an argument.

* **printBook(int id)**

Calls searchNode function in RedBlackTree.java passing the id. If the node is present, then writes the book details to the file else writes “Book id not found in the Library”.

* **printBooks(int from, int to)**

calls printBooks function in RedBlackTree.java which returns the books that are present in the tree having id between from and to. After getting the list out books it prints the details of the book to the file.

* **borrowBook(int patronID, int bookID, int patronPriority)**

calls searchNode function in RedBlackTree.java. If the node is present with the bookID and it is available, sets the borrowedBy fields of the book with patronID and availability as false. If the book’s availability is false, then adds the patron to the waitlist of the book reservation heap.

* **returnBook(int patronID, int bookID)**

calls searchNode function in RedBlackTree.java. If the node is present and has reservations, then sets the book to the next patron from the reservation heap. If there are no reservations, then the book’s availability is set to true.

* **deleteBook(int id)**

gets the node from red black tree with the given id. If the node is present, then it deletes the node and writes the output to the file in corresponding output formats.

* **findClosest(int id)**

gets the node from the tree. If the node is present, then writes the book details to the output file. If the node is not present, then calls the function in red black tree to find the books with ceil(id) and floor(id) as their ids. After getting the ceil and floor id’s books it prints the book, which is the closest to the given id, If both books are in same distance then it prints both the books.

**3. RedBlackTree.java**

This class implements a red black tree with nodes having book as the data.

Instance variables

* Node root – reference to the root of the red black tree
* int noOfColorFlips – integer storing the number of color flips for all the nodes combined.

Functions

* **void rotateRight(Node node)**

Method to rotate the given node to the right. The time complexity of this operation is O(1) as it involves only constant number of pointer changes and colour changes. The space complexity is O(1).

* **void changeParentsChild(Node parent,Node oldChild,Node newChild)**

replaces parent’s old child with new child and sets the parent pointer of the newChild to the parent.

* **void rotateLeft(Node node)**

Method to rotate the given node to the left. The time complexity of this operation is O(1) as it involves only constant number of pointer changes and colour changes. The space complexity is O(1).

* **Node searchNode(int key)**

Method to check whether there exists any node in RBT with the given ride number. This function takes O(log(N)) time as it involves traversing the entire tree height in the worst case.  
The space complexity is O(1).

* **Node insertNode(Book data)**

Method to insert a new RBTNode inside the red black tree. The insert is similar to insertion in binary search tree, and the time complexity is order of height of the tree which is O(log(N)).

* **void changeColor(Node node, COLOR color)**

* **void fixRedBlackPropertiesAfterInsert(Node node)**

Method to restore the RBT properties after insertion of a new node. This operation involves colour changes and rotations, each takes constant amount of time. The overall time complexity is again order of height of the tree which is O(log(N)). The space complexity is O(1) as no extra space is used.

* **Node getUncle(Node parent)**

Method to get the uncle node of the given parent node. The time complexity of this operation is O(1) as it involves just pointer access. The space complexity is O(1).

* **void deleteNode(int key)**

Method to delete a RBTNode with the given ride number. The time complexity of this function is order of height of the tree (same as in BST) and is O(log(N)). The operation doesn’t use any extra space so the space complexity of O(1).

* **fixRedBlackPropertiesAfterDelete(Node node)**

Method to restore the RBT properties after a node is deleted. The node which violates the properties of the RBT is passed as a parameter. The time complexity of this function is O(log(N)) as it involves number of rotations and colour changes at each node(in worst case) that takes constant amount of time. No extra space is used so the space complexity is O(1).

* **List<Node> printBooks(int from,int to)**

This method is a parent function for traverseAndCollect function. It declares a list of nodes and returns it after it is populated in traverseAndCollect function.

* **void traverseAndCollect(Node node, int from, int to, List<Node> books)**

This method is used for printing nodes whose book ids ranges in between from and to both included. We perform an in-order traversal recursively in the RBT and add the nodes to the list that are in the required range.  
The time complexity of this operation is **O(log(N) + S)**. O(log(N)) to find the node with book id greater than or equal to the minimum book id and S for getting next S successors. The space complexity is O(1) as we don’t use any extra space.  
This function adds the nodes in to the books list which are in range from and to.

* **Node[] findCeilAndFloor(int id)**

This method returns the nodes with id’s as ceil(id) and floor(id). It is similar to finding ceil and floor of an element in a BST. The time complexity is log(N)

**4. Reservation.java**

This class is just a plain class representing the reservation with variables

* patronID
* priority
* timeOfReservation

It contains basic getters and setters for the variables and overrides toString() method.

**5. ReservationHeap.java**

This class implements a minimum heap internally with Reservation class as a heap node. The heap is constructed based on the priority, the node with minimum priority is at the root of the heap.

Instance variables

* Reservation[] heap – heap array to store reservations
* int size – integer storing the size of the heap.

Functions

* **int parent(int pos)**

Returns the parent index for given index which is pos/2.

* **int leftChild(int pos)**

Returns index of left child of the heap node which is at the index 2\*pos, as we are using 1 as a root index.

* **int rightChild(int pos)**

Returns index of right child of the heap node which is at the index 2\*pos+1, as we are using 1 as a root index.

* **void minHeapify(int pos)**

code defines a method **minHeapify** for maintaining the min-heap property in a binary heap data structure. It recursively adjusts the heap structure starting from the given position **pos** by comparing the node at **pos** with its children. If necessary, it swaps the node with the smaller of its children and continues the heapification process.

* **void insert(Reservation reservation)**

It first checks if the heap is at its maximum size (**MAX\_SIZE**) and returns if true. If there is space, it adds the reservation to the heap array and performs a heapify operation by comparing the inserted element with its parent, swapping them if necessary, and iteratively moving up the heap until the heap property is restored.

* **Reservation remove()**

It removes the element at the root of the heap (index FRONT), which is the smallest element in the min-heap. The removed element is stored in a variable called "popped," and the heap is adjusted using the "minHeapify" method to maintain the min-heap property. The size of the heap is updated, and the removed element is returned. Additionally, the code sets the vacated space at the end of the heap to null for cleanup.

All operations mentioned in the problem statement are implemented in **O(log(N))** complexity where N is the number of books.

The overall space complexity is **O(N\*sizeOf(reservationHeap))** which is used for storing Red black tree nodes and min Heap for each book.