University of Florida

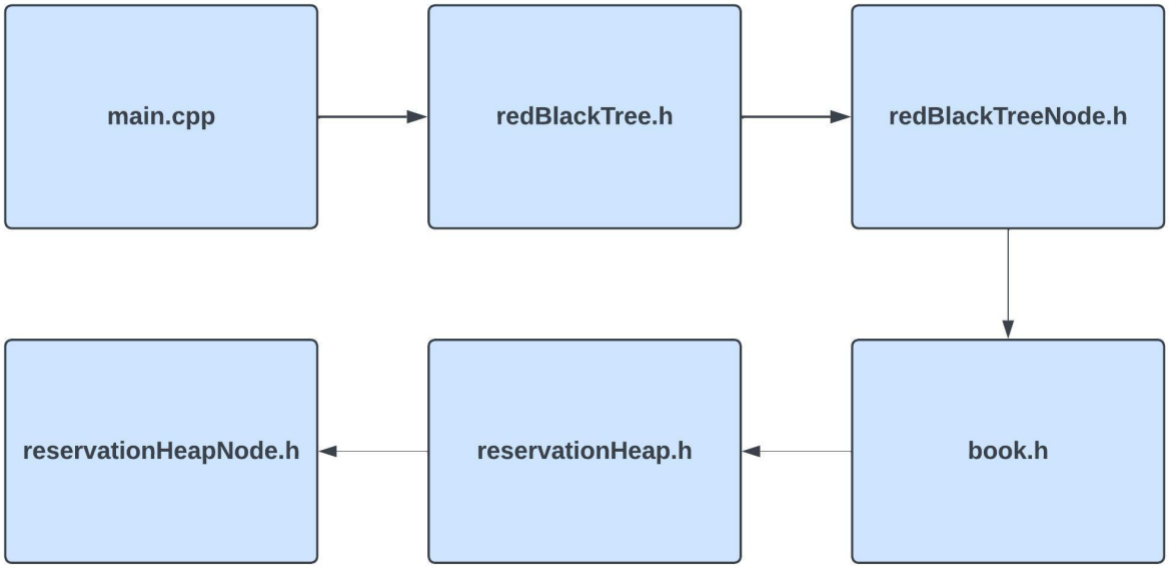
Department of Computer and
Information Science and Engineering

Programming Project

Name: Sheel Taskar
UFID: 2753-4463
Email: sheel.taskar@ufl.edu

Program Structure:

The program is divided into one main driver file and five header files. The program structure is given below.



main.cpp:

This file contains the driver code to read the input file, parse the operation and its associated parameters and call the appropriate functions for the Red-Black tree based on the input operation using an object of redBlackTree class. It also validates that the number of parameters for each operation is correct and terminates the program when 'Quit' operation is received.

redBlackTree.h:

This header file contains helper functions to implement the Red-Black Tree functionality and its associated functions to perform insertions, deletions and rotations after every operation given in the input file to maintain the Red-Black Tree property.

redBlackTreeNode.h:

Defines the node structure of the Red-Black Tree. It contains information about parent pointers, child pointers, node color and the book information stored in the node.

book.h:

This header file contains all the details of a book stored in a Red-Black Tree node such as book ID, book name, author name, availability status, borrowed by information and the reservation heap of patrons waiting for the allotment of the book.

reservationHeap.h:

This header file contains the implementation of a min heap which is used to store the patrons in an order of priority. The nodes with lower priority number and time of reservation will have a higher priority.

reservationHeapNode.h:

This header file contains the structure of the reservation heap node and helper functions to compare two reservation heap nodes based on priority and time of reservation.

Zip Contents:

- main.cpp
- redBlackTree.h
- redBlackTreeNode.h
- book.h
- reservationHeap.h
- reservationHeapNode.h
- Makefile
- Report.pdf

Usage:

Run the command 'make' to generate the 'gatorLibrary' binary. Then use the command './gatorLibrary test1.txt' to run the test case in the file 'test1.txt.'

Function Prototypes:

For each of the five files, the function prototypes are given here:

Main.cpp

```
// Function to extract parameters from a line
vector<string> extractParameters(string line);

// Function to generate an output file name based on the input file name
string getOutputFileName(string input);
```

RedBlackTree.h

```
// Print book with given bookId
void printBook(int bookId);

// Print books with bookId between bookId1 and bookId2
void printBooks(int bookId1, int bookId2);

// Perform borrow book for the given patronId, bookId, and patronPriority
void borrowBook(int patronId, int bookId, int patronPriority);

// Perform return book for the given patronId and bookId
void returnBook(int patronId, int bookId);

// Find and return the books closest to the given bookId
void findClosestBook(int bookId);

// Get the greatest bookId smaller than the given bookId
redBlackTreeNode* getGreatestOnLeft(int bookId);

// Get the smallest bookId larger than the given bookId
redBlackTreeNode* getSmallestOnRight(int bookId);

// Helper functions for finding closest book
void getGreatestOnLeftHelper(redBlackTreeNode* root, int bookId, redBlackTreeNode*
&result);
```

```
void getSmallestOnRightHelper(redBlackTreeNode* root, int bookId,
redBlackTreeNode* &result);

// Return flip count
int getFlipCount();

// Print flip count
void printFlipCount();

// Function to print tree map for debugging
static void printTreeState(unordered_map<redBlackTreeNode*, bool> treeState);

// Create a new node and set book fields
redBlackTreeNode* createNewNode(int bookId, string bookName, string authorName,
bool availabilityStatus);

// Function to update flip counts
void updateFlipCount();

// Function to insert a new book into the Red-Black Tree
void insertBook(int bookId, string bookName, string authorName, bool
availabilityStatus);

// Function to maintain Red-Black Tree properties after insertion
void maintainInsert(redBlackTreeNode* node);

// Function to delete a book from the Red-Black Tree
```

```
void deleteBook(int data);

// Function to remove a node from the Red-Black Tree
redBlackTreeNode* removeNode(redBlackTreeNode* root, int key);

// Function to maintain Red-Black Tree properties after deletion
void maintainDelete(redBlackTreeNode* nodeToBeDeleted);

// Function to search for a book in the Red-Black Tree by book ID
redBlackTreeNode* search(int bookId);

// Recursive function to search for a book node in the Red-Black Tree
redBlackTreeNode* searchTree(redBlackTreeNode* node, int key);

// Function to perform an inorder search within a given range of book IDs in the Red-Black Tree
vector<redBlackTreeNode*> inorderSearch(int bookIdLeft, int bookIdRight);

// Recursive function to perform an inorder search within a given range of book IDs
void inorderSearchTree(redBlackTreeNode* node, int leftRange, int rightRange,
vector<redBlackTreeNode*> &validNodes);

// Function to perform a ranged search within a given range of book IDs in the Red-Black Tree
vector<redBlackTreeNode*> rangedSearch(int bookIdLeft, int bookIdRight);

// Recursive function to perform a ranged search within a given range of book IDs
```

```
void rangedSearchTree(redBlackTreeNode* node, int leftRange, int rightRange,
vector<redBlackTreeNode*> &validNodes);

// Function to perform a left rotation on a node in the Red-Black Tree
void rotateLeft(redBlackTreeNode* node);

// Function to perform a right rotation on a node in the Red-Black Tree
void rotateRight(redBlackTreeNode* node);

// Function to replace a node in the Red-Black Tree with another node
void replaceTree(redBlackTreeNode* oldNode, redBlackTreeNode* newNode);

// Constructor for the Red-Black Tree
RedBlackTree();

// Function to find the node with the minimum key value in the subtree rooted at the
given node
redBlackTreeNode* getMinimum(redBlackTreeNode* node);

// Function to find the node with the maximum key value in the subtree rooted at the
given node
redBlackTreeNode* getMaximum(redBlackTreeNode* node);

// Function to find the next node in the Red-Black Tree
redBlackTreeNode* getNextNode(redBlackTreeNode* node);

// Function to find the previous node in the Red-Black Tree
```

```
redBlackTreeNode* getPreviousNode(redBlackTreeNode* node);

// Function to retrieve the root of the Red-Black Tree
redBlackTreeNode* getRoot();
```

RedBlackTreeNode.h

```
// Getter and Setter for 'key'
int getKey();
void setKey(int _key);

// Methods to handle Book details associated with the node
Book getBook();
void setBookDetails(int bookId, string bookName, string authorName, bool
availabilityStatus);

// Getter and Setter for 'parent'
redBlackTreeNode* getParent();
void setParent(redBlackTreeNode* _parent);

// Getter and Setter for 'left'
redBlackTreeNode* getLeft();
void setLeft(redBlackTreeNode* _left);

// Getter and Setter for 'right'
redBlackTreeNode* getRight();
void setRight(redBlackTreeNode* _right);
```

```
// Methods to handle node color (Black or Red)
bool isBlack();
void setBlack();

bool isRed();
void setRed();
```

Book.h

```
// Adds details of a book
void addDetails(int _bookId, string _bookName, string _authorname, bool
_availabilityStatus, int _borrowedBy = -1);

// Retrieves the book's ID
int getBookId();

// Retrieves the book's name
string getBookName();

// Retrieves the book's author name
string getAuthorname();

// Checks if the book is available
bool isAvailabilable();

// Retrieves the ID of the borrower
int getBorrowedBy();
```

```
// Retrieves the reservation heap for the book
ReservationHeap getReservationHeap();

// Prints details of the book
void printDetails();

// Handles borrowing a book by a patron
void borrowBook(int _patronId, int _patronPriority);

// Handles returning a book by a patron
void returnBook(int _patronId, int _bookId);
```

ReservationHeap.h

```
// Initializes the heap's capacity
void initializeCapacity(int _capacity);

// Adds a node to the heap's nodeList
void addToNodeList(ReservationHeapNode node);

// Retrieves the heap's capacity
int getCapacity() const;

// Retrieves the size of the heap's nodeList
int getSize() const;
```

```
// Displays the contents of the nodeList
void displayList();

// Retrieves the index of the parent node
static int getParentIndex(int index);

// Retrieves the index of the left child node
static int getLeftChildIndex(int index);

// Retrieves the index of the right child node
static int getRightChildIndex(int index);

// Checks if the heap is empty
bool isEmpty() const;

// Pushes a node into the heap and maintains the heap property
void push(ReservationHeapNode node);

// Removes the top element from the heap and reorganizes the heap structure
void pop();

// Reorganizes the heap structure to maintain the property
void heapify(int currentIndex, int lastIndex);

// Retrieves the top element of the heap
ReservationHeapNode top();
```

```
// Retrieves keys (patron IDs) of elements in the heap
vector<int> getHeapKeys();
```

ReservationHeapNode.h

```
// Constructor to initialize the node with patron ID, priority number, and reservation
timestamp
ReservationHeapNode(int _patronID, int _priorityNumber,
chrono::time_point<chrono::high_resolution_clock> _timeOfReservation);

// Getter method to retrieve the patron ID
int getPatronID() const;

// Setter method to modify the patron ID
void setPatronID(int id);

// Getter method to retrieve the priority number
int getPriorityNumber() const;

// Setter method to modify the priority number
void setPriorityNumber(int priority);

// Getter method to retrieve the reservation timestamp
chrono::time_point<chrono::high_resolution_clock> getTimeOfReservation() const;

// Setter method to modify the reservation timestamp
```

```
void setTimeOfReservation(chrono::time_point<chrono::high_resolution_clock> time);

// Method to compare the current node with another node based on priority and
reservation time
bool isLessThan(ReservationHeapNode other);

// Method to check if the current node has a higher priority than another node
bool isGreaterThan(ReservationHeapNode other);
```