GatorLibrary Program Structure Report

Mayur Reddy Junnuthula

November 21, 2023

1 Introduction

This document outlines the structure and function prototypes of the GatorLibrary program, which implements a Red-Black Tree data structure to manage a library system.

2 Class and Function Prototypes

2.1 Class BookNode

```
class BookNode {
public:
    BookNode();
    BookNode(int, string, string, string, int);
    int getKey() const;
    friend ostream& operator<<(ostream&, const BookNode&);

    // Attributes
    int BookID;
    string BookName;
    string AuthorName;
    string AvailabilityStatus;
    int BorrowedBy;
    priority_queue<HeapNode, vector<HeapNode>, greater<HeapNode>> ReservationH
};
```

2.2 Class RBTreeNode

```
class RBTreeNode {
public:
    RBTreeNode();
    RBTreeNode(int, string, string, string, int, int);
    RBTreeNode(int);
```

```
    // Attributes
    BookNode bookData;
    RBTreeNode *parent, *left, *right;
    int color;
};
```

2.3 Class RBTree

```
class RBTree {
private:
    RBTreeNode *root, *null;
    void _search(RBTreeNode*, int);
    void _findClosestTraverse(vector<RBTreeNode*>&, RBTreeNode*, const int&,
    vector<RBTreeNode*> _findClosest(RBTreeNode*, int);
    void _assignColor(RBTreeNode*, int);
    void _transplantRedBlack(RBTreeNode*, RBTreeNode*);
    void _balanceDelete(RBTreeNode*);
    void _nodeDelete(RBTreeNode*, int);
    void _balanceInsert(RBTreeNode*);
    void rotateLeft(RBTreeNode*);
    void rotateRight(RBTreeNode*);
    void postOrderDelete(RBTreeNode*);
    void printInRange(RBTreeNode*, int, int, ofstream&);

public:
    RBTree();
    ~RBTree();
    RBTreeNode* getRoot() const;
    RBTreeNode* getNull() const;
    RBTreeNode* search(int);
    vector<RBTreeNode*> findClosest(int);
    RBTreeNode* findMax(RBTreeNode*);
    RBTreeNode* findMin(RBTreeNode*);
    void insert(const BookNode&);
    void nodeDelete(int);
    int colorFlipCount;
};
```

2.4 Class GatorLibrary

```
class GatorLibrary {
private:
    RBTree tree;
```

```
public:
    void PrintBook(int, ofstream&);
    void PrintBooks(int, int, ofstream&);
    void InsertBook(int, string, string, string, int, ofstream&);
    void BorrowBook(int, int, int, ofstream&);
    void ReturnBook(int, int, ofstream&);
    void DeleteBook(int, ofstream&);
    void FindClosestBook(int, ofstream&);
    void ColorFlipCount(ofstream&);
    static string trim(const string&);
    static vector<string> tokenize(istringstream&);
    void ExecuteOperations(const vector<string>&, ofstream&);
};
```

2.5 Main Function

```
int main(int argc, char* argv[]) {
    // Main function implementation
}
```

3 RBTree Function Descriptions

3.1 _search Function

Description: The _search function is a recursive method to find a node with a specific key in the Red-Black Tree. It traverses the tree following binary search tree properties until it finds the node or reaches a null node.

3.2 _findClosestTraverse Function

Description: The _findClosestTraverse function recursively traverses the Red-Black Tree to find nodes whose keys are closest to a given target key. It keeps track of the minimum difference between the node keys and the target key, updating a list of closest nodes.

3.3 _findClosest Function

Description: This function initiates the search for nodes closest to a given key. It first uses the _search function to check if a node with the exact key exists. If not, it calls _findClosestTraverse to find the closest nodes.

3.4 _assignColor Function

Description: _assignColor is used to set the color of a node to either red or black. It also increments the colorFlipCount if the color change occurs.

3.5 _transplantRedBlack Function

Description: The _transplantRedBlack function is used to replace one subtree as a child of its parent with another subtree. It is a utility function for deletion and insertion operations in the tree.

3.6 _balanceDelete Function

Description: _balanceDelete adjusts the Red-Black Tree after a node deletion to ensure that the tree maintains its properties. It performs rotations and color changes as needed.

3.7 _nodeDelete Function

Description: _nodeDelete removes a node with a given key from the tree. It handles cases of node deletion as per Red-Black Tree properties and calls _balanceDelete to fix any violations after deletion.

3.8 _balanceInsert Function

Description: _balanceInsert ensures the Red-Black Tree properties are maintained after inserting a new node. It resolves violations caused by inserting a node by performing rotations and recoloring.

3.9 insert Function

Description: The insert function adds a new node with the given bookData into the tree. It places the node at the correct position and then calls _balanceInsert to adjust the tree for maintaining its properties.

3.10 rotateLeft and rotateRight Functions

Description: These functions perform left and right rotations on a given node. Rotations are fundamental operations for maintaining the balance of a Red-Black Tree.

3.11 nodeDelete Function

Description: nodeDelete calls the _nodeDelete function to remove a node with a specified key from the tree.

3.12 printInRange Function

Description: printInRange prints all nodes within a given range of keys. It recursively traverses the tree and prints nodes whose keys lie within the specified range.

4 GatorLibrary Function Descriptions

4.1 PrintBook Function

Description: The PrintBook function searches for a book in the library using its bookID. If found, it prints the book's details to both the console and an output stream. If the book is not found, it notifies the user.

4.2 PrintBooks Function

Description: PrintBooks prints details of all books whose IDs fall within the specified range (bookID1 to bookID2). It leverages the RBTree's printInRange function to achieve this.

4.3 InsertBook Function

Description: This function creates a new BookNode with the given parameters and inserts it into the tree. It's used for adding new books to the library's collection.

4.4 BorrowBook Function

Description: BorrowBook allows a patron to borrow a book if it's available. If the book is currently unavailable, it adds the patron to the reservation heap based on their priority.

4.5 ReturnBook Function

Description: ReturnBook handles the return of a book to the library. It updates the book's availability status and handles the transfer to the next patron in the reservation heap, if any.

4.6 DeleteBook Function

Description: DeleteBook removes a book from the library. It also handles the cancellation of any reservations associated with the book.

4.7 FindClosestBook Function

Description: This function finds and prints the book(s) closest to a given target ID. It uses the RBTree's findClosest method for this purpose.

4.8 ColorFlipCount Function

Description: ColorFlipCount prints the number of color changes that have occurred in the RBTree, indicating how many times the tree has been rebalanced.

4.9 Utility Functions

Description: The class also includes utility functions like trim and tokenize for string manipulation and parsing, enhancing the functionality of other methods.

Red-Black Tree Balancing Operations Pseudocode

The Red-Black Tree is a self-balancing binary search tree with specific properties that need to be maintained after operations like insertion and deletion. These operations might involve rotations and color changes to maintain the tree's balance.

Balancing After Insertion

After inserting a node in a Red-Black Tree, the tree might need rebalancing. The balancing process involves checking the color of the parent and uncle nodes and performing specific rotations and color changes.

```
_balanceInsert(Node cur):
    while cur.parent.color is RED:
        if cur.parent is left child of cur.parent.parent:
            uncle = cur.parent.parent.right
            if uncle.color is RED:
                // Case 1: Recolor parent and uncle to BLACK, grandparent to RED
                cur.parent.color = BLACK
                uncle.color = BLACK
                cur.parent.parent.color = RED
                cur = cur.parent.parent
            else:
                if cur is right child of parent:
                    // Case 2: Left rotation needed
                    cur = cur.parent
                    rotateLeft(cur)
                // Case 3: Right rotation and recoloring
                cur.parent.color = BLACK
                cur.parent.parent.color = RED
                rotateRight(cur.parent.parent)
        else:
            // Mirror cases for cur.parent being right child
            ... (Similar logic as above with mirrored cases)
        if cur is root:
```

```
            break
    root.color = BLACK
```

Balancing After Deletion

When a node is deleted, the Red-Black Tree properties might be violated, requiring specific operations to rebalance the tree.

```
_balanceDelete(Node cur):
    while cur is not root and cur.color is BLACK:
        if cur is left child of its parent:
            sibling = cur.parent.right
            if sibling.color is RED:
                // Case 1: Left rotation at parent required
                sibling.color = BLACK
                cur.parent.color = RED
                rotateLeft(cur.parent)
                sibling = cur.parent.right
            if sibling.left.color is BLACK and sibling.right.color is BLACK:
                // Case 2: Sibling's children are black, recolor sibling
                sibling.color = RED
                cur = cur.parent
            else:
                if sibling.right.color is BLACK:
                    // Case 3: Left rotation at sibling required
                    sibling.left.color = BLACK
                    sibling.color = RED
                    rotateRight(sibling)
                    sibling = cur.parent.right
                // Case 4: Right rotation at parent required
                sibling.color = cur.parent.color
                cur.parent.color = BLACK
                sibling.right.color = BLACK
                rotateLeft(cur.parent)
                cur = root
        else:
            // Mirror cases for cur being right child
            ... (Similar logic as above with mirrored cases)
    cur.color = BLACK
```

These pseudocode examples illustrate the core logic behind maintaining the balance of a Red-Black Tree after insertion and deletion operations.