Homework 1

COP5536 Spring 2025 Programming Project

Name: Dakshina Tharindu
UFID: 7158-2749
Email: tharinduo@ufl.edu

March 31, 2025

---

1 Project Overview

The Broomstick Management System is a fictional system used to manage flying broomsticks manufactured and registered in the Ministry of Magic Office. Every broomstick registered in the system has a uniquely identified license plate number with 4 characters consisting of A-Z letters 0-9 numbers. The owner of a broomstick registered in the system is required to pay the registration and management fee of 4 Galleons annually. Also, a user can customize the license plate number by their own if it is already not in the system. However, there will be an additional cost of 3 Galleons incurred in customized plates annually.

The Broomstick Management System utilizes a Red-Black Tree as its underlying data structure. It is built using C++ from the scratch without using any C++ standards libraries. The following section of the report consist in-detail explanation of project structure, program structure and function prototype.

2 Project Structure

The project consists following files:

- `red_black_tree.h` - Header file of Red-Black Tree Class
- `red_black_tree.cpp` - Source file of Red-Black Tree Class
- `main.cpp` - Main function, input/output files handling helper functions
- `Makefile` - Make file to build the project

Use the command `make all` to compile and build the executable binary. After that, a binary named `plateMgmt` will be created in the same directory. To run the program, use the command `./plateMgmt <input_file>`.

3 Program Structure

The program is structured as follows:

1. Node Structure
2. Red-Black Tree Class

3.1 Node Structure

The `Node` struct represents a single node in the Red-Black Tree. It contains the following members and the contructor.

3.1.1 Members

Member	Usage
`plateNum`	A string representing the license plate number of the broomstick.
`color`	An enum value representing the color of the node
`left`	A pointer to the left child of the node.
`right`	A pointer to the right child of the node.
`parent`	A pointer to the parent node.
`customized`	A boolean value. True for customized broomsticks.

3.1.2 Constructor

```
Node(std::string plateNum);
```

The constructor of `Node` initize the members to following:

- `plateNum` to given `plateNum`.
- `left`, `right`, and `parent` pointers to `nullptr`.
- `color` to `RED`

3.2 Red-Black Tree Class

The `RedBlackTree` class represents the Red-Black Tree itself. It contains the following attributes and methods.

3.2.1 Attributs

Attribute	Usage
`root`	The pointer to the root node
`totalRevenue`	An integer to maintain the total revenue of the system

3.2.2 Methods

Public methods

1. *Constructor*

   ```
   RedBlackTree();
   ```

   The constructor initializes the `root` to `nullptr` and `totalRevenue` to 0.

2. *Destructor*

   ```
   RedBlackTree();
   ```

   The destructor is responsible for completely deleting the tree. It utilizes a recursive function to remove all nodes, beginning from `root`.

3. *Register new customized license plate*

```
bool addLicense(std::string plateNum);
```

This method calls the `insertLicense` method with the `customized` argument set to `true` in order to add a new license plate number to the Red-Black Tree. If the insertion is successful, it returns `true`; if not, it returns `false`.

4. *Register new randomized license plate*

```
std::string addLicense();
```

This method generates a random license plate number using the `randomPlate` method. It then invokes the `insertLicense` method with the customized argument set to `false`, in order to add the generated plate number to the tree. If the insertion fails, the method generates a new random plate number and repeats the process. This loop continues until a successful insertion occurs. Once a valid random number is found, the method returns it.

5. *Remove a license plate from the tree*

```
bool dropLicense(std::string plateNum);
```

This method manages the deletion of a license plate from the tree. It locates the node `z` with the specified `plateNum`. If found, the node is removed based on whether it is a leaf, has one child, or has two children. In cases where `z` has two children, the successor node `y` from its left subtree replaces `z`. The function ensures red-black properties are preserved through the `fixDeletion` method if necessary. It adjusts `totalRevenue` based on whether the plate was `customized`, deletes `z`, and confirms successful deletion by returning `true`. If `z` is not found, `false` is returned, indicating the plate does not exist in the tree.

6. *Check if a license plate exists*

```
bool lookupLicense(std::string plateNum);
```

This function checks if a given license plate number, `plateNum`, exists within a Red-Black tree. It starts at the `root` of the tree and traverses through its nodes by comparing the current node's license plate number to the input `plateNum`. Depending on whether the input is greater or lesser, the function navigates right or left, respectively. If a match is found, the function returns `true`, indicating the presence of the license plate. If the search reaches the end of the tree without finding a match, it returns `false`, indicating the absence of the license plate.

7. *Check the license plate number lexicographically previous to the input*

```
bool lookupLicense(std::string plateNum);
```

This function finds the license plate number that immediately precedes a given `plateNum` in the tree. It traverses through the tree starting from the `root`, updating a pointer `prev` to the last node visited that has a plate number less than `plateNum`. This navigation follows a binary search pattern, moving right when the current

node's plate number is less than `plateNum` and left otherwise. Upon reaching the end of the tree (node becomes `nullptr`), the function returns the plate number of the `prev` node if it exists; otherwise, it returns an empty string if no such preceding plate number was found.

8. *Check the license plate number lexicographically next to the input*

```
std::string lookupNext(std::string plateNum);
```

This function finds the license plate number that immediately succeeds a given `plateNum` within the tree. Starting at the `root`, the function looks for a node where the plate number is greater than `plateNum`, updating the next pointer to such nodes. The traversal uses a binary search approach, moving left if the current node's plate number is greater than `plateNum` and right otherwise. This ensures that the smallest plate number larger than the given one is retained. Upon completion of the traversal (node becomes `nullptr`), the function returns the plate number stored in the next pointer if it exists; if not, it returns an empty string, indicating no greater license plate number was found.

9. *Check all license plates between `lo` and `hi` in lexicographical order*

```
std::vector<std::string> lookupRange(std::string lo,std::string hi);
```

This method finds and returns all license plate numbers within a specified range, from `lo` to `hi`. The function begins by initializing an empty vector `result` to store the matching plate numbers and sets a pointer node to the tree's root. It uses an in-order traversal, implemented as a recursive lambda function `inOrder`, to explore the tree.

During the traversal, the function checks if the current node's plate number is within the given range (`lo` to `hi`). If the condition is met, the plate number is added to the result `vector`. The traversal first explores the left child, then the node itself, and lastly the right child. This make sure the lexicographical order of the output.

After the traversal is complete, the function returns the result `vector` containing all license plate numbers found within the specified range. This method efficiently collects and returns a subset of nodes based on their key values in a lexicographical order.

10. *Report the annual revenue*

```
int revenue();
```

This function returns the value of `totalRevenue` attribute. The `totalRevenue` is updated in the `insertLicense` mehod and `dropLicense` method.

Private methods

11. *Generate random plate number*

```
std::string randomPlate();
```

Generates a random license plate number consists of four characters that can be either uppercase letters (A-Z) or digits (0-9). It builds the license plate by iterating four times. In each iteration, a random number between 0 and 35 is generated, corresponding to the combined total of 26 letters and 10 digits. If the random number is less than 26, it corresponds to a letter, which is determined by adding the random number to the ASCII code of 'A'. If the random number is 26 or greater, it corresponds to a digit, calculated by subtracting 26 from the random number and then adding the result to the ASCII code of '0'. The chosen character is then appended to the `plateNum` string. After four iterations, the function returns the generated four-character license plate number.

12. *Perform the LL Rotation*

```
void LLRotation(Node *&node);
```

Executes a left-left rotation around the `node` pointer to maintain the red-black properties after insertion or deletion operations that might have disturbed the tree's balance.

13. *Perform the RR Rotation*

```
void RRRotation(Node *&node);
```

Executes a right-right rotation around the `node` pointer to maintain the red-black properties after insertion or deletion operations that might have disturbed the tree's balance.

14. *Transplant*

```
void transplant(Node*& u, Node*& v);
```

This method replaces one subtree(`u`) with another(`v`). This replacement involves updating the parent of `u` to point to `v` instead of `u`, effectively removing `u` from its position in the tree and inserting `v` in its place. If `u` was the `root` of the tree, then `v` becomes the new `root`. If `u` was a child (either left or right), the appropriate child reference of `u`'s parent is updated to `v`. Additionally, `v`'s parent pointer is set to what was previously `u`'s parent.

15. *Find the maximum node in a subtree*

```
Node* maximumNode(Node* node);
```

Find and return the node with the maximum value starting from a given node in the tree. It performs this by traversing the right child nodes iteratively. Since the right child of any node in a binary search tree, contains values greater than the node itself, the function keeps moving right until it reaches the farthest right node, which does not have a right child.

16. *Insert a license plate into the red-black tree*

```
bool insertLicense(std::string plateNum, bool customized);
```

This function inserts a new license plate into the tree. It starts by creating a new node, `newPlate`, with the given `plateNum` and customized status. The function then searches for the correct insertion point by comparing `plateNum` with existing nodes in the tree. If a duplicate `plateNum` is found, the function deletes the newly created node to prevent insertion and returns `false`. If no duplicates are found, `newPlate` is inserted as a leaf at the appropriate spot determined by whether it is less than or greater than the `plateNum` of its parent node. After insertion, the `fixInsertion` function is called to ensure that the red-black tree properties are maintained post-insertion.

Additionally, the function updates the `totalRevenue`, adding a specific amount based on whether the plate is `customized`. It returns `true` upon successful insertion of the new node.

17. *Fix the red-black tree properties after insertion*

```
void fixInsertion(Node*& newPlate);
```

Addresses violations of red-black properties following the insertion of a new node. It checks if the newly inserted node and its parent are both red, which violates the property forbidding two consecutive red nodes. If there is a violations, it perform the `XYz` process starting from the `newPlate` pointer to fix the Red-Black tree properties. Step by step explanation is commented in the code.

18. *Fix the red-black tree properties after deletion*

```
void fixDeletion(Node*& x, Node*& x_parent);
```

Handles potential violations in the tree's properties following the deletion of a node. It aims to restore balance and maintain the red-black properties through recoloring and rotations. If there is a violations, it perform the `Xcn` strategies to fix the properties. Step by step explanation is commented in the code.