CH08-320143: Assignment #9

Harit Krishan
h.krishan@jacobs-university.de

Jacobs University Bremen — April 13, 2020

1 Understanding Red Black Trees

A red black tree is a self-balancing binary search tree. If elements are inserted in sorted order into a binary search tree, the result becomes a linked list due to the lack of a tree structure. Red black trees introduce a binary variable into the nodes, giving it an additional property which we call 'color'. This color is either red or black. The logic behind the color variable ensures we get a balanced tree as a result. The color constraint properties are embedded into the tree class.

1.1 Traversal Example 1

We insert the following integer array into an empty red black tree. After each insertion, we view the balanced red black tree. The tree we see after the last insertion is the resulting red black tree. Figure 1a shows the implementation of the described methodology written in C++.

```cpp
#include <bits/stdc++.h>
#include "rbtree.h"

using namespace std;

int main() {
    int arr[] = {13, 44, 37, 7, 22, 16};
    RedBlackTree tree;

    for(int i = 0; i < 6; i++) {
        tree.insert(arr[i]);
        tree.order();
    }

    cout << endl << endl;

    tree.rbtprint_text();

    cout << endl << endl;

    return 0;
}
```

Figure 1: Implementation of Insertion and Printing in C++

Since the color propert is binary, we denote colors with 0 (RED) and 1 (BLACK). In Figure 2, we can see the sequence of insertion and the resulting tree after each insertion operation. We observe the sequence of events as elements are inserted into the tree. The final resulting tree is described after the last insertion of the execution window.

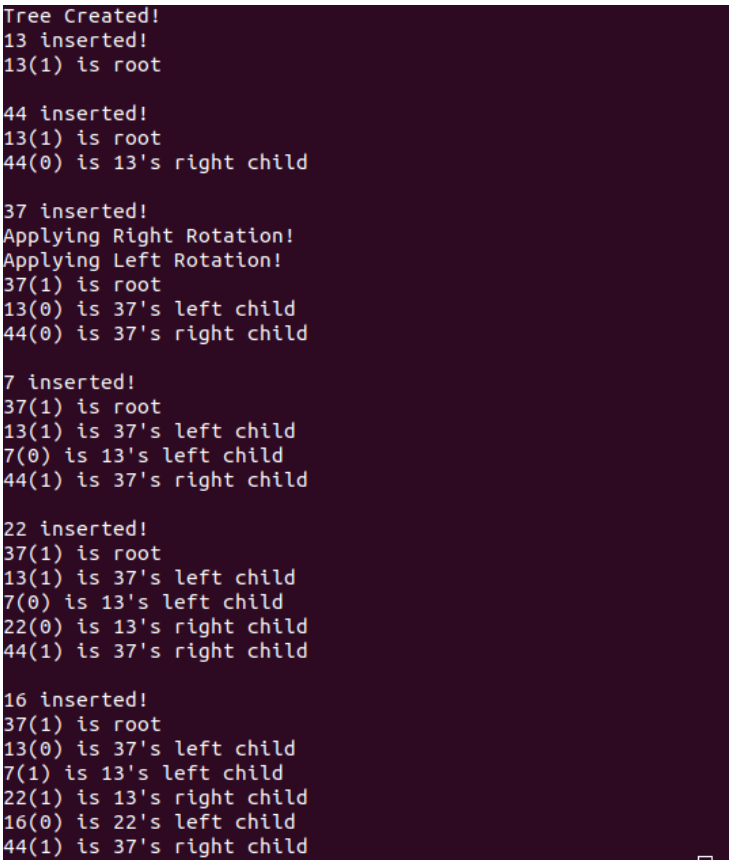


Figure 2: Insertion of Elements into Red Black Tree

1.2 Traversal Example 2

In this problem, we want to draw all valid red black trees that can store the integers $1, 2, 3, 4$. To test this, we must pass all possible variations of inserting these four integers. Figure 3 shows the code snippet which encodes this in C++.

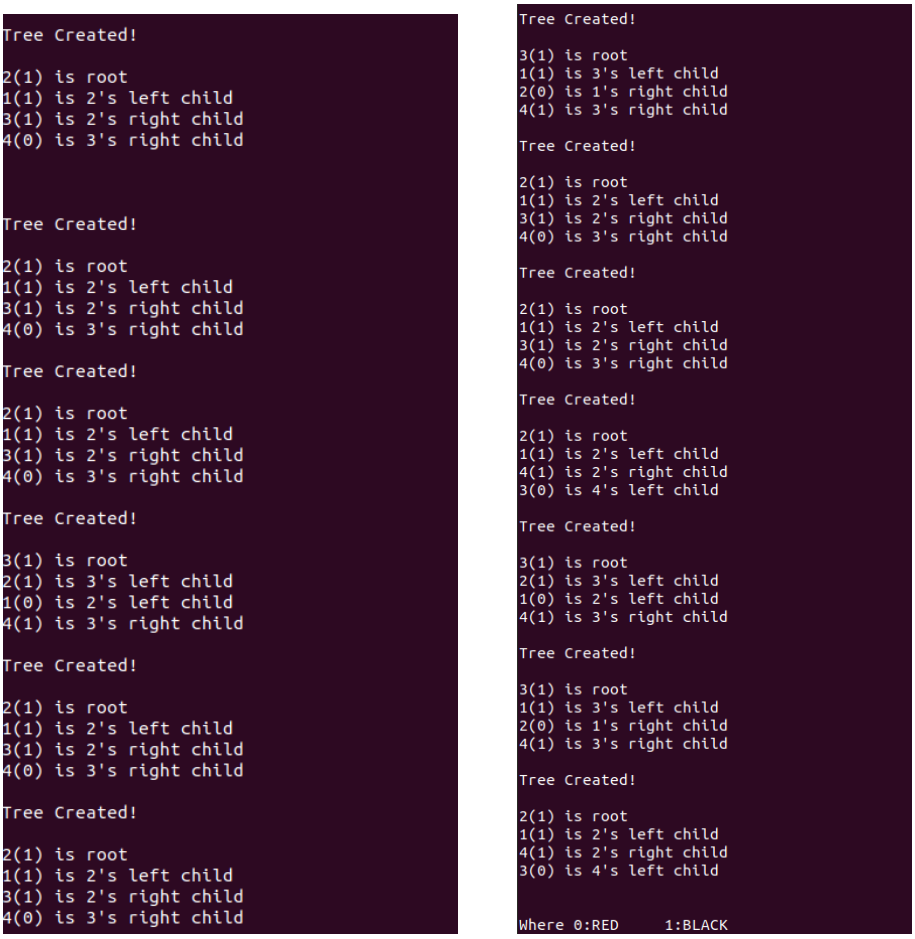
```cpp
#include <bits/stdc++.h>
#include "rbtree.h"

using namespace std;

int main() {

    RedBlackTree tree_init;

    int arr1[] = {1,2,3,4};

    for(int i = 0; i < 4; i++) {
        tree_init.insert(arr1[i]);
    }

    tree_init.rbtprint_text();

    cout << endl << endl;

    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            if(i != j) {
                // All variations of input
                int temp[4];
                copy(begin(arr1), end(arr1), begin(temp));
                swap(temp[i], temp[j]);
                RedBlackTree tree;
                for(int i = 0; i < 4; i++) {
                    tree.insert(temp[i]);
                }

                tree.rbtprint_text();
            }
        }
    }

    cout << endl << endl;

    cout << "Where 0:RED\t1:BLACK\n";

    return 0;
}
```

Figure 3: Trying All Variations of Array to Store in Red Black Tree (C++)

The execution of this script will print all possible red black trees that can be constructed with the elements 1, 2, 3 and 4. Figure 4 shows the execution logs and the balanced trees.

2 Implementing Red Black Trees

For implementing a red black tree, I use C++. We follow the template shown in Figure 5. The completed header file is available in the root directory of this submission called $rbtree.h$. In my implementation, I define some additional private functions to facilitate in implementing all the suggested functions and constructors. I have solved parts 9(a) and 9(b) using this implementation. The respective tester files have also been provided in the submission directory.



(a) (b)

Figure 4: All Possible Red Black Trees Consisting of Integers 1,2,3,4

```
enum Color {RED, BLACK};
struct Node
{
  int data;
  Color color;
  Node *left, *right, *parent;
};
class RedBlackTree
{
  private:
    Node *root;
  protected:
    void rotateLeft(Node *&);
    void rotateRight(Node *&);
  public:
    RedBlackTree();
    void insert(int);
    void delete(Node *&);
    Node * predecessor(const Node *&);
    Node * successor(const Node *&);
    Node * getMinimum();
    Node * getMaximum();
    Node * search(int);
};
```

Figure 5: Class Definition of Red Black Tree in C++