Kyle Tham

Professor Wu

CS146

15 December 2017

Red-Black Tree Programming Assignment Report

I. Key Concepts

The key concepts of the red-black tree that I have implemented is creating an rb-tree that maintains the 5 properties;

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

In addition, the rb-tree must at least contain the following functions:

a) Build up a patients RB-Tree. The color property of each node must be presented.
b) Insert a patient to the RB Tree based on his or her priority code. Show the changes of the color property changes of patient nodes.
c) Search a patient's name by entering a priority code.
d) Delete a patient from the RB-Tree, and show the color property changes.
e) Make a sorted list of patients based on the priority codes.

So, with the specifications and the 5 properties in place, the first key concept I glossed over is creating a **Node** class. The node class represents the patients in our hospital waiting room. The class contains a constructor that takes a String (the name of our patient) and an integer (the key, or level of severity of the said patient's injury). Now, each Node also has a left child, right child, a parent node and a COLOR, which can be either red or black. In the class, they are not initialized because the RBTree class will automatically initialize it in its methods, which we will cover. In the **RBTree** class, the key concepts are the main functions that help maintain the properties – **rbTreeSearch, inOrderTreeWalk, rbInsert, and rbDelete**. The way that all of these functions come together is that we can create a Red-Black tree, and insert patients into the tree by using the **rbInsert** function. This function places the nodes into the tree by using regular BST method – every node has a left child and a right child, and the nodes in the left subtree are smaller than the key of the node, and the nodes in the right subtree are bigger than the key of the node. But, we can not only follow that property, so another function called **rbInsertFixup** is introduced to help maintain the color properties of the tree. We will specify more on that later. In addition, we can use **inOrderTreeWalk** to print out the red-black tree in order. **treeSearch** is a versatile function that can be used in many different ways, but the main reason it exists is to see if a node with a certain key exists in the tree. Lastly, rbDelete does exactly what it sounds like and deletes a node from the tree. A function **rbDeleteFixup** is introduced to maintain the properties of the rb-tree because deleting is very tricky. Again, more will be touched upon on this method later.

II. Explanation of Classes and the Purpose of Each Function

As explained above, there are two classes in our Red-Black Tree project. The first class is **NODE.** The nodes basically represent our patients in the waiting room. Each node has a name and a level of severity of their injury. In addition, each node has a left child, right child, parent node, and a color. Here are a list of functions in the Node Class:**Node(String name, int key) –** this is the constructor of the class.

Parent() – this function returns the parent of a node, which is useful in all the methods talked about in the introductory paragraph.

Left() – this function returns the left child of a node.

right() – this function returns the right child of a node.

key() – this function returns the key of a node.

Name() – this function returns the name of a node.

Color() – this function returns the color of a node.

The second class in our project is **RBTree**. Here are a list of functions in the class:

Root() – this function returns the root of a tree, which is the very top node.

inOrderTreeWalk() – the purpose of this function is to print out a trees nodes, in order of keys. It takes a Node as a parameter, and prints the left subtree as well as the right subtree in order.

printPreOrder() – this function returns the pre-order of a tree, which is useful when we want to redraw the tree to see if it maintains the binary search tree property.

rbTreeSearch() – the purpose of this function is to check if a target node is in the red-black tree. If it is, we return the node.

treeMin() – the purpose of the function is to find the Node with the smallest key of a specific nodes Tree.

treeMax() – it has the same purpose as treeMin, except it finds the Node with the largest

leftRotate() – Changes the pointer structure of several nodes, with the purpose of maintaining the red-black tree properties. When we do a left rotation on a node x, we assume that its right child y is not T:*nil*; x may be any node in the tree whose right child is not T:*nil*. The left rotation "pivots" around the link from x to y. It makes y the new root of the subtree, with x as y's left child and y's left child as x's right child.

rightRotate() – Same purpose and functionality as leftRotate, except it is done symmetrically.

rbInsert() – The purpose of this function is to insert a node into our red-black tree, while maintaining the red-black tree property.

rbInsertFixup() – When we insert a node into the RB-Tree, we might break a property. This function makes sure we fix the violation of the properties by recoloring nodes based on 3 cases:

i. A node *z's* uncle *y* is red
ii. A node *z's* uncle *y* is black and *z* is a right child
iii. A node *z's* uncle *y* is black and *z* is a left child.

Transplant – this function is used in delete, where it replaces one subtree as a child of its parent with another subtree.

rbDelete – tthis function deletes a node from our Red-Black tree, while maintaining the properties by calling rbDeleteFixup.

rbDeleteFixup – This function restores properties 1, 2, and 4 after deletion by recoloring and doing rotations.

randomNameList – the purpose of this function is to generate an array list of 20 names in a random order. This is used in our simulation so that there is an arbitrary order in the name, rather than a fixed order. It inserts 20 names into an arraylist and shuffles the list.

Simulation – this is where the actual simulation occurs, which will be walked through below.

III. Simulation With Screenshots

```java
/*
 * This function simulates an emergency room where 20 patients with
 * different, random severity of injuries are inserted into a red-black
 * tree one by one.
 */
static void simulation() {
    int patient = 0;
    ArrayList<String> names = randomNameList();
    RBTree eRoom = new RBTree();
    ArrayList<Integer> numbers = new ArrayList<>();
    for (int i = 1; i <= 20; i++) {
        numbers.add(i);
    }
    Collections.shuffle(numbers);
    for (int i = 0; i < numbers.size(); i++) {
        Node test = new Node(names.get(i), numbers.get(i));
        rbInsert(eRoom, test);
    }
```

```java
/*
 * Creates an array list with 20 names, which is then shuffled.
 */
static ArrayList<String> randomNameList() {
    ArrayList<String> names = new ArrayList<String>();
    names.add("Eric");
    names.add("Nancy");
    names.add("Jonathan");
    names.add("Kevin");
    names.add("Kyle");
    names.add("Justin");
    names.add("Monty");
    names.add("Professor Mike Wu");
    names.add("Mike Wu's Old Manager");
    names.add("Phuong");
    names.add("Mark");
    names.add("Sam");
    names.add("Connor");
    names.add("Katie");
    names.add("Martin");
    names.add("Adrian");
    names.add("Jackie");
    names.add("Vidya");
    names.add("Edwin");
    names.add("Hung");
    Collections.shuffle(names);
    return names;
}
```

The purpose of the function is typed out in the header. It starts by creating an array list of 20 names in a randomized order. Then, an array that contains the number 1-20 is created in a random order as well. What proceeds is a for loop, where inside a Node is created with the first element of the array list of names as well as the first element of the array list of numbers, and will continue to do this for the whole list. With each node that is created, it is inserted into the red-black tree.

```java
/*
 * Inserts a node into an RBTree, while maintaining the RB-Tree properties by
 * calling rbInsertFixup.
 */
static void rbInsert(RBTree T, Node n) {
    Node y = nil;
    Node x = T.root;
    while (x != nil) {
        y = x;
        if (n.key < x.key) {
            x = x.left;
        } else {
            x = x.right;
        }
    }
    n.parent = y;
    if (y == nil) {
        T.root = n;
    } else if (n.key < y.key) {
        y.left = n;
    } else {
        y.right = n;
    }
    n.left = nil;
    n.right = nil;
    n.color = RED;
    T.root().color = BLACK;
    rbInsertFixup(T, n);
}
```

```java
/*
 * Maintains the red black tree property after using rbInsert by
 * recoloring nodes based on certain cases.
 */
static void rbInsertFixup(RBTree T, Node n) {
    while (n.parent.color == RED) {
        if (n.parent == n.parent.parent.left) {
            Node y = n.parent.parent.right;
            if (y.color == RED) {
                n.parent.color = BLACK; // Case 1
                y.color = BLACK; // Case 1
                n.parent.parent.color = RED; // Case 1
                n = n.parent.parent;
            } // Case 1 }
            else {
                if (n == n.parent.right) {
                    n = n.parent;
                    leftRotate(T, n);
                }
                n.parent.color = BLACK;
                n.parent.parent.color = RED;
                rightRotate(T, n.parent.parent);

            }
        } else { // Reverse
            if (n.parent == n.parent.parent.right) {
                Node y = n.parent.parent.left;
                if (y.color == RED) {
                    n.parent.color = BLACK;
                    y.color = BLACK;
                    n.parent.parent.color = RED;
                } else {
                    if (n == n.parent.left) {
                        n = n.parent;
                        rightRotate(T, n);
                    }
                    n.parent.color = BLACK;
                    n.parent.parent.color = RED;
                    leftRotate(T, n.parent.parent);
                }
            }
        }
    }
    T.root().color = BLACK;
}
```

Here is the insert function, as well as the rbInsertFixup function.

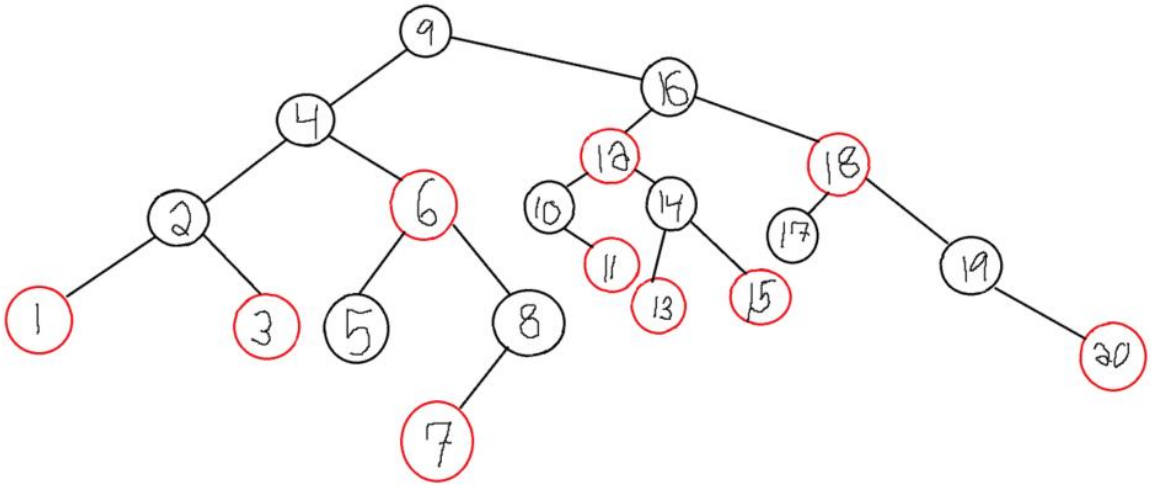
After the nodes are inserted, we prompt the user to see if they'd like to see the list of patients. If they type Y, then the list will be printed IN ORDER. Otherwise, they can choose who they want to delete.

```
Scanner scan = new Scanner(System.in);
System.out.println("You may now delete a patient from the binary search tree.");
System.out.print("Would you like to see the list first? Type Y if so, and type anthing else if not: ");
String input = scan.next();
if (input.toUpperCase().equals("Y")) {
    System.out.println("LIST OF PATIENTS.");
    inOrderTreeWalk(eRoom.root());
    System.out.println("============================");
    System.out.println(eRoom.root().key + ", " + eRoom.root().name + "(" + eRoom.root.color + ")   (Root)");
    System.out.println("Sorted Left Subtree");
    inOrderTreeWalk(eRoom.root().left);
    System.out.println("Sorted Right Subtree");
    inOrderTreeWalk(eRoom.root().right);
    System.out.println("----------------------------");
    if (eRoom.root().left != nil) {
        System.out.println("Roots left child = " + eRoom.root().left.key);
    } else {
        System.out.println("Roots left child is null.");
    }
    if (eRoom.root().right != null) {
        System.out.println("Roots right child = " + eRoom.root().right.key);
    } else {
        System.out.println("Roots right child is null.");
    }
    System.out.println("Pre-Order: ");
    printPreOrder(eRoom.root());
    System.out.println();
}
System.out.print("Which patient would you like to delete? Enter their level of severity (A NUMBER): ");
```

```
You may now delete a patient from the binary search tree.
Would you like to see the list first? Type Y if so, and type anthing else if not: y
LIST OF PATIENTS.
1, Adrian. Color: RED
2, Katie. Color: BLACK
3, Hung. Color: RED
4, Sam. Color: BLACK
5, Monty. Color: BLACK
6, Jonathan. Color: RED
7, Jackie. Color: RED
8, Mark. Color: BLACK
9, Martin. Color: BLACK
10, Nancy. Color: BLACK
11, Vidya. Color: RED
12, Phuong. Color: RED
13, Eric. Color: RED
14, Connor. Color: BLACK
15, Edwin. Color: RED
16, Kyle. Color: BLACK
17, Justin. Color: BLACK
18, Professor Mike Wu. Color: RED
19, Kevin. Color: BLACK
20, Professor Wu's Old Manager That He Hated. Color: RED
============================
9, Martin(BLACK)   (Root)
Sorted Left Subtree
1, Adrian. Color: RED
2, Katie. Color: BLACK
3, Hung. Color: RED
4, Sam. Color: BLACK
5, Monty. Color: BLACK
6, Jonathan. Color: RED
7, Jackie. Color: RED
8, Mark. Color: BLACK
Sorted Right Subtree
10, Nancy. Color: BLACK
11, Vidya. Color: RED
12, Phuong. Color: RED
13, Eric. Color: RED
14, Connor. Color: BLACK
15, Edwin. Color: RED
16, Kyle. Color: BLACK
17, Justin. Color: BLACK
18, Professor Mike Wu. Color: RED
19, Kevin. Color: BLACK
20, Professor Wu's Old Manager That He Hated. Color: RED
----------------------------
Roots left child = 4
Roots right child = 16
Pre-Order:
9 4 2 1 3 6 5 8 7 16 12 10 11 14 13 15 18 17 19 20
Which patient would you like to delete? Enter their level of severity (A NUMBER):
```

As we can see from the code, an if statement checks if the input is Y, and prints out the list in order. If it is not, it asks the user which patient they'd like to delete from the tree.

This is the output of the above code if the user types y. As we can see, we have the sorted list of patients with their respective level of severity of their injury, their name, and their color. In addition, the root is shown, as well as the pre-order. Using the pre-order, we can see if the tree maintains the red-black tree properties by drawing it out. The drawing is on the next page:



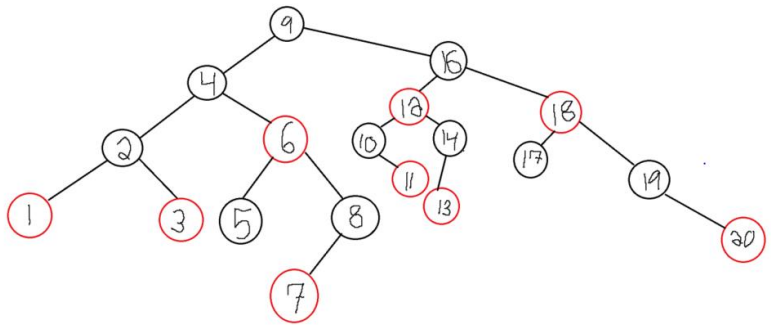
Here is the resultant tree. As we can see, it maintains the BST Property, where every node has a left subtree that contains nodes with keys less than the nodes key, and a right subtree that contains nodes with keys greater than the nodes key. We can also see that it maintains all the RB-Tree properties!

Next in the simulation, after we prompt the user to enter a number in the range of the list of injured patients, we will check to see if the number the user enters is valid. If they do not enter a number in the list, we keep prompting the user until they enter a valid number in the list. If they enter something outrageous, such as a fraction, decimal, or a string, we deem them incompetent and by default delete the most critically injured patient, which would be 20.

```
try {
    patient = scan.nextInt();
    while (patient < 1 || patient > 20) {
        System.out.print("You did not enter a number in the range of the list. Please try again: ");
        patient = scan.nextInt();
    }
} catch (Exception e) {
    System.out.println("You did not enter a whole number. Sorry. By default, the most");
    System.out.println("critically injured patient will now be deleted.");
    patient = treeMax(eRoom.root()).key;
}
System.out.println("================================");
System.out.println("AFTER DELETING THE TARGET PATIENT:");
rbDelete(eRoom, eRoom.rbTreeSearch(eRoom.root(), patient));
System.out.println(eRoom.root().key + ", " + eRoom.root().name + " " + eRoom.root.color + "  (New Root)");
System.out.println("New Sorted Left Subtree");
inOrderTreeWalk(eRoom.root().left);
System.out.println("New Sorted Right Subtree");
inOrderTreeWalk(eRoom.root().right);
System.out.println("--------------------------------");
if (eRoom.root().left != null) {
    System.out.println("Roots new left child = " + eRoom.root().left.key);
} else {
    System.out.println("Roots new left child is null.");
}
if (eRoom.root().right != nil) {
    System.out.println("Roots new right child = " + eRoom.root().right.key);
} else {
    System.out.println("Roots new right child is null.");
}
System.out.println("================================");
System.out.println("Pre-Order: ");
printPreOrder(eRoom.root());
```

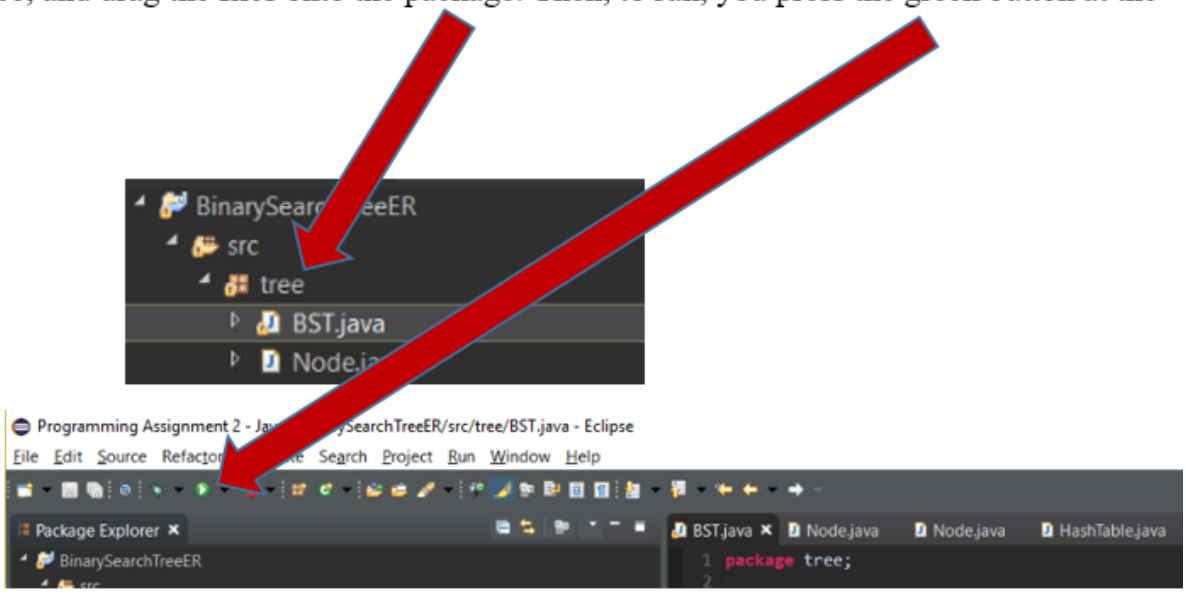
```
Which patient would you like to delete? Enter their level of severity (A NUMBER): 40
You did not enter a number in the range of the list. Please try again: 25
You did not enter a number in the range of the list. Please try again: 0
You did not enter a number in the range of the list. Please try again: 15
================================
AFTER DELETING THE TARGET PATIENT:
9, Martin BLACK  (New Root)
New Sorted Left Subtree
1, Adrian. Color: RED
2, Katie. Color: BLACK
3, Hung. Color: RED
4, Sam. Color: BLACK
5, Monty. Color: BLACK
6, Jonathan. Color: RED
7, Jackie. Color: RED
8, Mark. Color: BLACK
New Sorted Right Subtree
10, Nancy. Color: BLACK
11, Vidya. Color: RED
12, Phuong. Color: RED
13, Eric. Color: RED
14, Connor. Color: BLACK
16, Kyle. Color: BLACK
17, Justin. Color: BLACK
18, Professor Mike Wu. Color: RED
19, Kevin. Color: BLACK
20, Professor Wu's Old Manager That He Hated. Color: RED
--------------------------------
Roots new left child = 4
Roots new right child = 16
================================
Pre-Order:
9 4 2 1 3 6 5 8 7 16 12 10 11 14 13 18 17 19 20
```

As we can see, we enter the numbers 40, 25, and 0. When we enter those numbers, it prompts the user again to enter another number. Once we enter a valid number (15), it shows us the new tree, in order. It shows the roots new left and right child, if it changed. In addition, the pre-order is printed so that we can see if it still maintains the property. As we can see, it still maintains the 5 properties.



IV. Unzipping, Installing, and Testing the Code

The way to retrieve the files of this code is to open the zip file, and extract all the files inside into a folder. Inside, there's should be two java files, which is the source code. Once those files are in a folder, you create a new java project, with a package in the source, and drag the files onto the package. Then, to run, you press the green button at the top.



V. Problems Encountered

The main problem that I encountered during implementation is that the pseudocode that was provided by the textbook was wrong. After implementing the code into java from the textbook, I tested it, and to no avail it did not work. It would get stuck in infinite loops in certain places. The way that I got past this obstacle was by using the debug feature in Eclipse. I traced the code and saw that there was a part in the rbInsertFixup method, in the case where if a Nodes uncle was nil, the method would loop forever. In addition, creating a nil node was a challenge for me. At first, I let each nodes left, right, and parent be the regular "NULL" variable in java. But, when inserting nodes into the tree, it would throw a NullPointerException, because a null object is literally nothing in Java, just a nameholder for an empty space of data. So, I fixed that by creating a static final Node *nil*, which was initialized as Node(null, 0). So, it was a node still, but it held no data. Then, every node who didn't have a valid left or right child pointed to this nil node that I created.

VI. Lessons Learned

The lessons that I learned by making this implementation was really seeing how red-black trees functioned. I learned that each function served one sole purpose – to maintain the five properties of red-black trees. I learned that this was very, very important because maintaining the five properties of red-black trees guarantees that the tree is balanced. In a Binary Search Tree, which is very similar, the tree has a chance to be very imbalanced. For example, if the root was 5, and 50 nodes that are inserted into the tree have keys larger than 5, there would be 50 nodes in the right subtree of 5, and 0 in the left subtree. Red-black trees guarantee that it is balanced, and since we can guarantee it is balanced, we can guarantee a O(*log n)* time in the worst case, which is very critical for fast searching, inserting, and deleting in a red-black tree.