Project 4 Report

Paul Chon

November 2020

Note:

I mainly used Intellij to code, but I used Eclipse for the JUnit testing and exporting the project.

A Red Black Tree

A.1 OBJECTIVE:

The objective of this project is to implement a red black tree and use it to create a dictionary to use as a spellchecker.

A.2 IMPLEMENTATION:

I did a generic implementation of a red black tree instead of just a red black tree for strings. I created a Node class, a Visitor interface, and a RedBlackTree class to implement a generic red black tree. The Node class represents a single node in the RedBlackTree and has a key to hold data, a parent node, a left child node, a right child node, a boolean isRed to show whether the node is red or not, and an int color to also show whether the node is red or not (0 is for red and 1 is for black). This class also contains a method, isLeaf, to check whether a node is a leaf node or not. The Visitor interface has a method to visit a single node in the RedBlackTree. The RedBlackTree class represents the whole red black tree. Also, this class uses a similar way to add nodes like a binary search as seen in the addNode method, but since this is a red black tree, the fixTree method, seen in Figure 1 is added to make sure that the properties for red black trees is kept. In addition, for the RedBlackTree class, I use a node called NULL to represent any null leaves.

```java
public void fixTree(Node<Key> current) {
    // Temporary variable to hold another Node.
    Node<Key> temp;

    // If the current node's parent's color is red (0), then need to fix tree because the
    // current node will be 0 because of the addNode method.
    while (current.parent.color == 0) {
        temp = getAunt(current);
        if (isLeftChild(getGrandparent(current), current.parent)) {
            if (temp.color == 0) {
                // Case 1: Current's aunt is red
                current.parent.color = 1;
                current.parent.isRed = false;
                temp.color = 1;
                temp.isRed = false;
                getGrandparent(current).color = 0;
                getGrandparent(current).isRed = true;
                current = getGrandparent(current);
            } else if (current == current.parent.rightChild) {
                // Case 2: Aunt is black and current node is a right child
                current = current.parent;
                rotateLeft(current);
            } else {
                // Case 3: Aunt is black and current node is a left child
                current.parent.color = 1;
                current.parent.isRed = false;
                getGrandparent(current).color = 0;
                getGrandparent(current).isRed = true;
                rotateRight(getGrandparent(current));
            }
        } else {
```

Figure 1: Part of the fixTree method. The rest of the method is symmetrical cases and at the end changes the color of the root node to black.

A.3 Testing:

To test my implementation of the Red Black Tree. I have a manual test where I manually insert letters into the red black tree and then use makeString and makeStringDetails to make sure that they were inserted correctly. I have another test called spellChecker() that takes words from a txt file full of words and inserts them into a Red Black Tree. Then, I have two more txt file, noErrorDocument and exampleDocument, where I will use the lookup() method to see if words in noErrorDocument and exampleDocument are found in the Red Black tree. Also, I will time the creation of the dictionary and the time to perform lookup(). For the time to perform lookup, I take the times of every single lookup in one test and average them up, so one single entry in Figure 3 is technically already an average.I also have another manual test called intTest where I manually insert integers into the red black tree and use makeString and makeStringDetails to make sure that they were inserted correctly. This test is done to show that my implementation of Red Black Tree is generic.

Input Size	Elapsed Time (in ns)
1	208649500
2	233555700
3	219893600
4	216707100
5	276052600
6	217178800
7	215054300
8	250143900
9	207830200
10	209268300

Figure 2: Time to create dictionary inserting words into a red black tree.

Input Size	Elapsed Time (in ns)
1	1678.81
2	1904.38
3	1560.94
4	1636.04
5	917.96
6	1741.77
7	1572.55
8	3554.19
9	1410.87
10	1475.8

Figure 3: Time to look up word in red black tree.

Operation Name	Average time (in ns)
Creating Dictionary	255433400
Looking up words	1745.331

Figure 4: Average elapsed times for different operations.