Shohin Abdulkhamidov

Professor K. Potika

CS-146

December 3, 2021

Programming Project 4 and its goal.

In this project we created a Dictionary, aka a Red Black Tree and inserted the provided link to a dictionary with the help of Red Black Tree

It consists of such classes below:

a. RedBlackTree.java consists of isLeaf, visit, printTree, addNode, insert, lookup, getSiblings, getAunt, getGrandparent, rotateLeft, rotateRight, fixTree, isEmpty, isLeftChild, preOrderVisit methods.
b. RBTTester.java is JUnit for RedBlackTree class
c. Dictionary.java consists of generateDictionary(), wordsOfPoem(), and spellChecker() methods
d. DictionaryJUnit.java is JUnit for Dictionary class
e. Node.java consists of getRoot, setRoot, compareTo parameters and isLeaf methods
f. Visitor.java consists of visit parameter

Additional Requirements (Optional)

a. I created JUnit class and separated all the test cases in that class to avoid having long lines of code inside the Main class

b. Inside the JUnit class I created a method to better understand how efficient my algorithms. It basically shows the nanoseconds spent while traversing the method.

Cases Consider

Tested cased according to requirements

Conclusion

In conclusion I learned how Red Black Tree works.

ZOOM IN INSIDE PDF TO GO THROUGH THIS CODE

```java
package cs146.project4.shohin;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;

public class Dictionary extends RedBlackTree {

    //Inserts the dictionary into a RBT
    public static RedBlackTree generateDictionary() {
        RedBlackTree dictionaryRBT = new RedBlackTree();
        try {
            //Read the dictionary text file
            BufferedReader reader = new BufferedReader(new FileReader("/Users/insidious/San Jose State
University/2cmpe146/Project/4PrRBT/src/cs146/project4/shohin/dictionary.txt"));
            String line;
            //initiate starting time
            double startingTime = System.currentTimeMillis();
            while ((line = reader.readLine()) != null) //if next isnt null, continue to insert
            {
                dictionaryRBT.insert(line);
            }
            //ending time of insertion
            double endingTime = System.currentTimeMillis();
            System.out.println("Runtime of insertion into dictionary: " + (endingTime - startingTime)+"
ms");
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return dictionaryRBT;
    }

    //Creates an arraylist containing words from the poem
    public static ArrayList<String> wordsOfPoem() {
        ArrayList<String> listOfWordsFromPoem = new ArrayList<>();
        try {
            //reads the poem text file, pretty similar to before
            BufferedReader reader = new BufferedReader(new FileReader("/Users/insidious/San Jose State
University/2cmpe146/Project/4PrRBT/src/cs146/project4/shohin/poem.txt"));
            String line;
            double startingTime = System.currentTimeMillis();
            while ((line = reader.readLine()) != null) {
                //puts the words from the string line into an array of strings
                String[] changedLine = line.replaceAll("[^a-zA-Z ]", "").toLowerCase().split("\\s+");
                //Insert words into arraylist from the array of strings
                listOfWordsFromPoem.addAll(Arrays.asList(changedLine));
            }
            double endingTime = System.currentTimeMillis();
            System.out.println("Runtime of string to word: " + (endingTime - startingTime)+" ms");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return listOfWordsFromPoem;
    }

    public static void spellChecker() throws IOException {
        //initiates counter (how many words are nonexistent
        int counter = 0;
        double startingTime = System.currentTimeMillis();
        RedBlackTree dictionaryRBT = generateDictionary();
        ArrayList<String> listOfWordsFromPoem = wordsOfPoem();
        for (String s : listOfWordsFromPoem) {
            //if the dictionary does not contain the word, increase counter
            if (dictionaryRBT.lookup(Node.root, s) == null) {
                counter++;
            }
        }
        double endingTime = System.currentTimeMillis();
        System.out.println("Runtime of spellChecker: " + (endingTime - startingTime) +" ms");
        System.out.println("Number of words in poem: " + listOfWordsFromPoem.size());
        System.out.println("Number of mismatched/nonexistent words: " + counter);

    }

    public static void main(String[] args) {
        //tests spellChecker, runs it
        try {
            spellChecker();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

```java
package cs146.project4.shohin;

import org.junit.Test;

import java.io.IOException;

public class DictionaryJUnit {
    @Test
    public void testDictionary() throws IOException {
        Dictionary.spellChecker();
        //Number of words in poem: 254
        //Number of mismatched/nonexistent words: 96
    }
}
```

```java
package cs146.project4.shohin;

public class Node<Key extends Comparable<Key>> {

    public static Node<String> root;

    Key key;
    Node<String> parent;
    Node<String> leftChild;
    Node<String> rightChild;
    boolean isRed;
    int color;

    public Node(Key data) {
        this.key = data;
        leftChild = null;
        rightChild = null;
    }

    public static Node<String> getRoot() {
        return root;
    }

    public static void setRoot(Node<String> root) {
        Node.root = root;
    }

    public int compareTo(Node<Key> n) {    //this < that  <0
        return key.compareTo(n.key);    //this > that  >0
    }

    public boolean isLeaf() {
        if (this.equals(getRoot()) && this.leftChild == null && this.rightChild == null) return true;
        if (this.equals(getRoot())) return false;
        return this.leftChild == null && this.rightChild == null;
    }
    public boolean isLeaf(Node<String> n) {
        if (n.equals(root) && n.leftChild == null && n.rightChild == null) return true;
        if (n.equals(root)) return false;
        return n.leftChild == null && n.rightChild == null;
    }

}
```

```java
package cs146.project4.shohin;

public interface Visitor<Key extends Comparable<Key>> {
    /**
     * This method is called at each node.
     *
     * @param visitKeyNode the visited node
     */
    void visit(Node<Key> visitKeyNode);
}
```

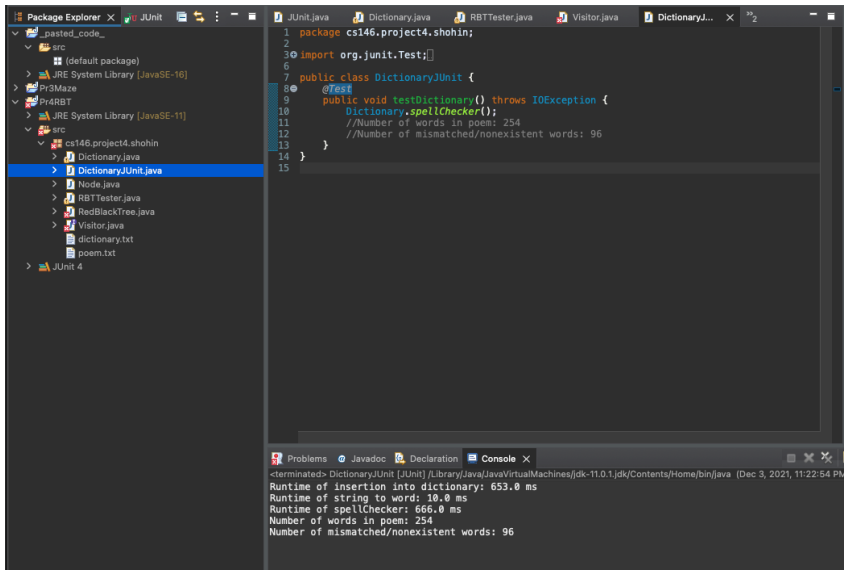
```java
package cs146.project4.shakira;

public class RedBlackTreeKey extends ComparableKeyre {

    public boolean isLeaf(Node<String> node) {
        if (node.equal(Node.root) && node.leftChild == null && node.rightChild == null) return true;
        if (node.equal(Node.root)) return false;
        return node.leftChild == null && node.rightChild == null;
    }


    public void visit(Node<Key> node) {
        System.out.println(node.key);
    }

    public void printTree() { //preorder: visit, go left, go right
        Node<String> currentNode = Node.root;
        printTree(currentNode);
    }

    public void printTree(Node<String> node) {
        System.out.print(node.key);
        if (node.isLeaf()) {
            return;
        }

        if (node.leftChild != null) {
            printTree(node.leftChild);
        }
        if (node.rightChild != null) {
            printTree(node.rightChild);
        }
    }

    // Grow a new node in the RB tree with data the parameter and color it red
// Find a new node in the RB tree with data the parameter and color it red
    public void addNode(String data) {  //this = that -->B   this = that -->R
        Node<String> rover = null; // holder
        Node<String> current = Node.root;
        Node<String> addedNode = new Node<>(data); // added node
        while (current != null) {
            rover = current;
            if (addedNode.compareTo(current) < 0) //if added node is less than current, go to current's
left child
            {
                current = current.leftChild;
            } else {
                current = current.rightChild; //if added node is greater than current, go to currents
right child
            }
        }
        addedNode.parent = rover;
        if (rover == null) {
            Node.root = addedNode;
        } else if (addedNode.compareTo(rover) < 0) {
            rover.leftChild = addedNode;
        } else {
            rover.rightChild = addedNode;
        }
        addedNode.leftChild = null;
        addedNode.rightChild = null;
        addedNode.color = 0;
        fixTree(addedNode);
    }


    public void insert(String data) {
        addNode(data);
    }

    //Search up a node in the RBT
    //Recursively searches for a node in RBT
    public Node<String> lookup(Node<String> current, String k) {
        Node<String> searchedNode = new Node<>();
        if (current == null || searchedNode.compareTo(current) == 0) {
            return current;
        }
        if (searchedNode.compareTo(current) < 0) {
            return lookup(current.leftChild, k);
        }
        return lookup(current.rightChild, k);
    }

    //Returns the sibling of a node in the RBT
    public Node<String> getSiblingNode(Node<String> n) {
        Node<String> parentOfN = n.parent;
        if (parentOfN != null) {
            if (parentOfN.compareTo(n) < 0) {
                return parentOfN.leftChild;
            } else if (parentOfN.compareTo(n) > 0) {
                return parentOfN.rightChild;
            }
        }
        return null;
    }

    //Returns the grandparent of a node in the RBT
    public Node<String> getGrandparent(Node<String> n) {
        Node<String> grandparent = getGrandparent(n);
        if (grandparent != null) {
            if (grandparent.compareTo(n) < 0) {
                return grandparent.leftChild;
            } else if (grandparent.compareTo(n) > 0) {
                return grandparent.rightChild;
            }
        }
        return null;
    }

    //Returns the grandparent of a node in the RBT
    public Node<String> getGrandparent(Node<String> n) {
        if (n.parent != null) {
            if (n.parent.parent != null) {
                return n.parent.parent;
            }
        }
        return null;
    }

    public void rotateLeft(Node<String> node) {
        //up = node's parent
        //where the RBT is set up so it is X X X f B C
        //   f   X
        //  B    X
        //  C    B C
        //rotating left turns it to
        //   X   X
        //   f   B
        //  B C  f C
        //  C    B
        //  C    C
        Node<String> rightChildOfNode = node.rightChild;
        node.rightChild = rightChildOfNode.leftChild;          //turns y's left subtree into x's
right subtree
        if (rightChildOfNode.leftChild != null) {
            rightChildOfNode.leftChild.parent = node;
        }
        rightChildOfNode.parent = node.parent;                 //links x's parents to y
        if (node.parent == null) {
            Node.root = rightChildOfNode;
        } else if (node == node.parent.leftChild) {
            node.parent.leftChild = rightChildOfNode;
        } else {
            node.parent.rightChild = rightChildOfNode;
        }
        rightChildOfNode.leftChild = node;                     //put x on y's left
        node.parent = rightChildOfNode;
    }

    public void rotateRight(Node<String> node) {
        //up = node's parent
        Node<String> leftChildOfNode = node.leftChild;
        node.leftChild = leftChildOfNode.rightChild;
        if (leftChildOfNode.rightChild != null) {
            leftChildOfNode.rightChild.parent = node;
        }
        leftChildOfNode.parent = node.parent;
        if (node.parent == null) {
            Node.root = leftChildOfNode;
        } else if (node == node.parent.rightChild) {
            node.parent.rightChild = leftChildOfNode;
        } else {
            node.parent.leftChild = leftChildOfNode;

        }
        leftChildOfNode.rightChild = node;
        node.parent = leftChildOfNode;
    }

    //Followed the pseudocode in the book, turned it into code
    public void fixTree(Node<String> current) {
        if (current == Node.root) {
            current.color = 1;
        } else if (current.color == 0 && current.parent.color == 0) {
            if (getAunt(current) == null || getAunt(current).color == 1) {
                if (current.parent == getGrandparent(current).leftChild && current ==
current.parent.rightChild) {
                    rotateLeft(current.parent);
                    current = current.parent;
                    fixTree(current);
                } else if (current.parent == getGrandparent(current).rightChild && current ==
current.parent.leftChild) {
                    rotateRight(current.parent);
                    current = current.parent;
                    fixTree(current);
                } else if (current.parent == getGrandparent(current).leftChild && current ==
current.parent.leftChild) {
                    current.parent.color = 1;
                    getGrandparent(current).color = 0;
                    rotateRight(getGrandparent(current));
                } else if (current.parent == getGrandparent(current).rightChild && current ==
current.parent.rightChild) {
                    current.parent.color = 1;
                    getGrandparent(current).color = 0;
                    rotateLeft(getGrandparent(current));
                }
            } else {
                current.parent.color = 1;
                getAunt(current).color = 1;
                getGrandparent(current).color = 0;
                current = getGrandparent(current);
                fixTree(current);
            }
        }
    }

    public boolean isEmpty(Node<String> node) {
        return node.key == null;
    }

    public boolean isLeftChild(Node<String> parent, Node<String> child) {
        //if child is left child
        return child.compareTo(parent) < 0;
    }


    public void preOrderVisit(Visitor<String> traverse) {
        preOrderVisit(Node.root, traverse);
    }


    private static void preOrderVisit(Node<String> node, Visitor<String> traverse) {
        if (node != null) {
            traverse.visit(node);
            preOrderVisit(node.leftChild, traverse);
            preOrderVisit(node.rightChild, traverse);
        }
    }


    public static void main(String[] args) {

        RedBlackTree RBT;
        RBT = new RedBlackTree();

        RBT.insert("B");
        RBT.insert("B");
        RBT.insert("A");
        RBT.insert("B");
        RBT.insert("D");
        RBT.insert("E");
        RBT.insert("C");
        RBT.insert("B");
        RBT.insert("D");
        RBT.insert("F");
        RBT.printTree();
        System.out.println(" is RBT");

        System.out.println(" ");

        RBT.insert("B");
        RBT.insert("B");
        RBT.insert("A");
        RBT.insert("B");
        RBT.insert("D");
        RBT.insert("E");
        RBT.insert("B");
        RBT.insert("D");
        RBT.insert("F");
        RBT.printTree();
        System.out.println(" is Rotated Left");

        RBT.rotateRight(Node.root);

        System.out.println(" ");

        RBT.printTree();
        System.out.println(" is Rotated Right");
        System.out.println(" ");

    }


}
```

```java
package cs146.project4.shohin;

import static org.junit.Assert.*;
import org.junit.Test;


public class RBTTester {


    public static double getRunTime(long startTime) {
        long endTime = System.nanoTime();
        return (endTime - startTime) / 1000000.0;
    }

    public static long getStartTime() {
        return System.nanoTime();
    }

    @Test
    //Test the Red Black Tree
    public void test() {


        RedBlackTree rbt = new RedBlackTree();
        rbt.insert("D");
        rbt.insert("B");
        rbt.insert("A");
        rbt.insert("C");
        rbt.insert("F");
        rbt.insert("E");
        rbt.insert("H");
        rbt.insert("G");
        rbt.insert("I");
        rbt.insert("J");
        assertEquals("DBACFEHGIJ", makeString(rbt));
        String str = "Color: 1, Key:D Parent: \n" +
                "Color: 1, Key:B Parent: D\n" +
                "Color: 1, Key:A Parent: B\n" +
                "Color: 1, Key:C Parent: B\n" +
                "Color: 1, Key:F Parent: D\n" +
                "Color: 1, Key:E Parent: F\n" +
                "Color: 0, Key:H Parent: F\n" +
                "Color: 1, Key:G Parent: H\n" +
                "Color: 1, Key:I Parent: H\n" +
                "Color: 0, Key:J Parent: I\n";
        assertEquals(str, makeStringDetails(rbt));

        rbt.printTree();

        long startTime = getStartTime();
        double runTime = getRunTime(startTime);
        System.out.println(" output has runtime of " + runTime + " ms");


    }

    //add tester for spell checker

    public static String makeString(RedBlackTree t) {
        class MyVisitor implements Visitor {
            String result = "";

            public void visit(Node visitKeyNode) {
                result = result + visitKeyNode.key;
            }

        }

        MyVisitor v = new MyVisitor();
        t.preOrderVisit(v);

        long startTime = getStartTime();
        double runTime = getRunTime(startTime);
        System.out.println("makeString method has runtime of " + runTime + " ms");

        return v.result;


    }

    public static String makeStringDetails(RedBlackTree t) {
        {
            class MyVisitor implements Visitor {
                String result = "";

                public void visit(Node visitKeyNode) {

                    //changed method to account for parent of n being null

//                      if(!(n.key).equals(""))
//                      result = result +"Color: "+n.color+", Key:"+n.key+" Parent: "+n.parent.key+"\n";


                    if (visitKeyNode.parent == null) {
                        result = result + "Color: " + visitKeyNode.color + ", Key:" + visitKeyNode.key
+ " Parent: " + "\n";
                    } else {
                        result = result + "Color: " + visitKeyNode.color + ", Key:" + visitKeyNode.key
+ " Parent: " + visitKeyNode.parent.key + "\n";
                    }

                }
            }


            MyVisitor v = new MyVisitor();
            t.preOrderVisit(v);

            long startTime = getStartTime();
            double runTime = getRunTime(startTime);
            System.out.println("makeStringDetail method has runtime of " + runTime + " ms");

            return v.result;
        }
    }



}
```

Instructions:

1. Downloaded Eclipse for Mac
2. Created Java Project
3. Created Package as written in the requirements.
4. Created RedBlackTree, Dictionary, DictionaryJUnit, RBTTester, Node, and Visitor classes
5. Before running JUnit, we exported JUnit 4 of what's required like Assert Equals, Assert True and etc.
6. Please Zoom In

This is separate Screenshot for Dictionary class JUnit that outputs number of words and mismatched words



And this is separate Screenshot for RBTester that outputs DBACFEHGIJ