Project #5
Red-Black Trees
due at 9pm, Wed 13 Apr 2016

1 Introduction

In this project, we’ll be implementing Red-Black trees. We’ll be constructing them from 2-3-4 trees, and performing insertions using both the top-down and bottom-up algorithms; however, I will not be requiring that you implement deletions.

In an effort to speed grading, I have implemented an automatic testing system. 70% of your grade will come from the automatic tests; 30% will come from a manual inspection - covering issues like style, comments, etc. I’m providing many testcases; be warned that we may be adding additional testcases that we do not provide to you.

You are encouraged to write your own testcases, and to share them with the rest of the class. The more tests we have, the better chance everybody has of catching subtle bugs!

2 Program Overview

In this project, you will implement a Red-Black Tree. You have complete flexibility about how to implement it internally, but you must conform to the external interface that I’ve defined.

I have provided two classes for you to use: Proj05Main implements the main() method for your program, and includes code to read testcases, execute the tests, and report the results. Node234 implements one node of a 2-3-4 tree. The starting point of every testcase will be to read a 2-3-4 tree from the testcase file, which main() will pass to your code; you will convert the 2-3-4 tree into an equivalent Red-Black tree.

To run the test program, you run Proj05Main, and pass any number of testcase files as command-line arguments. The main() method will read each file and execute the testcase; it performs a series of checks to confirm that each step completed successfully. If any step fails, the code will terminate the testcase, but will move on to run other testcases; at the end, it reports the number of testcases which passed.

Your task in this project is simply to implement RedBlackNode.java, which implements the Red-Black tree logic.

All of the files for this project are available at:
http://www.cs.arizona.edu/classes/cs345/spring16/projects/proj05_files
3 Program Requirements

Your program must fulfill the following requirements:

- You must implement a class named RedBlackNode; it may also implement additional classes, if desired.
- You must not modify the Proj05Main or Node234 classes.
- RedBlackNode must implement the methods detailed later in the spec, and compile against an unmodified Proj05Main.
- RedBlackNode must implement a Red-Black tree, which stores String keys and not allow duplicates.
- The .dot files (as returned by toDotFile()) must generate .png files on Lectura, and the files generated must be a reasonable picture of the Red-Black tree.

4 The RedBlackNode Class

You must implement a RedBlackNode class, and it must fulfill the interface that I define in this spec. (If you don’t, we won’t be able to compile your code.)

Your class must implement the following methods:

4.1 public static RedBlackNode buildFrom234(Node234)

This static method takes a single parameter: the root of a 2-3-4 tree. It returns a Red-Black tree which is exactly equivalent to the 2-3-4 tree; it must implement each node in the 2-3-4 tree with the appropriate widget (see the slides).

The parameter will never be null, so you do not have to implement any logic to deal with empty Red-Black trees.

2-3-4 nodes with either 1 key or 3 keys have only one possible widget; nodes with 2 keys have two possible digets. Your program must turn all 2-3-4 nodes with 2 keys into a left-leaning widget (that is, with the red node on the left).

However, you will enforce this only in this method; do not enforce it on insertions!

4.2 public String[] toArray()

This method must return an array of Strings, which is an in-order traversal of the entire tree. Each element in the array must contain the key at that node, plus the prefix "b:" (for black nodes) or "r:" for red.

For example, if we have the keys A,B,C arranged in a simple triangle shape, and all three nodes are black, then the elements of the array would be

b:A b:B b:C
However, if the tree had a black root, with two red children, then elements of the array would be

\[ r:A \quad b:B \quad r:C \]

4.3 \textbf{public String[]} \texttt{toArray\_preOrder()}

This method works just like the method \texttt{toArray()}, except that it must be a pre-order traversal of the tree.

4.4 \textbf{public String} \texttt{toDotFile(boolean isRoot)}

This must return the contents of a \texttt{.dot} file; \texttt{main()} will call this each time that we insert a new node, and will write the data out to disk.

We will be testing to make sure that these \texttt{.dot} files can be turned in to \texttt{.png} files on Lectura, and we will take a quick look to ensure that they seem like reasonable pictures. However, in this project, I do not have requirements about exactly how you write these files, or how the final pictures look. These files are mostly there for your use in debugging; make them good enough to be useful for you!

4.4.1 \textbf{HINT: Coloring Nodes in .dot}

You can set the color of a node in \texttt{.dot} with the \texttt{fillcolor} attribute (so long as you also set \texttt{style=filled}); you can set the color of the text with \texttt{fontcolor}. I encourage you to use this feature!

A black node, as generated by my solution to this project, looks like this:

\[
\text{vertex\_83473181 [label="3" style=filled fillcolor=black fontcolor=white];}
\]

while a red node looks like this:

\[
\text{vertex\_90061714 [label="2" style=filled fillcolor=red];}
\]

4.5 \textbf{public RedBlackNode} \texttt{insert\_td(String)}

This method inserts a new key into the Red-Black tree; it must return the root of the new tree. If we attempt to insert a duplicate key, this method must throw an \texttt{IllegalArgumentException}.

This method \textbf{must} use the top-down strategy for insertions.

4.6 \textbf{public RedBlackNode} \texttt{insert\_bu(String)}

This method is exactly like \texttt{insert\_td()}, except that it must use the bottom-up strategy.
5 Implementation Hints

The previous section describes the **required interface** for your `RedBlackNode` class. As stated elsewhere, you have complete freedom about how you implement the internal logic of this class; you may even use it as a “shell” which models the entire tree - with another class modeling the nodes!

Internally, there are a variety of ways you could implement it. However, I thought I’d offer a sketch of my implementation. If you want to emulate my implementation, that’s fine - if you want to ignore it, that’s fine, too. Everything in this Section of the spec is optional information.

My implementation doesn’t use parent pointers (even for bottom-up!); it does all of its rebalancing by having each node check all 4 of its grandchildren. This is not necessarily the most efficient implementation, but it works.

For bottom-up insertion (and, to handle the case of the one-step recursion in top-down insertion), I use a simple model of “perform fixup after returning.” That is, you can “recurse toward the root” without parent pointers simply by performing the cleanup **after** you return from the recursive insertion call.

Here is a **partial** list of the private methods and helper methods that I have in my solution.

- private `RedBlackNode _insert(String key, boolean isTopDown)`
- private `RedBlackNode fixupLL()`
- private `RedBlackNode fixupLR()`
- private `RedBlackNode fixupRL()`
- private `RedBlackNode fixupRR()`
- private `RedBlackNode pushdownBlackLevel()`
- private `RedBlackNode rotateLeft()`
- private `RedBlackNode rotateRight()`
- private static `boolean _isRed(RedBlackNode)`

Most of these methods return a `RedBlackNode` - the root of the tree after the modification is performed. Also note that my `rotate*()` methods keep track of colors, and modify them as appropriate.

5.1 What the Class Represents

I suggest (but don’t require) that you use this class like we did in `BSTNode` and `AVLNode` - where each object of this class models a single node in the Red-Black tree. However, if you want to do something else (like using this class as the tree itself, and having another class to model the nodes), I’ll allow that.
In this project, we’ll never have empty trees, and so you don’t have to worry about ever having the root of the tree be null; thus, there are multiple ways you can implement this.

Like we did in BSTNode (and I suggested for Project 3/4), this project requires that you use the “return the new root of the tree” model for the insert() method. I encourage you to use this model internally, throughout your implementation - but it’s not required. If you want, for instance, for RedBlackNode to model the entire tree, then it would be OK to simply return the same pointer (pointing to the tree itself), over and over and over.

6 The Testcases

Here’s a quick summary of the testcases:

- TEST.00-TEST.04 - converting simple 2-3-4 trees to Red-Black
- TEST.05-TEST.12 - testing basic rotations
- TEST.13-TEST.20 - testing insertion on slightly larger trees
- TEST.21 - testing the one-step recursion in top-down
- TEST.22 - testing that you reject duplicates
- TEST.50-TEST.50 - building a tree from the root down
- TEST.52-TEST.53 - inserting the alphabet, in order
- TEST.54-TEST.55 - inserting into a large, balanced tree

I’d love it if all of you came up with more testcases! The more we have, the better all of our code will be tested.

REMEMBER: This is not an exhaustive list of testcases! We will likely add more testcases that we won’t tell you about - and we’ll use those (along with these testcases) to test your code, and assign your grade.

7 Compilation Requirements

Your code must compile on Lectura, using javac. If you write your code in an IDE (like Eclipse), you may have added some package statements which will make your code not compile on Lectura - if so, make sure to remove them before you turn in your code! Code that does not compile will not be graded; if you have to turn in a replacement version to fix a compile issue, it will be treated as if it was late by one day (or one additional day, as the case may be).
8 Comments and Style

You must comment your code. Make sure that your comments are clear; they need to express both what you are trying to do, and how you plan to do it. Clear comments will make it easier for us to grade your code, and to give you partial credit if/when we find bugs.

Use good Java programming style, including good use of whitespace, consistent indentation, and meaningful variable names. Follow the Java variable naming conventions (start with lowercase, and camel case after that).

Each file should include a header comment, which includes:

- Java class name
- Our class and assignment name (CSc 345 Spring 16 - Project 3)
- Your name
- A description of the basics of the class

9 Turning in Your Program

You must turn in at least one file:

RedBlackNode.java

You may turn in additional files if you want, but any copy of Proj05Main.java or Node234.java that you turn in will be ignored.

Turn in the files using the assignment name

cs345_s16_proj5

9.1 Using turnin

Hopefully, you remember how to use turnin from previous CS courses. However, here are a few reminders:

- turnin can only be run from lectura.cs.arizona.edu; upload your files to lectura using SCP (or a thumb drive in the labs).
- To turn in files, use the command:

  turnin <assignment_name> <file1> <file2> ...

- To confirm that your files have been turned in correctly, check with the command:
turnin -ls <assignment_name>

- You may turn in as many times as you want; turn in one file at a time, or all of them together. You may also turn in the same file multiple times; turnin will only keep the most recent version.