Program 1: Red-Black Trees

CS 223

1 Introduction

Standard Binary Search Trees provide excellent performance if they are well-balanced, but run into trouble if nodes are inserted in "bad" orders. Red-Black trees get around this problem by emulating 2-3-4 trees, which have robust balance guarantees.

For this project you will implement a remove-less version of the `SymbolTable` *Abstract Data Type* (ADT) using a Red-Black tree. I am providing you with skeleton code and a test suite. More details on those in Section 2. Implementation details are covered in Section 3. Section 4 covers the use of the provided test suite, and Section 6 covers how to turn in your project.

2 Provided Code

This project is accompanied by a file named `rbsymboltable-handout.zip`. Inside, you will find four Java source files that will help you with your project. (You'll also find five other files, which can be safely ignored.) The Java source files are:

RBSymbolTable.java – Skeleton you'll fill in and then submit

SymbolTable.java – A Java Interface for a the Symbol Table ADT

TreePrinter.java – A Java class that produces images of trees (See Section 5)

AVLTests.java – JUnit tests for your AVL tree class

To get started, I recommend adding all four Java files to IntelliJ (or your development environment of choice). IntelliJ will report errors with the test file as it does not include JUnit in the set of available libraries by default. To fix this, expand the imports at the top of the file, hover over the import for `org.junit.Test`, and select "Add JUnit4 to classpath". You'll be prompted to install the library. Click OK and give it a moment to install.

Only changes made in `RBSymbolTable.java` will be graded. You are, however, welcome to add your own tests or whatnot. I have provided a simple `main` method which uses the TreePrinter and may help with testing. The JUnit tests are more extensive, and more indicative of what we'll run for grading.

3 Red-Black Tree Implementation

3.1 Storing Node Color

Each node needs to store its color in order to handle red-black balancing operations. You may accomplish this any way you desire, but I would suggest using an enumeration. If you include this definition in your class, and an appropriate field in your node class, you'll be able to refer to colors as `Color.RED` or `Color.BLACK`:

```
private enum Color {BLACK, RED};
```

It is also helpful, but not required, to include a method which can report the color of a node. The following method, placed in the `RBSymbolTable`, will accomplish that for you:

```
private static boolean red(Node n) { // is node red? (null nodes are black)
    return (n == null) ? false : (n.color == Color.RED);
}
```

It may also be helpful to modify the constructor for `Node`s to take a color, but this isn't required.

3.2 Red-Black Insert

Most of your work will be in implementing the `insert` method. Note that there is no insert helper in the skeleton code. It is easier to implement red-black insertion iteratively than recursively.

You must supply a complete *red-black* insert method. That is, your insert must perform relevant color and balance checks and take corrective actions as needed. Failure to maintain a balanced tree, or balancing using some other mechanism, will both be considered wrong. Insert methods that fail in some cases but not others may receive partial credit, depending on the severity and frequency of failures.

Hint: Don't try to implement insert all in one go. Instead, worth through the cases in the slides, making sure each works before continuing. The ordering of the provided tests can also help with this.

3.3 Rotation and Color Updates

I have provided rotation methods for you. They do not, and *should not* include color updates. Instead, handle color changes after the rotations are finished.

3.4 Serializing the BST

I have provided a basic serializing function, along with helper methods for using `TreePrinter` (See Section 5). At present, the serializer assumes all nodes are black, which you will need to do correct by changing `serializeHelper` a little. The method already contains a
texttt nodeColor variable, which is initialized to ":black". All you need to do is set the variable to ":red" when the node is red. There is a TODO in the code for this, and you should only need 1-2 lines of code to make it happen.

3.5 Encapsulation

You are welcome to add additional helper methods to your implementation. Only the following methods should be publicly visible:

1. `RBSymbolTable` constructor
2. `insert`

3. `search`
4. `remove`
5. `serialize`

All other methods and fields should be private.

4 Testing

I have provided you with JUnit test suites for your Red-Black tree. IntelliJ understands how tests work (once you add JUnit, anyway), and can manage running the tests for you. You can run all of the tests at once, or run them individually. There are 13 tests in total. Your submission must pass all 13 to receive full credit. Please note that you will need to fix the serialization code (Section 3.4) to pass most of the tests.

5 Tree Printer

The included TreePrinter class is designed to produce a nice, graphical representation of a tree. I've provided the `printTree` method in the `RBSymbolTable` class to make it easy to interact with. To use it, call `printTree` on an instance of your tree, and pass in the name of an output file (ending in .svg). It'll then use TreePrinter to generate a vector image with that name. The file should end up in the same directory as your source files, or in the overall project directory, depending on how your project is set up. IntelliJ can open SVG files for you.

6 Submitting your solution

You will be submitted exactly one file for this project: `RBSymbolTable.java`. Once you have implemented the project to your satisfaction, submit the file, uncompressed, to Autolab (found at `autolab.encs.vancouver.wsu.edu`). Your submission will be passed through automatic testing, so you should get feedback on functionality quickly.

6.1 General Style and Source Documentation

Your code must be neatly formatted, with reasonable indentation and good variable names. Code which is hard to understand due to issues of formatting, naming, or other concerns will be marked down.

You must provide comments in your code to aid in understanding it. Minimally, this needs to include a comment just before each method, describing what it does. You should also provide comments breaking your code into blocks working toward a similar purpose, e.g. "went left on insert".