Assignment 5
Due: Oct 28, 2021 at 11:59PM

1 Assignment 5 Specification

1.1 Implementation

This assignment has a very similar purpose to Assignment 4. The skeleton code at `https://classroom.github.com/a/dLuIHPSR` is the start of a program to schedule I/O requests to a hard disk. Each I/O request is for a specific track and sector. Your program must track all of the incoming requests and then, when told where the drive head is, return the n next or previous I/O requests. (This is the 'elevator' method of I/O scheduling). We will assume that requests can be ordered first by track and then by sector, and that each request is unique. Tracks and sectors are both represented as non-negative numbers.

The current implementation has a linked list that works poorly under common loads. In the previous assignment, you implemented a binary search tree and the methods to support it. The BST, as it turns out, also performs poorly under common loads. For this assignment, you will implement a balanced binary search tree using the red-black tree data structure.

The skeleton code has an abstract **LocationContainer** class. This class has methods to build a data structure, find values in it, and find the next or previous location. It also has two important parts of a binary tree implemented: `root` and `nil`. There is one class already implemented:

L Implements a linked list data structure

You implemented a binary search tree in the previous assignment already. You may copy that code into this project's **BST.java** file, though none of the tests rely on it. (If you use the ChartMaker, your BST will show appropriate runtimes.)

For this assignment, you will implement the methods in the remaining class: **RBTree.java** as a red-black tree. For sources for this assignment, you should use the lecture slides and the book (chapter 13). I recommend sticking closely to the pseudocode, and using the *nil* construct that the book suggests. (*nil* is implemented in **LocationContainer** for you already.) You might also find it useful to reuse or modify parts of the code you wrote for the binary search tree.

This skeleton code has a lot of moving parts. There is an abstract **LocationContainer** class, which **BST** and **RBTree** both inherit from. There is a supporting node class called **DiskLocation.java** that holds the location and all necessary node information. For this assignment, a new field has been added: **color**. The node also has a comparator method, isGreaterThan, that you should use. You should not modify any file besides **BST.java** and **RBTree.java**. The color information is stored as an enum type.

In **RBTree.java**, there is a setRed method that demonstrates how to assign a color to a **DiskLocation**. Use this method instead of a simple assignment whenever you are setting a node to red to guarantee that you never change *nil* to the wrong color.

1.2 Assumptions

The following are assumptions that you can make about the program behavior.

- Assume that FIND is only called on locations that were previously inserted.
- Assume that HEIGHT is only called when the tree has at least one node.
- Assume that the parameters to NEXT and PREV are in the tree.
- Assume that none of the parent/child references are set for a node when it is passed to INSERT
- Assume that `root` is `null` when you first construct a BST.

1.3 Testing

For this assignment, JUnit tests are provided in the **edu.wit.cs.comp2350.tests** package. The tests check if insertion of various sizes works, and checks the **find**, **insert**, **next**, **prev**, and **height** methods. You should test the robustness of your program with additional inputs, as I will be grading the assignment on more tests than I have supplied.

insert must work in order for **find** to pass the tests, and **find** must work in order for **next**/**prev** to pass the tests.

A **ChartMaker** class is included, which runs on both randomized and sorted track/sector requests. You can use this charts to verify that your red-black tree's runtime is what you expect.

2 Grading

Red-black tree: 100%