Assignment 4
Due: Oct 21, 2021 at 11:59PM

1 Assignment 4 Specification

1.1 Implementation

The skeleton code at `https://classroom.github.com/a/bs7CQI3B` is the start of a program to schedule I/O requests to the head/arm of 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 are 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 uses a linked list that works poorly under common loads. Implement a binary search tree and the methods to support it.

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*. Use these values instead of reimplementing your own. There is one class already implemented:

L Implements a linked list data structure

For this assignment, you will implement the methods in the remaining class: **BST.java** as a binary search tree. For sources for this assignment, you should use the lecture slides and the book (chapter 12). I would recommend sticking closely to the pseudocode, and using the *nil* construct that the book suggests. (*nil* is implemented in **LocationContainer** for you already.)

This skeleton code has a lot of moving parts. There is an abstract **LocationContainer** class, which **L** and **BST** both inherit from. There is a supporting node class called **DiskLocation.java** that holds the location and all necessary node information. It also has a comparator method, isGreaterThan, that you should use to compare DiskLocations to each other. You should not modify any file besides **BST.java**. You can add as many private supporting methods as you need.

The height of a tree is defined as the number of steps it takes to get from the root to the deepest child. This means a tree with a single node has a height of 0. A balanced tree with 7 nodes has a height of 2. Check out the size method in **LocationContainer** for an idea how to measure the height. Think of the height of the node as 1 + the height of its larger subtree.

The find method should return the DiskLocation in your BST that has the same track/sector values as the input parameter.

The next and prev methods should return *nil* if they are called with the largest/smallest value in the tree, respectively.

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. INSERT must work correctly before any of the other methods can function. INSERT's tests don't confirm parent references, so it's possible to pass the test without correctly setting nodes' parents. The tests check if insertion of various sizes works, and checks the NEXT, PREV, and HEIGHT methods. In order for NEXT and PREV to work, you must implement FIND first. *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.

A **ChartMaker** class is included, which will create a chart of runtimes of each data structure based on input size. You can use this chart to verify that your binary tree's runtime is what you expect.

2 Grading

Binary search tree: 100%