Assignment 5
AVL Trees

Release Date	Due Date
3/27/2018	4/17/2018 (hard deadline - no extensions)

Objectives

To learn how to implement height balanced AVL Trees.

Introduction

You are asked to write a program for a company that sells hard drives. You know AVL trees maintain efficient insertion, deletion, and lookup times as they all take O(log(n)) in both their average and worst cases. Therefore, you decide to implement an AVL Tree.

Problem Specification

Note: In this assignment you are NOT allowed to use Java's (or other languages') default tree/stack/queue/etc utilities.

Part A: Implementation

1) Start off by reading in data from "*data_tiny.csv*" and store them in an array of **HDTestData** objects (you may recall your hw3).
2) Create an **AVLNode** class that at least keeps track of **leftChild**, **rightChild**, **height**, and **key**. You may also need an attribute to store a pointer to the data itself.
3) Create an **AVLTree** class that implements **insert**, **delete**, and **search**. Each of which can be made up of multiple methods if needed.
    - Your trees (and nodes) **must** be implemented with generics. I.e. it can hold any kind of data type.
4) Create 4 AVLTrees that each sort their contents by a different column of the data file (namely, serial_number, model, capacity_bytes, and power_on_hours).
5) Print out each of the HDTestData objects and their node's height in each tree in *breadth first order* and *depth first order*. (Implemented using queues and stacks (or recursion), respectively.)

Part B: Testing

1) Read the data of "*data_main.csv*" and store it in a new array of **HDTestData** objects.
2) Once you have read in the file, randomly choose *n* objects from the array.
3) Insert each of your chosen test data objects into each of your 4 trees. Time your inserts and print the average insertion time.
4) Select *m* random HDTestData objects from your array and find the first occurrence of each of its attributes in their respective trees. Print the full HDTestData information and the node's height for each occurance you find.
5) Calculate and print the average time it takes to search in each tree.
6) Delete each object in your tree one by one. Time each deletion and print the average time.
7) Compare the time it takes to insert, search, and delete.

The values for "n" should be as large as possible. But at least run your program for n = 10, 100, 250, 500, 1000, 10000, and 100000. Print trees (i.e., item A5 above) only for n=10, 100 and 250.

The values for "m" should be as large as possible. But at least provide outputs for m = 10, 50 and 100.

Extra Credit

(1) In your output if the tree is of size n < 50, print out the actual tree representation as if you would draw it by hand. With root being the center and its children are to the left and right of it along with their keys. (+20)

(2) For above item A3, in addition to implementing operations **insert**, **delete**, and **search,** implement **split** and **concatenate** operations. For a given key x, **split(x)** splits the AVL tree into two AVL trees, one containing keys less than or equal to x and another containing keys greater than x. Return the two trees with root pointers .*TlessOrEqualX* and *TlargerThanX*, respectively. Given two trees T1 and T2 such that the largest key in T1 is less than the smallest key in T2, **concatenate(T1, T2)** concatenates the two trees into one AVL tree and returns the resulting AVIL tree's root pointer. Note that you may have to repeatedly rotate in the concatenate operation to get a reasonably height-balanced binary search tree. Show that these two operations work by printing the input and output trees using depth-first and breadth-first traversals of the trees for n=10, 20 and 50. (+30)

Clearly indicate in your readme file and at the beginning of main() which, if any, extra credit you have implemented.

Submission Requirements

Output Requirements

Your output should be printed to the console as well as a text file and should be ***clear and informative.***

Code Documentation

You must include documentation for your code. This means that you should include detailed information on the usage of your program as a whole, as well as each function/class within it. This can be accomplished via Javadoc or you can hand-type the documentation. ***This includes how to compile and run your program.***

Analysis Report

Write a brief report (submitted as a .pdf file) on your observations of comparing theoretical vs empirically observed time complexities. This report should include:

- A brief description of problem statement(s).
- Algorithm descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms.
- Theoretically derived complexities of the algorithms used in your code.
- Table(s) of the observed times.
- Plots comparing the measured times of small trees vs. bigger trees. (As n grows, how do

your measured times grow?)
  - Plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).

Coding Conventions and Programming Standards

You must adhere to all conventions in the CS 3310 Java coding standard. This includes the use of white spaces for readability and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions for naming files, classes, variables, method parameters and methods. Read the material linked from our class web-pages (in case you can't recall programming styles and conventions from your CS1 and CS2 courses).

Assignment Submission

Generate a .zip file that contains all your files, including:

- Signed Plagiarism Declaration
- Source code files
- The original data input files
- Your sample output files
- Documentation of your code
- Your analysis report

Don't forget to follow the naming convention specified for submitting assignments.