Programming Assignment #2
Binary Search Trees: Balanced vs. Optimal

Problem Statement

When all keys in a binary search tree (BST) are equally probable, the best possible BST, from the standpoint of average search time, is the one with minimum path length, namely a complete binary search tree. However, when some keys are known to be more likely to occur than others, the best possible BST will not necessarily be balanced. For this programming assignment, we design a few variants of BSTs: *Optimal Binary Search Tree (OBST)* and *Nearly Optimal Binary Search Tree (NOBST)* as well as the AVL and plain BSTs. For this assignment, we assume that all keys are words (or strings), and the frequency of a word is used as its weight.

Requirements & Implementation

- You are to write two Java classes that implement the plain BST and the AVL Tree. Skeleton files of the classes are provided at eTL. They are named `BSTskel.java` and `AVLskel.java`. You can copy them to your working directory and rename to `BST.java` and `AVL.java`. You should then provide implementations of all the methods defined in the class skeletons. You can add more methods as you wish, but are not allowed to remove or change any method defined in the class skeletons.
- The `BST` methods must meet the following requirements.
    1. The `insert()` method inserts a key into the tree using the plain BST insertion algorithm. If the same key exists already in the tree, increment its *frequency* by one without inserting the key itself into the tree. Otherwise, insert the key and set the *frequency* to one.
    2. The `find()` method probes the tree to find a search key. A Boolean value `true` is returned if the key is found. Otherwise, `false` is returned. In either case, the `find()` method increments the *access count* by one for all the nodes probed (*i.e.*, all the nodes on the search path from the root).
    3. The `sumFreq()` and `sumProbes()` methods return the frequency sum and the access count sum of all the keys in the tree, respectively. The `resetCounters()` method resets both the frequencies and access counts of all keys in the tree to zero.
    4. The `sumWeightedPath()` method returns the sum of weighted path lengths of the tree, which can be computed by $\sum_{i=1}^{n} w_i l_i$, where w_i and l_i are the weight and one plus the level of the i^{th} node, respectively.
    5. The `size()` method returns the number of keys in the tree. The `print()` method prints the keys in the tree in the increasing order of their values. Each key should appear on a separate line in the format of `[key:frequency:access_count]`.
    6. The `obst()` and `nobst()` methods convert the tree from a plain BST into an *Optimal BST (OBST)* and a *Nearly Optimal BST (NOBST)*, respectively. The frequencies of keys are used as weights.
    7. An NOBST can be constructed recursively in $\mathcal{O}(n \log n)$ time. First, the root node of an NOBST is chosen so as to minimize the difference in the weight sums of the left and right subtrees. Then, the root of each subtree is chosen in the same way recursively. (When a tie needs to be broken, make the right subtree heavier.)
- The `AVL` class is a subclass of `BST`. The AVL method must meet the following requirements.
    1. The `insert()` method inserts a key into the tree using the AVL insertion algorithm. If the same key exists already in the tree, increment its *frequency* by one without inserting the key itself into the tree. Otherwise, insert the key and set the *frequency* to one.
- Follow the directions given in the first assignment handout regarding 'no `main()` method.'