Introduction

I'm not in general in favour of asking you to implement behaviours or methods in data structures for which a library class exists. But, I'm going to make an exception for the case of binary search trees as understanding the fundamental properties of those trees and the manipulation of trees in general is such an important skill.

This lab contains a lot of material, counts double, and spans two weeks of labs (plus you have the mid-semester break). Half-credit may be available at the judgement of the demonstrators. Note there are some extensions at the end of this lab – while extremely valuable and (IMNSHO) interesting, they're not a part of the assessment per se.

Problem description

All the problems involve, or are related to, writing code to extend the provided `BST.java` class that represents binary search trees of strings and demonstrating through a test file provided that your implementations are working properly.

The *height* of a node in a tree is the length (in edges, i.e., links) of the longest path from that node to a leaf node. In particular, this means that the leaf nodes have height 0, and the root node is the highest node in the tree (perhaps a better definition or metaphor might have been found?)

The *size* of a binary search tree is the total number of nodes it contains. More generally, we could define the size of a node in a BST to be the total number of nodes in the subtree rooted at that node.

Problems

Problems marked (P) below require programming; those marked (A) require only answers.

- (P) Add a data field `size` to the inner `Node` class. Adjust the dynamic methods `add(String s)` and `delete(String s)` (as well as the private methods they may call) so that the `size` of any node in the tree is correctly maintained.
- (P) Add a method `size(String s)` that returns the size of the subtree rooted at `s` (or 0 if `s` is not in the tree).
- (P) Add a data field `height` to the inner `Node` class. Adjust the dynamic methods `add(String s)` and `delete(String s)` (as well as the private methods they may call) so that the `height` of any node in the tree is correctly maintained.
- (P) Add a method `height(String s)` that returns the height of the node whose key is `s` (or -1 if `s` is not in the tree).
- (P) In lecture 15, an algorithm to construct an "as balanced as possible" tree from a sorted array of `String` was described (recursively: make the midpoint of the list the root, and do the same to construct the left subtree from the things before it, and the right subtree from the things after it). Implement that method as `static BST makeBalanced(String[] dictionary)` inside the `BST` class.
- (A) How could you use that method to fully balance a given BST? That is, how would you code an instance method `void makeBalanced()` that makes a given BST as balanced as possible? It is generally not possible to assign to `this`, i.e., something along the line of `this = BST.makeBalanced(...)` is not going to work.
- (P) Add a method `next(String s)` that returns the first (in alphabetical order) string `t` contained in a BST that is greater than `s`. It should return `null` if there is not such a node.
- (A) How might the code for a method `private Node next(Node n)` that returns the next `Node` of a BST (i.e., the node whose key is the least key of the tree greater than the key at the given node – which might be `null`) differ fundamentally from the previous method? (for the purposes of increased efficiency) – in fact, doesn't it already exist?

Extension and reflection

- Using `size` and `height` convince yourself by means of experiment that if we construct a BST by adding strings in random order then it tends to be quite well-

balanced. What is the expected ratio between its height and the optimal height which would be produced by `makeBalanced`?

- Read up about `Iterator` and `Iterable` in Java. If `t` is a BST how could you make the following code legal?

  ```
  for(String s : t.between(lo, hi)) { ... }
  ```

  Here `lo` and `hi` should be strings and the effect of the code should be to loop over every string `s` contained in the BST that was after `lo` and before `hi` in alphabetical order. It should not be necessary to generate the list of all such strings i.e., only $O(1)$ extra memory should be required. The ideas underlying the coding of a `next` method as above would certainly form part of such a utility.