CSE 341, Spring 2008, Assignment 6
Due: Wednesday 28 May, 8:00AM

Last updated: May 18

Overview: This assignment has two parts. Problems 1–4 involve defining Ruby classes for trees of strings. Problem 5 involves extending an anagram-finder written in Ruby.

1. Define 2 classes `Leaf` and `BinaryNode` with the methods described below. An instance of either class is a "tree of strings" — a `Leaf` has one string and a `BinaryNode` has no string itself but has two smaller "trees of strings." Sample solution is about 40 lines, many of which are just `end`.

   (a) `Leaf`'s `initialize` takes one argument (assumed to be a string, no need to check).

   (b) `BinaryNode`'s `initialize` takes two arguments, both assumed to be "trees of strings." These are the node's children.

   (c) `concatAll` takes no arguments and returns a single string that is all of a tree's strings (i.e., all strings of all tree descendants) concatenated together in left-to-right order.

   (d) In `BinaryNode`, define a *class method* (like a Java static method) `self.firstAlphabetical` that takes two strings and returns the string that comes first alphabetically. The `casecmp` method already in the `String` class makes this *very* easy.

   (e) `firstAlphabetical` takes no arguments and returns the string in the whole tree that comes first alphabetically.

   (f) `iterate` takes one argument of class `Proc` (i.e., something produced by `lambda {|x| ...}`) and calls its argument with each string in the tree.

   (g) In `BinaryNode`, define a *class method* `self.concatAll` that takes one argument, a "tree of strings," and returns all its strings concatenated together. Do *not* use the tree's `concatAll` instance method. Instead, use its `iterate` method. Hint: Use a local variable that starts with the empty string and gets imperatively updated to a longer string during the iteration.

2. Define a class `NaryNode` that is like `BinaryNode` except it can have any positive number of children. Sample solution is about 20 lines. Note:

   - `intialize` should take an array of trees. It should raise an error if the array's length is 0. Else it should store a *copy* of the array in a field. Each tree in the array is one of the node's children.
   - For `concatAll`, `firstAlphabetical` and `iterate`, use the `each` method of the `Array` class (or other `Array` methods taking blocks if you prefer) so your answers are at most a few lines long.

3. Now suppose you get tired of using `Leaf.new` all the time when building trees. Make it so that you can put strings in your trees directly rather than using the `Leaf` class at all. Do this by adding `concatAll` and `firstAlphabetical` methods to the built-in `String` class. Hint: The solution is perhaps a "trick" but *extremely* short — just think about what these methods should return. Do not bother adding an `iterate` method.

4. In a comment in your code, answer each of these questions in a few English sentences:

   (a) If you built a tree using just the `Leaf` and `BinaryNode` classes but you put integers at each leaf instead of strings, what would happen if you called the tree's `concatAll` method? Why?

   (b) If you used integers as in the previous problem but part of your tree was built with `NaryNode`, what would happen if you called the tree's `concatAll` method? Why?

   (c) Why does `NaryNode`'s `intialize` method make a copy of its argument? What could happen if it did not?

   (d) Why might adding methods to the `String` class be a poor design choice in a large application?

5. The top-level method `anagrams` provided to you takes two strings, a word and a filename. It prints every line of the file that is a permutation of the letters in the word, i.e., an anagram. It uses the provided `LetterCount` class. Your job will be to modify this code so that it prints all anagrams that can be formed with one *or two* lines in the file.

   One-line anagrams should be printed as-is (that's what the provided code already does). Two-line anagrams should be printed on one line (i.e., two lines from the file become one printed line) with a space between the two lines. For example, if the file had lines containing `foo` and `bar` and the word was `ofobar`, you should print either `foo bar` or `bar foo`, but not both (you can pick what order the two lines are printed). As a minor point, you can use the same line twice, so for `ofofoo`, you should print `foo foo` if the file has a line `foo`.

   While this problem requires only about 15 total lines of additional code, it also purposely requires you to determine how some of the code provided to you works in order for you to figure out how change it.

   A fun file to use for testing is `/usr/share/dict/words` on attu, though this file is *very* large (it has every English word on its own line plus many other things — it is used for spell-checking) and Ruby is not fast. The course website has some example outputs using this file.

   How to do it:

   (a) Add a method `sum` to `LetterCount` that produces a new `LetterCount` that has the sum of the letter counts for `self` and its argument. This should make sense after reading the code or trying it on some examples to understand what a `LetterCount` represents. Hints:
      - This is easier than the `difference` method provided to you because `sum` always succeeds. Sample solution is 7 lines.
      - The hash in a `LetterCount` returns 0 for any key not in the hash. This is convenient here.

   (b) Edit the body of the `anagrams` method. In addition to the code already there, do the following:
      - Use an array to remember the previous lines that could possibly be combined with a second line to produce an anagram. This array will only grow. Have each element be another array with two elements: the text of the line and a `LetterCount` for the line. Do not put every line in this array; just lines that could be combined to form an anagram.
      - After adding a new element to this array, see if it can be combined with any of the previous elements to produce an anagram, and if so then print it out. Note that the `sum` method will prove useful here, along with `difference` and `all_zeros`.

      Sample solution added a total of 8 lines and changed none of the lines already there.

6. **Challenge Problems:** You can receive full challenge-problem credit for doing 2 of the following 3 (and partial credit for 1).
   - The versions of `concatAll` so far are inefficient in that for a tree with n strings, they create n different strings as intermediate results, each one longer than the previous one. Add different methods to produce the same result without creating all the intermediate strings. Hints: Make two passes through the tree. To make a string of length `len`, use `"x"*len`.
   - Provide another version of your anagram-finder that works for any number of words, not just 1 (like the code provided to you) or 2 (like your solution to problem 5).
   - Reimplement your anagram-finder in Scheme or ML and write a paragraph or two about the code's similarities and differences from the Ruby version.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw6.rb`, where `lastname` is replaced with your last name.
- The first line of your `.rb` file should be a Ruby comment with your name and the phrase `homework 6`.
- Go to `https://catalysttools.washington.edu/collectit/dropbox/djg7/2125` (link available from the course website), follow the "Homework 6" link, and upload your file.