Lab 3

Due Oct 20, 2017

October 4, 2017

1 Description

In this lab, you will use your LetterCount class and understanding of recursion to implement code capable of identifying all of the *anagrams* of a given word or phrase, where an anagram is the rearrangement of the letters in a word or phrase into other words. For example, "windmill air hex" is an anagram of "William Hendrix."

2 Specifications

The first task for your program will be to read a text file containing a dictionary of English words. This file will be named "dictionary.txt", and it will be located in the same directory as your program. (A sample dictionary file has been provided.) This file will contain a number of words, one on each line. You should prompt the user to "Enter a word or phrase: ", then print out all collections of dictionary words that are anagrams of the given word or phrase. Separate words with a space, and print one anagram per line. After printing the anagrams, terminate the program.

3 The anagram-finding function

Your function for finding anagrams should operate recursively. Basically, given a target string (or LetterCount) and a dictionary of words to form anagrams from, your function should iterate through the dictionary until it finds a word whose letters could be subtracted from the current target. Each time you find a word that can be subtracted from the target, add this word to your anagram and recursively find all of the words that can be made from the other letters. When you have matched all of the letters in the target phrase, add the current anagram to the output and return.

If this function appeared in the context of a larger program, it would probably be best to return a list of all anagrams, but for the purposes of this lab, you can just print out the anagrams as you discover them.

4 Recursive example

Consider finding the anagrams of the word “braise.” One such anagram is “sea rib.” Your recursive function should remove “sea” from “braise,” leaving “bri.” It should recurse on “bri” and remove “rib.” At this point, all letters in “braise” have been matched, so “sea rib” should be output. “Rib” can also be removed from “braise,” leaving “ase,” so “rib sea” would also be output.

While other words, such as “sir,” can be removed from “braise,” none of these result in an anagram. (When removing “sir,” we’re left with “bae,” which is not in our dictionary.)

5 Implementation recommendations

Your anagram function needs to keep track of at least 3 things as it’s running: the letters left in the target string, the dictionary of words, and the current anagram. The remaining letters should be stored in a LetterCount object or LetterCount reference in order to add and remove letters easily. The dictionary of words should have two parts: the collection of words (like “sea” and “rib”) and LetterCount objects associated with these words. (It may be easiest to store the strings and LetterCount objects in parallel arrays.) You can maintain the current anagram as a string (difficult) or as a vector of strings or indices (integers). These values should be stored in global variables (bad programming practice), function arguments (recommended), or static local variables, so that you can access and modify them as the function recurses.

My implementation of this function accept 4 parameters, though you may choose to implement yours differently: a `LetterCount&` representing the letters to match, a `const vector<LetterCount>&` representing the dictionary of words to form anagrams from, a `vector<int>&` representing the indices of current words that have been matched, and a `vector<vector<int>>&` representing all of the anagrams that have been found so far.

6 Submission instructions and grading

You should submit a cpp file containing your source code. This code may use the LetterCount objects defined in Lab 2 (`#include "lettercount.h"`), and it should be able to print out anagrams as described in the sections above when compiled. Do *not* submit headers or source for Lab 2.

Your program will be tested against a number of inputs, and your grade will be determined by how many of the anagrams you identify for these inputs. Programs that do not compile will receive no credit.

7 Testing your code

When testing this function, it can be fun to enter your name or some longer phrases. However, as the length of the input grows, the number of anagrams may increase exponentially, so it's important when testing to choose inputs without too many letters.[1]

Based on the dictionary provided, "recognize" has "recognize" as its only anagram. "Braise" has two anagrams: "sea rib" and "rib sea." (These are really the same anagram, just in a different order.)

"Xenophobes" has two main anagrams: "she open box" and "shoe pen box," along with all other arrangements of these words.

"Make an A" has no anagrams, while "Don't make an F" has several: "fan tank mode," "nod tank fame," and all of their rearrangements.

Lastly, despite having an x, "William Hendrix" has 648 anagrams when you count different orders of the same words.

[1]Also, our code could be made faster by using a data structure known as a prefix tree, but we haven't studied these.