Recursion – Anagrams

**Purpose**

The lab was designed to teach you more about parameters, recursion, and backtracking.

**Program Shell**

AnagramMain.java

Anagrams.java

LetterInventory.java – in jar

**Description**

Attached is a jar file with the following LetterInventory class:

|  |  |
| --- | --- |
| Method | Description |
| LetterInventory(String data) | Constructs an inventory (a count) of the alphabetic letters in the given string, ignoring the case of letters and ignoring any non-alphabetic characters |
| int get(char letter) | Returns a count of how many of this letter are in the inventory. Letter might be lowercase or uppercase (your method shouldn’t care). If a nonalphabetic character is passed, your method should throw an IndexOutOfBoundsException |
| void set(char letter, int value) | Sets the count for the given letter to the given value. Letter might be lowercase or uppercase. If a nonalphabetic character is passed, your method should throw an IndexOutOfBoundsException |
| int size() | Returns the sum of all of the counts in this inventory. This operation should be “fast” in that it should store the size rather than having to compute it each time this method is called. |
| boolean isEmpty() | Returns true if this inventory is empty (all counts are 0). This operation should be fast in that it should not need to examine each of the 26 counts when it is called. |
| String toString() | Returns a String representation of the inventory with the letters all in lowercase and in sorted order and surrounded by square brackets. The number of occurrences of each letter should match its count in the inventory. For example, an inventory of 4 a’s, 1 b, 1 l and 1 m would be represented as “[aaaablm]”. |
| LetterInventory add(LetterInventory other) | Constructs and returns a new LetterInventory object that represents the sum of this letter inventory and the other given LetterInventory. The counts for each letter should be added together. |
| LetterInventory subtract(LetterInventory other) | Constructs and returns a new LetterInventory object that represents the result of subtracting the other inventory from this inventory (i.e., subtracing the counts in the other inventory from this object’s counts). If any resulting count would be negative, your method should return null. |

Anagrams

An anagram is a word or phrase made by rearranging the letters of another word or phrase. For example, the words “midterm” and “trimmed” are anagrams. If you ignore spaces and capitalization and allow multiple words, a multi-word phrase can be an anagram of some other word or phrase. For example, the phrases “Clint Eastwood” and “old west action” are anagrams.

In this assignment, you will create a class called Anagrams that uses a dictionary to print all anagram phrases of a given word or phrase. You will use recursive backtracking to implement your algorithm. You are provided with a client program AnagramMain.java that prompts the user for phrases and then passes those phrases to your Anagrams object. It asks your object to print all anagrams for those phrases.

Anagrams should have the following constructor and 2 public instance variables (for testing):

List<String> dictionary, prunedDictionary;

public **Anagrams**(List<String> dictionary)

This constructor should initialize a new Anagrams object that will use the given list as its dictionary. You should not change the list in any way. You may assume that the dictionary is a nonempty collection of nonempty sequences of letters and that it contains no duplicates.

Anagrams should also implement the following method:

public void **print**(String text, int max)

This method should use recursive backtracking to find combinations of words that have the same letters as the given string. It should print all combinations of words from the dictionary that are anagrams of text and that include at most max words (or an unlimited number of words if max is 0) to System.out. You should throw an IllegalArgumentException if max is less than 0.

Implementation Details

Using LetterInventory

An important aspect of the recursive backtracking solutions is separation of the recursive code from the code that manages low-level details of the problem. In this assignment, you will follow a similar strategy. In the anagrams problem, the low-level details involve keeping track of various letters and figuring out when one group of letters can be formed from another

group of letters. Review the LetterInventory class specification to remind yourself of the available methods. The “subtract” method of the LetterInventory class is the key to solving this problem. For example, if you have a LetterInventory for the phrase “george bush” and ask if you can subtract the LetterInventory for “bee”, the answer is yes, because every letter in the “bee” inventory is also in the “george bush” inventory. Since null is not returned, you need to explore this possibility. Of course, the word “bee” alone is not enough to account for all of the letters of “george bush”, which is why you’d want to work with the new inventory formed by subtracting the letters from “bee” as you continue the exploration.

print Algorithm

Your print method must produce the anagrams in the same format as in the example execution below. You are required to solve this problem by using recursive backtracking. In particular, you should write a recursive method that builds up an answer one word at a time. On each recursive call, you should search the dictionary from beginning to end and to explore each word that is a match for the current set of letters. The possible solutions should be explored in dictionary order. For example, in deciding what word might come first, you are to examine the words in the same order in which they appear in the dictionary. Note that the dictionary might not be sorted ! For any given phrase, you can reduce the dictionary to a smaller dictionary of “relevant” words. A word is relevant if it can be subtracted from the given phrase. Only a fraction of the dictionary will, in general, be relevant to any given phrase. So, reducing the dictionary before you begin the recursion will allow you to speed up the searches that happen on each recursive invocation. To implement this, you should construct a short dictionary for each phrase you are asked to explore that includes just the words relevant to that phrase. You should do this once before the recursion begins–not on each recursive call. You may continue to prune this smaller dictionary on each recursive call, but keep in mind that it is not required and it will make the code more difficult to write. If you decide to prune on each recursive call, clearly document it.

Full Example Walk-Through

Suppose we have the following dictionary, and we ask the program to print all anagrams of “eleven plus two”: demo.txt

one

plus

potato

twelve

won

zebra

Step 1: Prune the Dictionary

The letters in “eleven plus two” do not support all the words in the dictionary. In particular, “potato”

and “zebra” contain letters (“a” and “z”, respectively) that are not in elevenplustwo. So, we prune the dictionary to the following:

one

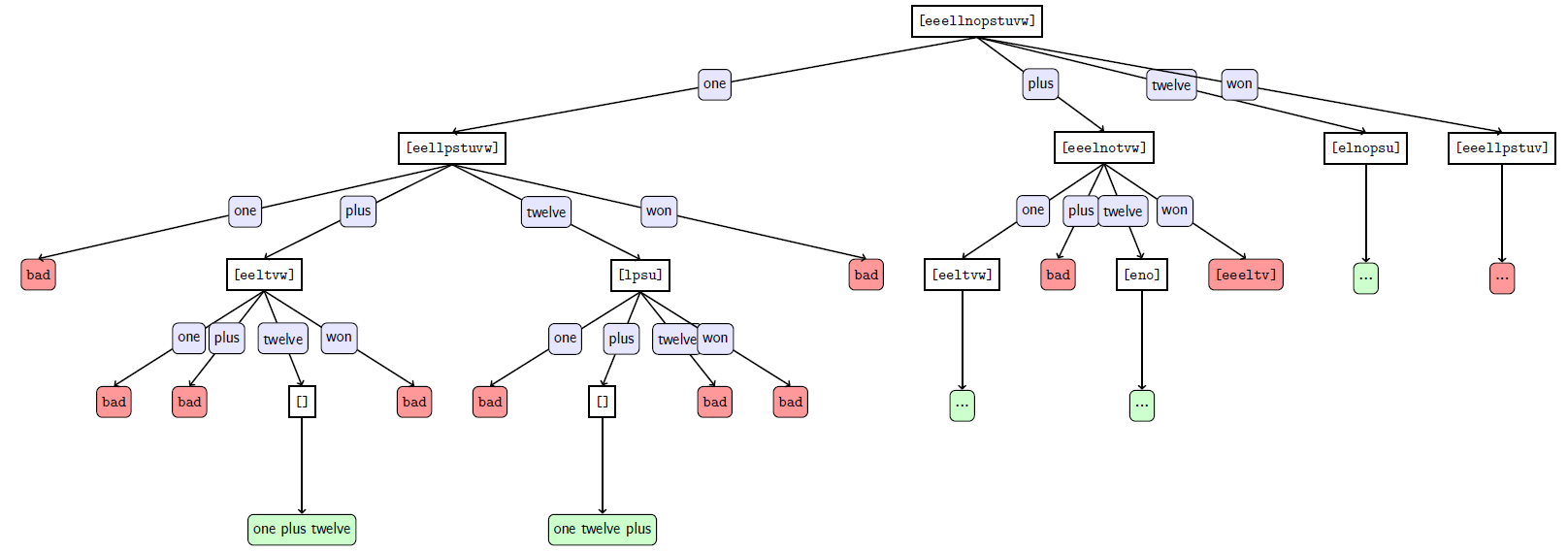
plus

twelve

won

Step 2: Find the Words

Now that we’ve pruned our dictionary, we know what our choices are at each step. (Namely, they’re the remaining words in the dictionary.) So, we go through our words recursively, keeping track of “the letters we have left”:



Example Execution

Welcome to the CS Anagram Solver!

What is the name of the dictionary file? demo.txt

Phrase to Scramble (return to quit)? eleven plus two

Max Words to Include (0 for no max)? 0

one plus twelve

one twelve plus

plus one twelve

plus twelve one

twelve one plus

twelve plus one

Phrase to Scramble (return to quit)?