PA12 Anagrams

Due: Wednesday 12/09 by 11:30PM

Submission:

- PA12Main.java

Overview

This assignment has multiple goals:

- To practice recursive backtracking
- Decomposition: Using a provided class to help solve a problem
- Using data structures

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 program called PA12Main.java that uses a dictionary to find all anagram phrases that match a given word or phrase. To use the program, command line options will be provided to indicate a dictionary file, phrase with no spaces to find anagrams of, and a limit on the number of words in the found anagrams (or 0 to indicate no limit).

For example the command line options:

```
dict1.txt barbarabush 0
```

should produce the following:

```
Phrase to scramble: barbarabush

All words found in barbarabush:
[abash, aura, bar, barb, brush, bus, hub, rub, shrub, sub]

Anagrams for barbarabush:
```

```
[abash, bar, rub]
[abash, rub, bar]
[bar, abash, rub]
[bar, rub, abash]
[rub, abash, bar]
[rub, bar, abash]
```

See the .out files in the public test cases for more examples. The command line options are encoded in the .out files similar as they were for previous PAs except the extension for the input files has changed. For example, the file dict3-defleppard-0.out indicates usage of the following command line arguments:

```
dict3.txt defleppard 0
```

Assignment

Your program should obtain the dictionary file name, phrase, and max number of words in each possible anagram from the array of strings passed to main. Your program should read all of the words from the dictionary file and use them to generate the required anagrams. You can assume that the dictionary file will contain one word per line. You can also assume the command-line input will be well formed.

You will want to find all possible words that can be found by using subsets of letters from the provided phrase. For example, in the phrase hairbrush the following words from dict1.txt can be found:

```
All words found in hairbrush:
[bar, briar, brush, bus, hub, huh, hush, rub, shrub, sir, sub]
```

You should use recursive backtracking to find and print all anagrams that can be formed using all of the letters of the given phrase. Each phrase should include at most max words. If max=0 then all possible words should be specified. Your output should exactly match the provided examples, including order and formatting. For example, if your anagram solver is using the dictionary corresponding to dict1.txt and a user types the phrase hairbrush, with a max of 0 your program should produce the following output:

```
[bar, huh, sir]
[bar, sir, huh]
[briar, hush]
[huh, bar, sir]
[huh, sir, bar]
[hush, briar]
[sir, bar, huh]
[sir, huh, bar]
```

If your anagram solver is using the dictionary corresponding to dict1.txt and the user types the phrase hairbrush and a max of 2, your program should produce the following output:

```
[briar, hush]
[hush, briar]
```

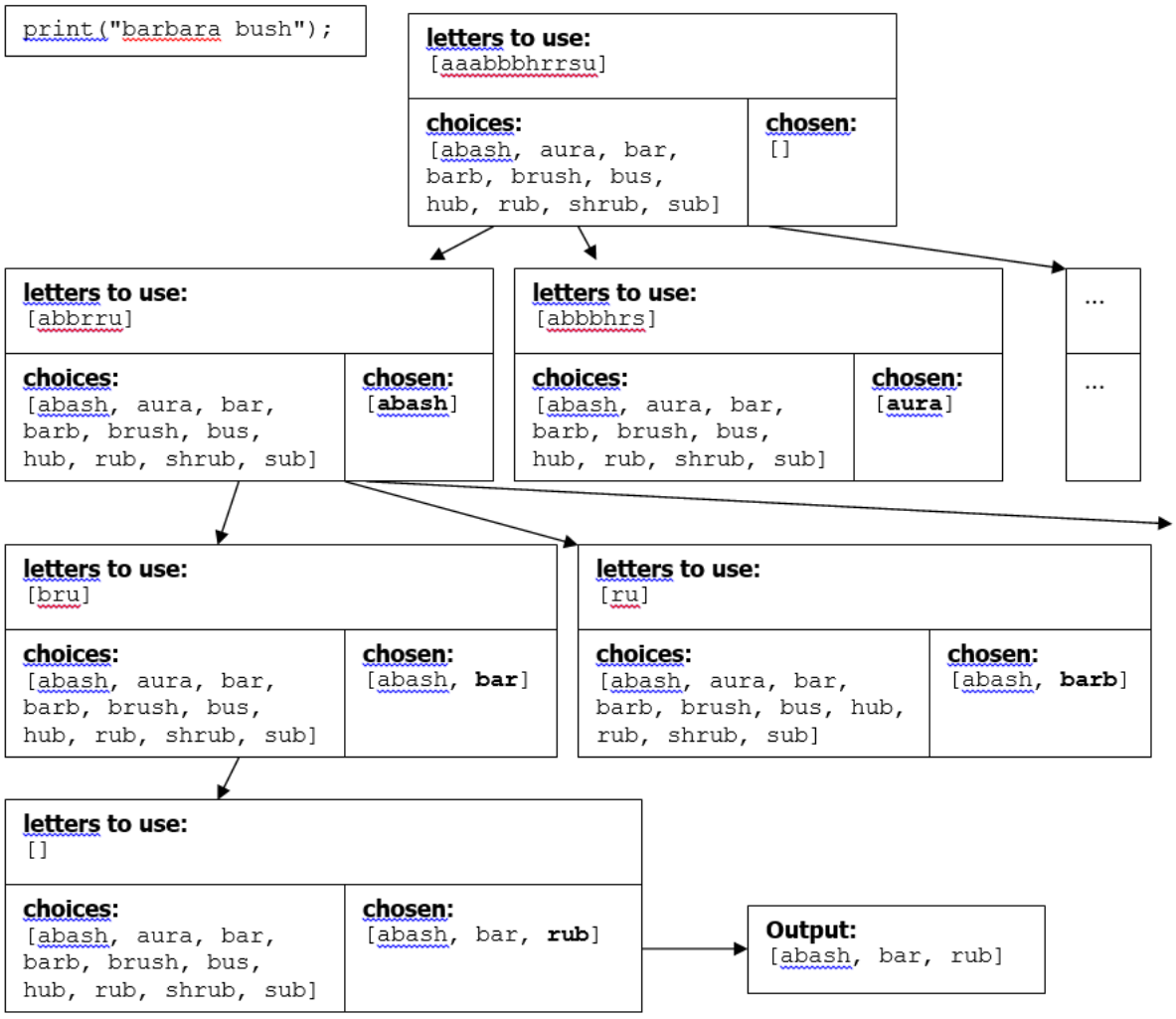
Algorithm

Generate all anagrams of a phrase using recursive backtracking. Many backtracking algorithms involve examining all combinations of a set of choices. In this problem, the choices are the words that can be formed from the phrase. A "decision" involves choosing a word for part of the phrase and recursively choosing words for the rest of the phrase. If you find a collection of words that use up all of the letters in the phrase, it should be printed as output.

Part of your grade will be based on efficiency. One way to implement this program would be to consider every word in the dictionary as a possible "choice." However, this would lead to a massive decision tree with lots of useless paths and a slow program. **Therefore for full credit, to improve efficiency when generating anagrams for a given phrase, you should first find the collection of words contained in that phrase and consider only those words as "choices" in your decision tree. You should also backtrack immediately once exceeding the max.**

The following diagram shows a partial decision tree for generating anagrams of the phrase barbarabush. Notice that some paths of the recursion lead to dead ends. For example, if the recursion chooses aura and barb, the letters remaining to use are [bhs], and no choice available uses these letters, so it is not possible to generate any anagrams beginning with those two choices. In such a case, your code should backtrack and try the next path.

One difference between this algorithm and other backtracking algorithms is that the same word can appear more than once in an anagram. For example, from barbara bush you might extract the word bar twice.



LetterInventory

An important aspect of simplifying the solution to many backtracking problems is the separation of recursive code from code that manages low-level details of the problem. We have seen this in several of our backtracking examples, such as 8 queens (recursive code in Queens.java, low-level code in Board.java). You are required to follow a similar strategy in this assignment. The low-level details for anagrams involve keeping track of letters and figuring out when one group of letters can be formed from another. We are providing you a class called LetterInventory to help with this task.

A LetterInventory object represents the count of each letter from A-Z found in a given string (ignoring whitespace, capitalization, or non-alphabetic characters). For example, a LetterInventory for the string "Hello there" would keep count of 3 Es, 2 Hs, 2 Ls, 1 O, 1 R, and 1 T. The toString of this inventory would be [eeehhllort].

You can add and subtract phrases from a LetterInventory and ask whether an inventory contains a phrase. For example, adding "hi ho" to the above inventory would produce [eee-

hhhhilloort]. Subtracting "he he he" from this would produce [hilloort]. If we asked whether this inventory .contains("tool"), the result is true.

Constructor/Method	Description
public LetterInventory(String s)	constructs a letter inventory for the given string
public void add(LetterInventory li) public void add(String s)	adds the letters of the given string/inventory to this one. Note these are two separate methods overloaded. You can use the add method passing in either a letterInventory or a string.
public boolean contains(LetterInventory li) public boolean contains(String s)	returns true if this inventory contains all letters at least as many times as they appear in the given string/inventory
public boolean isEmpty()	returns true if the inventory contains no letters
public int size()	returns the total number of letters in the inventory
public void subtract(LetterInventory li) public void subtract(String s)	removes letters of the given string/inventory from this one; throws IllegalArgumentException if not contained
public String toString()	string version of inventory, such as [eehhllort]

Error Handling

All of the inputs will be correctly formed for this assignment.

Hints

- Your program should produce the anagrams in the same format as in the expected output files. The easiest way to do this is to build up your answer in a list, stack, or other ordered collection. Then you can println the collection and it will have the right format.
- Read and understand the LetterInventory class.
- Several different dictionary files are provided. We recommend initially using a very small dictionary dict1.txt to make testing easier. But once your code works with this dictionary, you should test it with larger dictionaries such as the provided dict2.txt and dict3.txt.
- One difficult part of this program is limiting the number of words that can appear in the anagrams. We suggest you do this part last, initially printing all anagrams regardless of the number of words.
- Start early!

Grading Criteria

We are providing testcases for this PA. We will also have our own private grading testcases.

We encourage you to write your own JUnit testcases to ensure your classes work properly, but we will not be collecting or grading these test files.

Your grade will consist of similar style and code clarity points we have looked for in earlier assignments.

Write your own code. We will be using a tool that finds overly similar code. Do not look at other students' code. Do not let other students look at your code or talk in detail about how to solve this programming project. Do not use online resources that have solved the same or similar problems. It is okay to look up, "How do I do X in Java", where X is indexing into an array, splitting a string, or something like that. It is **not** okay to look up, "How do I solve {a programming assignment from CSc210}" and copy someone else's hard work in coming up with a working algorithm.