CS 132, Winter 2022
Programming Assignment #4: Anagrams (20 points)
Due Tuesday, February 8, 2022, 11:59 PM

This program focuses on recursive backtracking. Turn in files named `Anagrams.h` and `Anagrams.cpp` from the Project section of the course web site. You will need your `LetterInventory` files from project 3 and support files `anagramMain.cpp` and various dictionary text files. All of the instructor provided support files as well as empty files to copy your project 3 code into are included in the starter zip linked on the Projects page of the course website.

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 find all anagram phrases that match a given word or phrase. You are provided with a client program `anagramMain` 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. Below is a sample log of execution, wrapped into two columns (user input is underlined):

```
Welcome to the CS 132 anagram solver.
What is the name of the dictionary file? dict1.txt

Phrase to scramble (Enter to quit)? barbara bush
All words found in "barbara bush":
[abash, aura, bar, barb, brush, bus, hub, rub, shrub, sub]

Max words to include (Enter for no max)?

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


Phrase to scramble (Enter to quit)? john kerry
All words found in "john kerry":
[he, her]

Max words to include (Enter for no max)?



Phrase to scramble (Enter to quit)? hairbrush
All words found in "hairbrush":
[bar, briar, brush, bus, hub, huh, hush, rub, shrub, sir, sub]

Max words to include (Enter for no max)? 2

[briar, hush]
[hush, briar]


Phrase to scramble (Enter to quit)? george bush
All words found in "george bush":
[bee, beg, bog, bogus, bough, brush, bug, bugs, bus, egg, ego,
erg, go, goes, gorge, gosh, grub, gush, he, her, here, hog,
hose, hub, hug, rub, she, shrub, shrug, sub, surge]

Max words to include (Enter for no max)? 0

[bee, go, shrug]
```

(continued on right)

```
[bee, shrug, go]
[bog, he, surge]
[bog, surge, he]
[bogus, erg, he]
[bogus, he, erg]
[bug, erg, hose]
[bug, goes, her]
[bug, her, goes]
[bug, hose, erg]
[bugs, ego, her]
[bugs, go, here]
[bugs, her, ego]
[bugs, here, go]
[bus, erg, go, he]
[bus, erg, he, go]
[bus, go, erg, he]
[bus, go, he, erg]
[bus, gorge, he]
[bus, he, erg, go]
[bus, he, go, erg]
[bus, he, gorge]
[egg, hose, rub]
[egg, rub, hose]
[ego, bugs, her]
[ego, grub, she]
[ego, her, bugs]
[ego, she, grub]
[erg, bogus, he]
[erg, bug, hose]
[erg, bus, go, he]
[erg, bus, he, go]
[erg, go, bus, he]
[erg, go, he, bus]
[erg, go, he, sub]
[erg, go, sub, he]
[erg, goes, hub]
[erg, he, bogus]
[erg, he, bus, go]
[erg, he, go, bus]
[erg, he, go, sub]
[erg, he, sub, go]
[erg, hose, bug]
[erg, hub, goes]

. . . (full output is on web site)
```

Implementation Details:

Your `Anagrams` class must have the following public constructor and member functions:

`Anagrams(vector<string>& dictionary)`

In this constructor you should initialize a new anagram solver over the given dictionary of words. You may assume that the words in the vector are in alphabetical order. Do not modify the vector passed to your constructor.

`vector<string> getWords(string& phrase)`

In this member function you should return a vector containing all words from the dictionary that can be made using some or all of the letters in the given phrase, in alphabetical order. For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and you are passed the phrase `"Barbara Bush"`, you should return a vector containing the elements `[abash, aura, bar, barb, brush, bus, hub, rub, shrub, sub]`.

`void print(string& phrase)`

In this member function you should use **recursive backtracking** to find and print all anagrams that can be formed using all of the letters of the given phrase, in the same order and format as in the example log on the previous page. For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and you are passed the phrase `"hairbrush"`, your member function 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]
```

An empty string should generate no output.

`void print(string& phrase, int max)`

In this member function you should use **recursive backtracking** to find and print all anagrams that can be formed using all of the letters of the given phrase and that include at most `max` words total, in the same order and format as in the example log on the previous page. For example, if your anagram solver is using the dictionary corresponding to `dict1.txt` and this member function is passed a phrase of `"hairbrush"` and a max of 2, your member function should produce the following output:

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

If `max` is 0, print all anagrams regardless of how many words they contain. For example, if using the same dictionary and passed a phrase of `"hairbrush"` and a max of 0, the output is the same as that shown earlier on this page with no max.

You should throw a string exception if the max is less than 0. An empty string should generate no output.

The provided `anagramMain` program calls your member functions in a 1-to-1 relationship, calling `getWords` every time before calling `print`. But you should not assume any particular order of calls by the client. **Your code should still work if the member functions are called in any order, any number of times.**

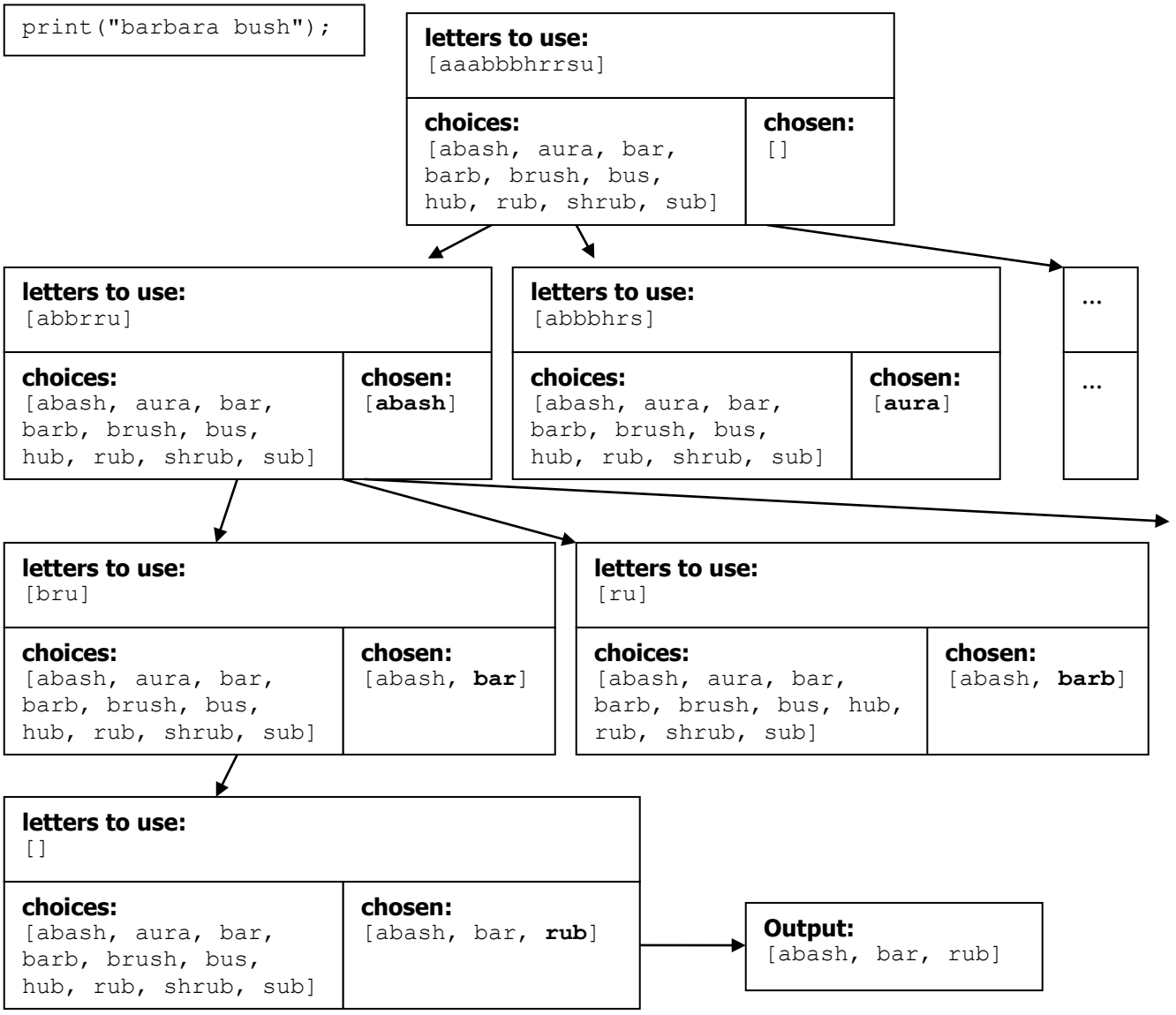
Recursive Algorithm:

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 must first find the collection of words contained in that phrase and consider only those words as "choices" in your decision tree. You should also implement the second `print` member function so that it backtracks immediately once it exceeds the max.

The following diagram shows a partial decision tree for generating anagrams of the phrase `"barbara bush"`. 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` Class:

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 the Maze example. 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. You must use your `LetterInventory` class from project 3 to help with this task.

Look back at the specification for project 3 if you have forgotten what the `LetterInventory` can do.

Development Strategy and Hints:

Your `print` member function must produce the anagrams in the same format as in the log. The easiest way to do this is to build up your answer in a `vector`.

The provided `anagramMain` program can read its input from different dictionary files. It is initially set to use 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`. You can find other larger dictionaries here:

- http://www.puzzlers.org/dokuwiki/doku.php?id=solving:wordlists:start

One difficult part of this program is the second version of `print` that limits 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.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing recursive backtracking to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary or repeat cases already handled.

As always, redundancy is another major grading focus; some member functions are similar in behavior or based off of each other's behavior. **You should avoid repeated logic as much as possible**. Your class may have other member functions besides those specified, but any other member functions you add should be `private`.

You should follow good general style guidelines such as: making member variables `private` and avoiding unnecessary member variables; appropriately using control structures like loops and `if`/`else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

In this project, we will be checking your code very carefully for good use of `const`. **For full credit, make every function and reference parameter `const` that can be.**

Comment your code descriptively in your own words at the top of your class, each member function, and on complex sections of your code. Comments should explain each member function's behavior, parameters, return, pre/post-conditions, and exceptions.