Spring 2018 CS 32

Programming Assignment 4
Anagrams

Time due: 11:00 PM Thursday, June 7

An *anagram* of a collection of letters is a word that is a rearrangement of all the letters in that collection. For example, anagrams of "idte" are "diet", "edit", and "tide". An anagram of "importunate" is "permutation". An anagram of "excitation" is "intoxicate".

Here is the interface for a class that stores words and lets you find anagrams:

```
class Dictionary
{
  public:
    Dictionary();
    ~Dictionary();
    void insert(std::string word);
    void lookup(std::string letters, void callback(std::string)) const;
};
```

As you would expect, the constructor creates an empty dictionary, and the `insert` function adds a word (stripped of any non-letters) to the dictionary. The `lookup` function takes a string and a *callback* function, and for every word in the dictionary that is an anagram of the letters in the string, it calls the callback function with that word as an argument. (The callback function must take one string as an argument, and returns void.) For example,

```
void printWord(string w)
{
    cout << w << endl;
}

vector<string> v;  // global variable

void appendWord(string w)
{
    v.push_back(w);
}

int main()
{
    Dictionary d;
    d.insert("cat");
    d.insert("dog");
    d.lookup("tca", printWord);  // writes  cat
    assert(v.empty());
    d.lookup("tca", appendWord);
    assert(v.size() == 1  &&  v[0] == "cat");
}
```

We have written a correct but horridly inefficient Dictionary implementation. Your assignment is to write a more efficient correct implementation. If you wish, you can do this by starting with our implementation in `Dictionary.cpp` (the only source file you will modify and turn in), and changing the data structures and algorithms that implement the class and its member functions. You are free to add, change, or even delete classes or functions in `Dictionary.cpp` if you want to.

Correctness will count for 40% of your score, although if you turn in a correct implementation that is no faster than the horridly inefficient one we gave you, you will get zero correctness points (since you could just as well have turned in that same horridly inefficient program).

Of course, we have to give you some assumptions about the way clients will use Dictionary so that you know what needs to be faster. The client may call `insert` tens of thousands of times. The collection of letters for which we want to find all the anagrams is typically about four to ten letters, but may well be more than that. Your program should be able to process tens of thousands of requests for anagrams quickly.

Performance will count for 50% of your score (and your report for the remaining 10%). To earn any performance points, your program must be correct. (Who cares how fast a program is if it produces incorrect results?) This is a critical requirement — you *must* be certain your program produces the correct results and does not do anything undefined. Given that you are starting from a correct program, this should be easier than if you had to start from scratch. The faster your program performs on the tests we make, the better your performance score.

When we do the performance tests on your program, we will create a dictionary, insert a bunch of words, start the clock, do a lot of lookups, stop the clock, and destroy the dictionary. This means that your insertions don't have to be fast (within reason) as they build the data structures required for efficient lookups.

When you use Visual C++ or Xcode, the default build configuration is the Debug configuration, one in which the compiler inserts extra code to check for a variety of runtime problems. This is nice when you're not yet sure your program is correct, but these extra checks take time, and in the case of checks involving STL containers, potentially a lot of time. The g32 command is similar in this regard.

When you are sure your program is correct, and you're ready to test its speed, you'll want to build an optimized version of your program (which can run an order of magnitude faster than the Debug version). See the last problem of Homework 4 for instructions on how to build in Release mode for Xcode and Visual C++; for g++, on `cs32.seas.ucla.edu` say

```
g32fast -o tester Dictionary.cpp interactivetester.cpp
./tester
```

To give you a taste of how fast the program can be, we did this project and produced this Windows program, this Mac program, and this Linux program. Put the executable file in the same directory as the `words.txt` file we supply in `anagrams.zip`, and try the program on `words.txt` with an collection of letters like "intoxicate" or "Veronica Snot".

When you believe your implementation is correct, and you're ready to test its correctness and speed, transfer it to `cs32.seas.ucla.edu`. On that server, run this command:

```
curl -s -L http://cs.ucla.edu/classes/spring18/cs32/Utilities/p4tester | bash
```

It will deposit several files in the current directory, then build and run correctness tests and a performance test. If your implementation does not pass the correctness tests, it will **not**earn **any** performance points. Our solution passes the basic and thorough correctness tests and runs in less than 8 milliseconds on`cs32.seas.ucla.edu`. While the exact performance scoring has not yet been determined, we expect something approximately like the following: times below 20 ms would earn the full 50 performance points; below 50 ms, 45 points; below 100 ms, 40 points; below 200 ms, 35 points; below 400 ms, 30 points; below 8000 ms, 25 points. For implementations that take longer than about 8 seconds, we will have to run a limited performance test, for which it's hard to say in advance what the scoring will be.

Here are some requirements your program must meet:

- You must not change `Dictionary.h` in any way. (In fact, you will not turn that file in; when we test your program, we'll use ours.) You can change `Dictionary.cpp`however you want, subject to this spec. (Notice that by giving Dictionary just one private data member, a pointer to a DictionaryImpl object (which you can define however you want in `Dictionary.cpp`), we have given you free rein for the implementation while still preventing you from changing the interface in any way. This is an example of what is known as the pimpl idiom (from "**p**ointer-to-**impl**ementation").

- You may design interesting data structures of your own, but the only containers from the C++ standard library you may use are `vector`, `list`, `stack`, and `queue` (and `string`). If you want anything fancier, implement it yourself. (It'll be a good exercise for you.) If you decide to use a hash table, it must not have more than 50000 buckets. Although we're limiting your use of *containers* from the library, you are free to use library *algorithms* (e.g., `sort`).

- The `Dictionary::lookup` function must call the provided callback function once for every word that is an anagram of the given collection of letters. If more than one word is an anagram of that collection, the order in which those words are passed to the successive calls to the callback function is up to you. For example,

```
Dictionary dict;
dict.insert("cat");
dict.insert("act");
dict.lookup("cat", f);  // f is the name of some function
```

  will call either `f("cat")` first and `f("act")`second, or `f("act")` first and `f("cat")`second, your choice.

- Duplicate words are allowed in the input, and each instance of a duplicate counts as an anagram. For example,

```
Dictionary dict;
dict.insert("cow");
dict.insert("cow");
dict.lookup("woc", f);  // f is the name of some function
```

  will call `f("cow")` twice, since "cow" appears twice in the dictionary. This requirement makes things easier for you, since there's nothing special for you to check.

- During execution, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

Turn it in

By Wednesday, June 6, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing two files:

- `Dictionary.cpp`. You will not turn in `Dictionary.h` or a main routine, since we'll supply our own.
- `report.docx`, `report.doc`, or `report.txt`, a report containing
  - a description of your algorithms and data structures (good diagrams may help reduce the amount you have to write), and why you made the choices you did. You can assume we know all the data structures and algorithms discussed in class and their names.
  - pseudocode for non-trivial algorithms.
  - a note about any known bugs, serious inefficiencies, or notable problems you had.

Your report will count for 10% of your score. We'll be reading it with this in mind: If you were standing around a whiteboard talking about your design and gave this description, would a CS grad student know enough about what you had in mind to be able to code it up without asking you any further questions?