CS 225

lab_dict Devious Dictionaries: Cleverness Through Association

Due: Sunday, October 25 at 11:59 PM
Doxygen for lab_dict

Assignment Description

This lab is all about making you think with dictionaries. Dictionaries (aka *maps* or *associative arrays*) are an abstract data type which stores pairs of data and can support operations like `insert`, `find`, and `remove`. We generally call these *(key, value)* pairs. The idea is to associate a *key* with a certain *value*. Hence, dictionaries are useful for when you want to quickly “lookup” a value associated with a certain key. The name *dictionary* comes from the more familiar physical dictionary, where you lookup the definitions of words. If we were to model a physical dictionary with an ADT dictionary the keys would be the words, and the values would be their definitions.

Since dictionaries are abstract in nature, they have no prescription for what underlying structure should be used to implement them. Common choices are tree-based structures (e.g. balanced binary search trees, BTrees, &c.) or some sort of hash table. Different underlying implementations have different performance considerations, and which you use in the real world will depend on things like desired performance and what is readily available.

For this lab, we’re not going to concern ourselves with the underlying implementation of dictionaries (we’ll do that in later labs). Instead, we’ll be using dictionary types that have already been defined for us. In C++ there are two “built-in” dictionary types: `std::map` and `std::unordered_map`. Both exist in the wonderful land of the Standard Template Library (STL). The STL is a subset of the C++ Standard Library (the library a compiler must implement to be standard-compliant) which provides data containers. As the name implies, these containers are templated and thus can contain any type (which meets the restriction of the container). You’ve probably experienced at least one of the STL types so far: `std::vector`, which is pretty much just a super fancy array. In general, the STL types are very convenient as they provide a lot of functionality (e.g. sorting, automatic resizing) that you don’t have to bother implementing yourself.

Dictionaries are amongst some of the most important abstract data types that are used in the real world. Therefore, this lab is perhaps one of the most useful labs from a real world perspective. Mastering the concepts that you see in this lab will be a tremendous help for most technical interviews and programming problems in the future.

C++11 Goodness

`auto` — easy types

Most of the code we present in this class uses old (C++03) style, but in this lab we encourage you to use some C++11 features to make your life easier.

The first helpful C++11 tidbit we'll talk about is `auto`. STL functions are really big on iterators and templates. Unfortunately, these types can get quite verbose. For example, if I have an `std::unordered_map<int, string>` (which maps an integer to a string value) called `m` and I want to get the value associated with the key `5` (if it exists), I have to do something like this:

```C++
int key = 5;
unordered_map<int, string> m;
unordered_map<int, string>::iterator lookup;


lookup = m.find(key); /* unordered_map::find returns an iterator t
if (lookup != m.end()) { /* If the key isn't in the map, m.end() i
    cout
        << "key: "
        << lookup->first  /* the 'first' item in the pair is the k
        << "  value: "
        << lookup->second /* the 'second' item in the pair is the
        << endl;
} else {
    cout << "key " << key << " not found" << endl;
}
```

Check out the type of `lookup`: `unordered_map<int, string>::iterator`. Yikes. The trouble with the STL is that every time we want an iterator we have to specify the full type of the container and then add `::iterator`. This can get pretty annoying. Thankfully, in C++11, there's `auto`. Instead of the above, I could do this:

```C++
int key = 5;
unordered_map<int, string> m;


auto lookup = m.find(key); /* unordered_map::find returns an itera
if (lookup != m.end()) { /* If the key isn't in the map, m.end() i
    cout
        << "key: "
        << lookup->first  /* the 'first' item in the pair is the k
        << "  value: "
        << lookup->second /* the 'second' item in the pair is the
        << endl;
} else {
    cout << "key " << key << " not found" << endl;
}
```

Instead of specifying the type of `lookup`, we now specify it as `auto`. `auto` replaces the type information and does something called “type inference”. What this means is that the compiler will look at the type of the expression on the right hand side of the assignment and “fill in” the type information of the variable for you. Cool stuff, right? This means that for things like `find` we don’t have to type out the nasty iterator type information. Now, don’t go crazy with `auto`. A good rule of thumb is to only use it when it makes your code more readable. Since things like STL objects are pretty common it’s often nice to use `auto` when dealing with functions that return iterators. For this lab I would type out the full type information the first few times (just so that you remember what the actual type is) and then after that use `auto`. Something that can be nice to do when using `auto` is to indicate (partially) what the type is in the name. Something like `lookup_itr` to indicate that this variable is really an iterator.

`map::operator[]` / `unordered_map::operator[]` — easy element access

A convenient way to access elements in a map is with `operator[]` (just like array subscripts). However, you have to be careful. If `key` doesn’t exist in `m`, `m[key]` will create a default value for the key, insert it into `m` (and then return a reference to it). **Because of this, you can’t use `[]` on a `const` `map`.**

Range-based loops — easy iteration

The next super cool thing in C++11 which is useful in this lab is the range-based `for` loop, also called for-each loops. As it turns out, a lot of programs involve iterating over collections of data (e.g. an array) and doing something with each of those values. The duty of a programming language is to make programming easier for us humans, so there’s usually some sort of construct in the language to accomplish this. In C++03 the main construct was `for` loops with iterators. So if I had a `map<int, string> m;` and I wanted to iterate over every `(key, value)` pair I would have to do something like this:

C++

```
map<int, string> m;
for (map<int, string>::iterator it = m.begin(); it != m.end(); it+
    cout << it->first << ", " << it->second << endl;
}
```

C++11 gives us a better way, taking advantage of what’s called a “for each loop” or “range-based for loop”:

C++

```
map<int, string> m;
for (std::pair<const int, string> & key_val : m) {
    cout << key_val.first << ", " << key_val.second << endl;
}
```

The way you should read that `for` loop is: “for each `key_val` in `m` do …“. What’s happening is that the `for` loop will update the variable `key_val` in each iteration of the loop to be the “next” element. Since we are iterating over a `map`, the elements are `(key, value)` pairs.

Notice that the type of key_val isn't an iterator: it's the "dereferenced" value of an iterator, in this case a reference to a std::pair<const int, string>. This value is a reference for two reasons: (1) so we don't make useless copies, and (2) so we can modify the values in the map.

When we combine the ranged-for loop with auto, we get nice, readable code:

C++

```
map<int, string> m;
for (auto & key_val : m) {
    cout << key_val.first << ", " << key_val.second << endl;
}
```

These kinds of for loops are simple, convenient, and intuitive to use. They have their limits, but for many iterating applications in C++ they're probably the best choice.

Now that we've talked about what dictionaries and underlined their importance, let's move on the actual contents of this lab. This lab is composed of a bunch of different components. We suggest you do them in the order that we describe them over here.

Checking Out The Code

You can get the code in the usual way, running svn up from your cs225 directory.

Like last week's lab, there's a data directory that you need to download separately. You can get it by running:

TERMINAL

```
make data
```

Memoization

Memoization is an aspect of Dynamic Programming, which you'll learn much more about when you take CS 374. However, it's a fairly simple idea, and a common implementation uses dictionaries at its core.

Factorial

Our first introduction to memoization will be through the use of factorials.

The factorial is a deterministic function: for a given input n, the value $(n!)$ is always the same. The formal (recursive) definition is:

$$0! = 1$$

$$n! = n \times (n - 1)!$$

For example, say we already know the value of $5!$ and we want to compute $6!$. The naïve way computing of $6!$ would be to calculate $6! = 6 \times (5 \times (4 \times (3 \times (2 \times (1)))))$. Since we already know the value of $5!$, however, we can save quite a bit of computation (and time) by directly computing $6! = 6 \times 5!$, using the known value of $5!$.

This idea is essentially what memoization is. According to Wikipedia:

> [M]emoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again

Now let's look at how we do this in code by looking at the `memoized_fac` function in `fac.cpp`. On line 97, we find:

```C++
static map<unsigned long, unsigned long> memo = { {0, 1} };
```

The left hand side is creating a variable named `memo` which is a `std::map` that associates keys of type `unsigned long` to values of type `unsigned long`. In our case, we're going to map a key n to the value $n!$.

The `static` keyword is a neat little thing that basically makes this local variable remain initialized for the lifetime of the program so that we can reuse it across multiple calls to the function. It's like a global variable, but it's *only* accessible by that function.

The right hand side initializes the map to contain a key-value pair where the key is `0` and the value is `1`. Notice that this key-value pair corresponds to the base case of our mathematical definition of $n!$ given above.

Alternatively, we could have used

```C++
static map<unsigned long, unsigned long> memo;
memo[0] = 1;
```

This disadvantage of this method is that now the key `0`'s value will be changed to the value `1` during every function call, as opposed to just at initialization as in the first example.

Line 102 is

```C++
auto lookup = memo.find(n);
```

`memo.find(n)` uses the `std::map::find` function. As mentioned above, this method returns an iterator; the type of `lookup` is `std::map<unsigned long, unsigned long>::iterator`.

Lines 103–110 are:

```C++
if (lookup != memo.end()) {
    return lookup->second;
} else {
    unsigned long result = n * memoized_fac(n - 1);
    memo[n] = result;
    return result;
}
```

Going back to what we already know about iterators and `std::map::find`, a return value of `memo.end()` basically implies that the key `n` does not exist in the map. In this case, we need to compute `n!` and update our map to store that value, which is exactly what we do in the `else` branch. A `std::map::iterator` is basically a pointer to a `std::pair<const key_type, mapped_type>` (`std::pair`). In the `if` branch, `lookup->second` returns the value stored in the map for key `lookup->first`, which is going to be `n` in our case. If we wanted to be clever, we could rewrite this code as:

```C++
if (lookup == memo.end()) {
    memo[n] = n * memoized_fac(n - 1);
}
return memo[n];
```

Memoizing the factorial function doesn't actually do much for us in terms of computation time. To see this for yourself you can run:

```TERMINAL
make fac
time ./fac 20
```

and compare it to the time it takes with:

```TERMINAL
time ./fac 20 -m
```

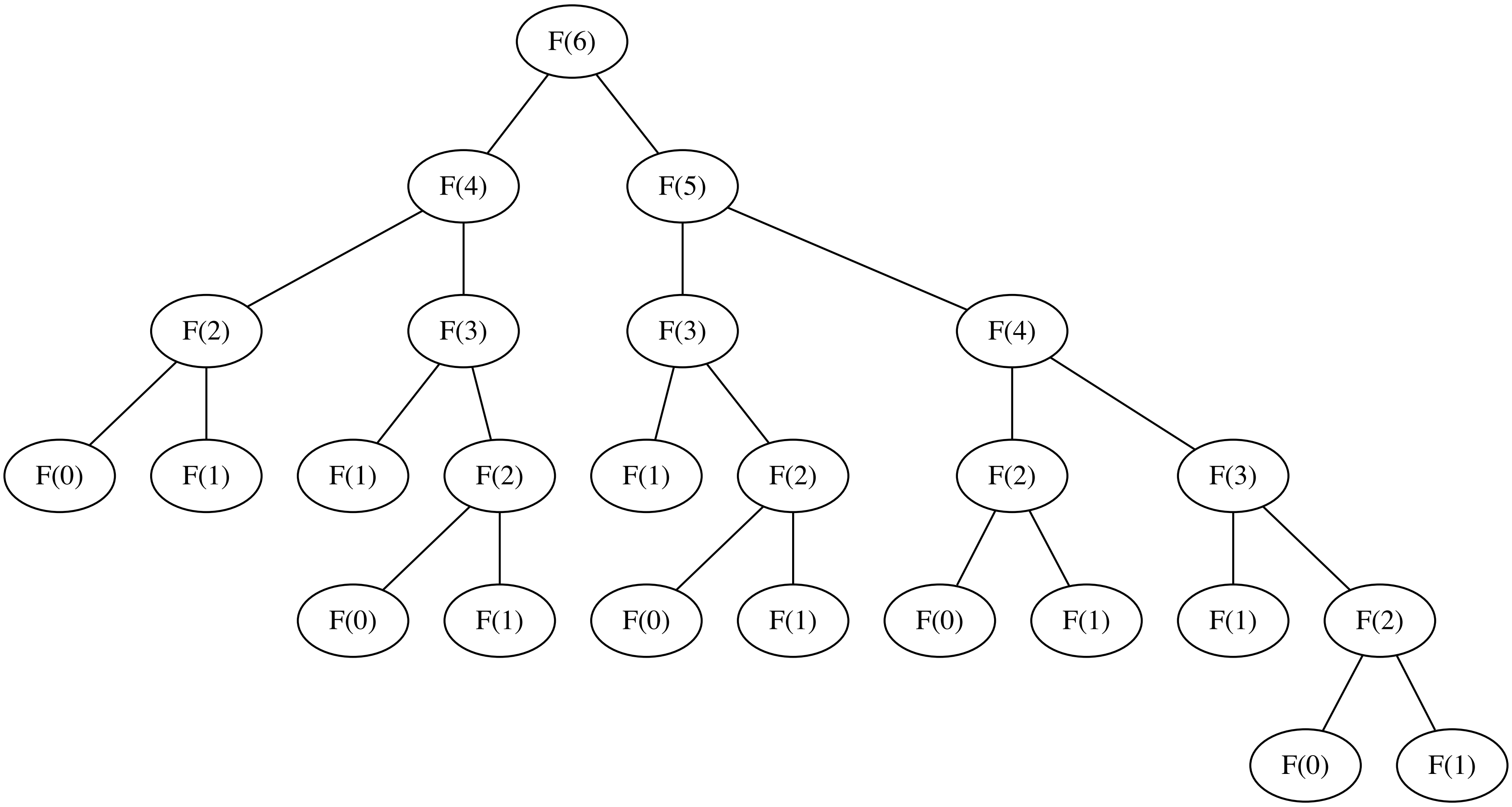
Fibonacci

Now that you're familiar with how to memoize the factorial function, you'll apply memoization to the Fibbonacci function. As we'll soon find out, memoizing this function has real speed benefits.

The Fibonacci numbers are recursively defined as:

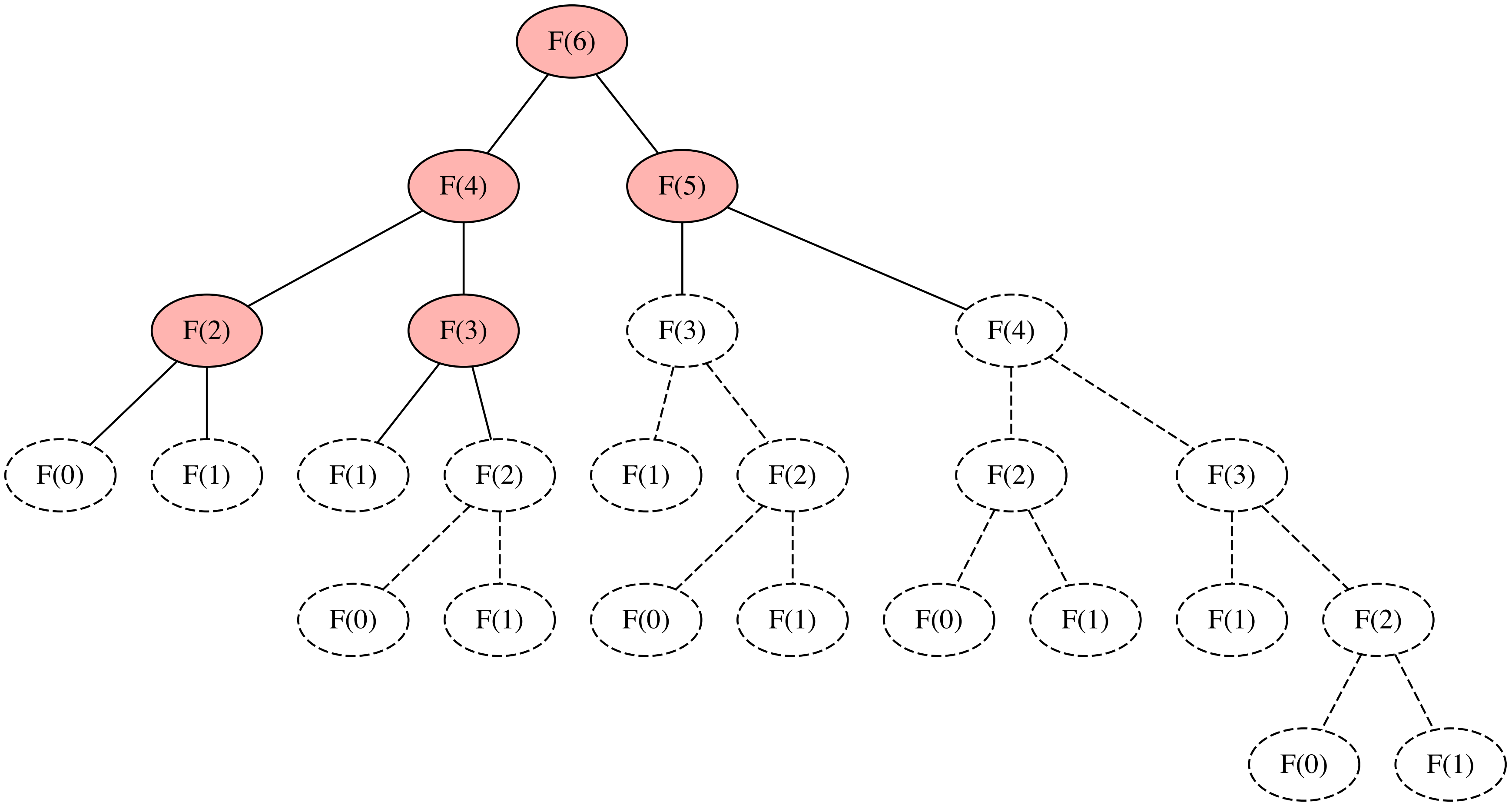
$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

If you draw out some trees for calculating these numbers (which I highly recommend) you'll notice something interesting. For example here is such a tree for $F(6)$; each node represents a function call:



There's quite a bit of repetition in this tree. Specifically the entire $F(4)$ subtree appears twice in the tree, as does $F(3)$. The values in those trees get calculated twice. For larger values of n, the tree naturally gets larger and so too do the repeated subtrees. Remember that the nodes in this tree represent function calls, so each one takes some time. Suppose the $F(4)$ call on the right happens before the $F(4)$ call on the left. By the time we get to the $F(4)$ call on the left, we already knew the value of $F(4)$. What if we could store the result of $F(4)$? If we stored it, then when we call $F(4)$ on the left we don't need to bother with going through that entire subtree.

As we found out with factorial, we can use a dictionary for exactly this purpose. Imagine that inside our F function that we associate an input number n with a Fibonacci number. Every time we successfully calculate a value $F(n)$ we store the pair $(n, F(n))$ in our dictionary. So now in our function we first check to see if n already exists in the dictionary (i.e. we've already computed it). If it is, we return that immediately. Otherwise, we make the normal recursive call, store the result in the dictionary, and then return the result. Using memoization the tree becomes:



The dashed nodes are ones which didn't need to be calculated; one of their parent's values was taken from the dictionary rather than being recalculated from scratch The red nodes are the only calls where it's calculated. As you'll see in a while, the difference memoization makes is rather dramatic.

Your task is to implement both the normal and memoized version of the `fib` function in `fib.cpp`. After you do this you can race them with the `fib_generator` executable:

TERMINAL

```
make fib_generator
time ./fib_generator 45
```

To use the memoized version, pass the `-m` flag:

TERMINAL

```
time ./fib_generator 45 -m
```

Common Words

The next part of this assignment involves making an object that can find common words in different files. The end goal is: given a list of files and a number n find all words that appear in every file **at least** n times. This is a sort of conditional intersection of the words in the files. As you might expect, you'll be using a dictionary to solve this problem, with the words as the keys, and their frequencies as the values. (This is a common application of associative structures). You will accomplish this by finishing the `CommonWords` class.

You can find the Doxygen for the `CommonWords` class here. You are responsible for writing the `init_file_word_maps`, `init_common`, and `get_common_words` functions. The former two will help you write the latter function. The header file (`common_words.h`) has descriptions of the variables you are initializing.

> **Alternate Implementations**
>
> This is just one way to solve the problem. If a different way of using `maps` (and other structures) seems more intuitive to you, feel free to do that instead. For instance, you may want to use a map from a word to a vector of integers, where each integer corresponds to the number of times that word appears in the corresponding file.

Once you've written your class, you can compile it with `make find_common_words`. You can then test it on some small text files or novels:

```
./find_common_words data/small1.txt data/small2.txt -n 3
dog
pig
```

```
./find_common_words data/PrideAndPrejudice.txt data/Beowulf.txt data/S
and
in
of
the
to
```

The executable command below finds all the words that appear greater than or equal to n times in ALL the parameter text files. Here is the usage:

```
./find_common_words [TEXT FILES] -n [NUM] [-o FILE]
```

Pronunciation Puzzler

For this part of the lab you'll complete an object which will help solve the following puzzle (taken from CarTalk):

> This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you

> remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?
>
> Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.' As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.
>
> But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?

Basically, we're looking to find words such that

- the word itself,
- the word with its first character removed, and
- the word with its second character removed

are all homophones (i.e., have the same pronounciation) of each other.

To accomplish this we are going to employ the help of CMU's Pronouncing Dictionary. The data associated with that dictionary is stored in a file (`data/cmudict.0.7a`) and read on construction of a `PronounceDict` object (see `pronounce_dict.h`). You don't have to worry about the constructors this time since they're not very interesting. Instead, you're responsible for writing the `homophones` function which determines whether two words are homophones are not. Check out `pronounce_dict.h` to see what kind of dictionary structure you're working with, and remember that it is mapping a word to its pronunciation.

Here is the Doxygen for the `PronounceDict` class.

`PronounceDict` expects the words to be uppercase. Here's how you can transform a `std::string str` to uppercase in C++:

```C++
std::transform(str.begin(), str.end(), str.begin(), ::toupper);
```

It uses `std::transform` (you know how to look this function up, right?) We'll leave it up to you to try and figure out exactly how it works. If you need help, post on Piazza!

Next you will have to write the code which actually solves the puzzle. This function resides in `cartalk_puzzle.cpp`. It takes a word list file name (similar to before) and a `PronounceDict`.

You need to be able to read in a list of words form the word list. To do this, you'll need to know some basic C++ file I/O. This following snippet will print out every word in a word list file:

```C++
ifstream wordsFile(filename);
string word;
```

```
if (wordsFile.is_open()) {
    /* Reads a line from `wordsFile` into `word` until the file ends.
    while (getline(wordsFile, word)) {
        cout << word << endl;
    }
}
```

To solve the actual puzzle, methods like the `std::string`'s `substr` function will probably prove useful. If all goes well you should be able to make it and run the executable with:

TERMINAL

```
make homophone_puzzle
./homophone_puzzle
```

If you want to debug with `cout`, I would recommend typing `using std::cout;` and `using std::endl;` instead of a blanket `using namespace std;` or else you may get some nasty errors.

There should be 5 (only 4 if you include the length requirement) resulting triples of words, but only one which matters. I believe the answer CarTalk was looking for is the only one in the list where the two homophones are distinct (it doesn't matter for us if you don't make sure the homophones are distinct).

Anagrams

The final part of this assignment involves making a dictionary for looking up anagrams of a given word. Two words are anagrams of one another if we can rearrange the letters of one to form the other. For example, the letters in the word "dog" can be rearranged to form the word "god". We want an object (built from a word list) whom I can feed the string `"dog"` and it will spit out both `"dog"` and `"god"`.

You'll find the definition of the `AnagramDict` class in `anagram_dict.h`. You are responsible for implementing both constructors, as well as the functions `get_anagrams` and `get_all_anagrams`. `AnagramDict` has an instance of map named `dict` which maps a `string` to a `vector` of `strings`.

> You **MUST** implement both constructors. The `./anagram_finder` executable uses the constructor that takes a filename; monad uses the constructor that takes a `vector`.
>
> Make sure you check your code with monad!

Remember that when using dictionaries we need to figure out a way to uniquely identify each of our values. In this case we want to uniquely identify a group of words which are anagrams of one another: we need some sort of function `f(key)` that we can apply to a key such that `f("dog") == f("god")` and `f("silent") == f("listen") == f("tensil")`, but `f("dog") != f("silent")` Phrased differently, you'll need to come up with some sort of a function, such that it always returns the same value for all words that are anagrams of one another.

Once you figure out what a function `f` must do satisfy that constraint (and there are many possible answers), you'll find it useful to Google how to actually implement that function if it seems non-trivial to you. (Actually, you'll find it useful to Google for pretty much anything, including what should `f` do.)

Here is the Doxygen for the `AnagramDict` class.

One of the `AnagramDict` constructors takes a word list file name, which is expected to be a newline-separated list of words (see `words.txt`). Read this file the in same way as you did for the pronounciation puzzle.

To test your dictionary you can use the `anagram_finder` executable, which you can compile by running `make anagram_finder`. Example usage and output:

TERMINAL

```
./anagram_finder dog
Anagrams for dog:
dog
god
```

You can run it with the flag `-a` to print out a list of all the known anagrams for this word list. `-o` will redirect the output to a file, e.g.

TERMINAL

```
./anagram_finder -a -o anagrams.txt
```

You can `diff` your output of the above command with `data/all_anagrams_soln.txt`. You may also specify your own word list file with `-w` if you're feeling adventurous (it defaults to the `words.txt` in the `data` directory).

Testing Your Program

As usual, there are Monad tests for this lab, which can be run with:

TERMINAL

```
./monad lab_dict
```

or (to have it use your code on SVN):

TERMINAL

```
./monad NETID/lab_dict
```

Also, fun fact, you can run monad with `--parallel` and it will compile your code with four threads instead of one. This will make a nice speed increase on machines with more than one core.

Committing Your Code

Commit in the usual way.

Grading Information

Only the following files are used for grading:

- `anagram_dict.cpp`
- `anagram_dict.h`
- `fib.cpp`
- `pronounce_dict.cpp`
- `pronounce_dict.h`
- `cartalk_puzzle.cpp`
- `common_words.cpp`
- `common_words.h`

All other files including any testing files you have added will not be used for grading.

Good Luck!