Evil Hangman

In this lab we're going to take Hangman to the next level. We'll modify the game slightly to make it look more like regular hangman, but we'll also modify what is going underneath a lot so the hapless human player will have little or no chance to win.

This assignment is based on the Evil Hangman assignment first developed by Keith Schwarz at Stanford (http://www.keithschwarz.com/cs106l/fall2010/handouts/020_Assignment_1_Evil_Hangman.pdf). It is also worth noting that it contains some significant differences.

Goals

This will be your introduction to building your own data structures. In fact, the data structures you build for this lab will be as sophisticated as any you build all term (largely because we'll take advantage of built-in data structures in future labs). Not only will we use **Lists**, but we'll have **Lists of Lists** and **Arrays of Lists**.

Evil Hangman

The premise of Evil Hangman is that the computer is a cheater (you've probably always suspected that about computer opponents when you've played video games anyway). Imagine that you are the person guessing the word and the game is in the following state:

D O - B L E

There are two english words that match this pattern - "double" and "doable." When played fairly you have a fifty-fifty chance of winning in one guess. If your opponent hasn't committed to either word though then you have no chance if you are down to one guess. If you guess "u" the computer will simply tell you that the word was actually "DOABLE" if you guess "a" the computer would say that it was "DOUBLE."

Here's an example. Suppose you are playing and it is your turn to choose a word and that the word will be four letters long. And because you play a lot of Scrabble you know every four-letter word in English (for simplicity let's assume there are only nine). Here is our theoretical list of all four-letter words

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

Now suppose your opponent guesses the letter 'E.' You now need to tell them which letters in the word you've "picked" are E's. Since you haven't actually picked a word you have options. Here's the list agin with the E's highlighted:

ALLY B**E**TA COOL D**E**AL **E**LS**E** FL**E**W GOOD HOP**E** IB**E**X

Every word in the list falls into one of five "word families."

_ _ _, which contains the words ALLY, COOL, and GOOD.
_ E _ _, which contains BETA and DEAL
_ _ E _, which contains FLEW and IBEX
E _ _ E, which contains ELSE
_ _ _ E, which contains HOPE.

The rules of the game dictate that you have to reveal something so you have to pick one of these families. There are lots of ways to do this (it is actually a fascinating problem on its own), but we'll stick to a simple one - simply choose the family with the largest number of words remaining. In this case it means that you should choose _ _ _ _ since it contains the most words (three). That reduces the list down to:

ALLY COOL GOOD

Since you didn't reveal any letters you would tell your opponent that their guess was wrong.

Now imagine that your opponent guesses O. That breaks your new list down to two families:
_ O O _, which contains COOL and GOOD
_ _ _ _, which contains ALLY

The first is larger than the second so you choose it. Thus you reveal the two O's in the word and reduce your list down to
COOL GOOD

You simply continue this process until your opponent either runs out of guesses (the usual result) or they actually manage to narrow your list down to one word and guess it.

The Assignment

You need to modify your Hangman program (or start over if you'd like) in the following ways:

1. Your program should now play a single word version
2. Your program should read the file **dictionary.txt**, which contains the full contents of the *Official Scrabble Player's Dictionary, Second Edition*. The word list has over 120,000 words.
3. Your program should prompt the user for a word length, reprompting as necessary until they enter a number such that there is at least one word that long (the longest in the dictionary is 29 letters if I'm not mistaken).
4. Prompt the user for the number of guesses (must be an integer greater than 0).
5. Have a debugging option that can be turned on through a #DEFINE that prints a running total of the number of words left.

Here's how your program should play:

1. Construct a list of all words in the English language whose length matches the input length.
2. Print out the state of the game as in the last assignment.
3. Ask the user for guesses as in the last assignment.
4. Based on the guess, partition the words in the dictionary into groups by family.
5. Find the most common "family" in the remaining words. That family becomes your new word list.
6. Based on that list, you have a new game state, so proceed as normal.

Plan of Attack

Some of the program can come directly from your last assignment, or from small modifications of your last assignment. Do use it. Before we get into your code we should go over the data structures.

Data Structure - Array of Lists

The first major data structure you'll encounter in this program will come when you read in the dictionary. The goal of this data structure is to organize the words in the dictionary such that you can quickly access all of the words of a given length to make your initial word list. The simplest way to do this is to make an array of lists. Each element of the array is essentially the head of a list. Then when you read in a new word, for example a five-letter word, you can add the word to the appropriate list. After you have read in the dictionary you'll have somewhere in the neighborhood of 25 lists (there are no 0 or 1 letter words in the Scrabble dictionary obviously, and a few other lengths are empty). Then when the user selects a word size making the word list can be done in a simple assignment statement by grabbing the head of the right list.

So what would an array of lists look like? Here's how I defined one in my program:

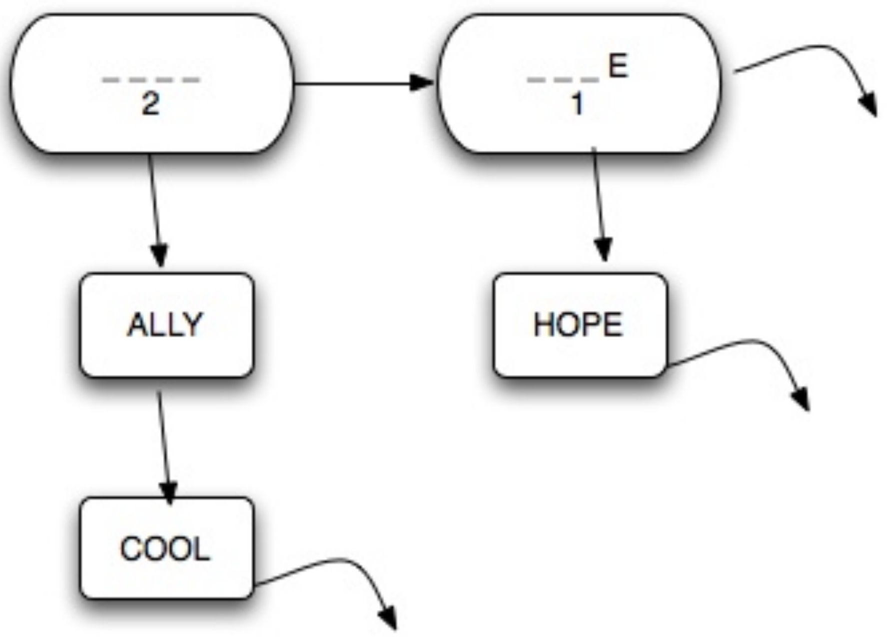
```
struct node* lists[MAX];
```

So it is an array of size MAX (29 or 30 depending on how you handle word lengths) that contains pointers to nodes. Adding a word to any of these slots is the same as adding a word to a normal list.

It may be worth making an auxiliary data structure to go along with this - a second array that just holds the number of words in each list of the primary array. You probably don't have to have it, but it can be a nice add-on. I used one, for example, and used it to print out all of the frequencies after I read the dictionary.

Data Structure -List of Lists

The second major data structure will contain the “word families” and their associated lists. Where the first structure was an array of lists, this one will be a list of lists. When the user makes their guess you will need to generate all of the word families on the fly based on the words left in your main list. Essentially you will loop through the main list and create the word family for each word and check if it has already been found for another word. If it had been found you would add that word to the list, and if it hasn’t been found you would create a new list.



The figure shows our data structure after three four letter words have been processed with a user guess of “E”. The first element of the list is the word family “_ _ _ _” and it has two elements. The second element is the family “_ _ _ E” with one word.

Here’s how I defined the struct that serves as the basis for this data structure.

```
struct family {
    struct node* members;
    int number;
    char *description;
    struct family *next;
};
```

Here *members* is our list of words, *number* is the number of members, *description* is the word family description and *next* is the next list in our list.

How to Proceed

Reading the dictionary

Your first goal should be to read in the dictionary and set up the first data structure. This is a bit tricky. I suggest you do something called "buffered" reading. The idea is that you have a buffer (just some memory) where you do your reading. Once you've read a word into that buffer then you can go on and process it.

So the buffer might be defined like so:

```
    char buff[50];
```

Then your read routine would loop something like this (assuming that you have set up file_ptr1 properly):

```
    while (fscanf(file_ptr1, "%s", buff) != EOF) {
           // string processing code

    }
```

This will read the words from the dictionary one at a time into buff. You'll want to examine *buff*, find its length, use that length to malloc some memory for a new string, copy the *buff* string into the new string, and then add that string to your array of lists data structure.

It should go without saying that you should get this running (and tested) before you move on to anything else.

A word on file paths. When we get to Objective-C we'll learn that it is pretty easy to get the file paths to things included in your XCode project. Unfortunately this does not seem to be the case for C (if anyone can find a better way please let me know!). Here's what I suggest: you can drag a copy of **dictionary.txt** directly into your XCode project. This will put it in the same directory as main.c (if you drag it into the same folder anyway). Once there you can run your program from the command line and it will find your dictionary. I used this:

```
char mode1[] = "r";
char file1[] = "dictionary.txt";
```

I am a big fan of starting small. In this case you might do well to create your own dictionary to use during debugging. Such a dictionary might only contain about 10 words. That will make it simpler to track what is going on when you have bugs. Also, we'll run into this again later this term when we use a similar dictionary in another application.

Once you have read the dictionary you can work on having the user select a word size and then using that size to create your initial list of words.

The User Loop

At this point you are ready for the main user loop. It will work a lot like the user loop from the last lab with the exception of the computer cheating element. When the guess is made you need to generate the word families, select the appropriate one and update the state of the game appropriately. Here is some pseudo code for this:

```
     reset the word family data structure
     loop through all of the remaining words {
          create the new word family for the current word
          does the family exist?
               Yes – insert the word into it
               No – create it and insert the word into it
     }
     choose the family that appears the most times
     that family becomes the new “answer” string to show user
     the list associated with that family becomes the new word
list
```

I suggest you do this using several functions (e.g. generateFamilies, insertToFamilies, . . .).

Extensions

Figure out a better way to decide on which family to choose. Make an argument as to why your algorithm is better. It is relatively easy to find cases where the “most words” left heuristic doesn’t work well (these generally involve situations where the “most words” heuristic gives the user a “correct” response).

Details

This will be a two-week assignment, but I will require you to hand in at least part of it by next Tuesday. By next Tuesday you should have the following running and tested:

1. Reading in the dictionary
2. Creating the array of lists
3. Initial user interactions asking for word length and number of guesses

Both next Tuesday and the following week you should make sure to hand in both electronic (through Blackboard) and hard copies. Your hard copies should have your signature on them with a statement that you have followed the course rules.

This assignment is hard. I’m giving you two weeks because you will need to work hard to get it done not because I’m generous.