Assignment 2

Due Feb 14 by 4pm **Points** 10

Assignment A2: Structs and Dynamic Memory

The goal of this assignment is to practice linked lists, strings, pointers, and dynamic memory allocation in C. You'll be implementing a word guessing game.

Wheel of Misfortune

Wheel of Misfortune is a two-player word game that goes like this. Player 1 picks a secret word, and tells player 2 the length. (For example, ----- would mean that the word selected by player 1 has five letters.) Player 2 then begins guessing letters. For each letter guessed by player 2 that appears in the word, player 1 reveals the positions of that letter. Otherwise, if the letter does not occur at all in the word, player 2 loses a guess. Player 2 wins the game if they are able to fully uncover the word before exhausting their allowed number of guesses. Player 1 wins if player 2 runs out of guesses.

But this isn't an ordinary opponent!

In this assignment, you will be implementing a Wheel of Misfortune game where the computer has the role of player 1, and a human has the role of player 2. However, rather than choosing a word at the beginning of the game like a good little computer is supposed to do, this nasty computer opponent is going to cheat and make it very difficult for the human to win. It will actively dodge human efforts to determine the word, as we'll explain now.

When the game starts, you will tell the computer the length of the word you want to guess. Let's say that you choose a word-length of four letters, so that you see ---- as the current representation of the word. Rather than playing fair and committing to one particular four-letter word from the outset, the computer instead compiles a list of every four-letter word in its dictionary. For simplicity, let's assume that it has exactly the following four-letter words: ally beta cool deal else flew good hope ibex. For your first guess, let's say you choose the letter e. The computer now must tell you whether there are any e's in the word and, if there is at least one e, it must also show you the word with each e written in its proper position.

But how can the computer do this, since it has **not** chosen a word yet? The trick is for the computer to divide all current words into **word families** based on where the e's lie in each word. Here is our vocabulary again, with the e's in bold: ally b**e**ta cool d**e**al **e**ls**e** fl**e**w good hop**e** ib**e**x. There are five word families:

- ----, which contains the words ally, cool, and good
- -e--, containing beta and deal
- --e-, containing flew and ibex
- e--e, containing else
- ---e, containing hope

What the computer will do is choose one of these word families, then uncover the e's that occur in each word of that chosen family. There are several means by which the computer could choose a family; in this

assignment, the computer will always choose the family with the most words. If there is a tie for the most words, the computer will chose any one of the families tied for the most words.

As evidenced in our running example, the largest word family is of size three (the family ----). The computer will choose this word family, reduce its candidate word list (now ally, cool, and good), and report to you that there are no e's in the word. Nasty!

It's not necessarily the case that the chosen word family includes no copies of your guess. (Remember, the computer is using the heuristic of always choosing the largest family. This is not always optimal.) To demonstrate this, let's say that the computer is working with ally, cool, and good, and your next guess is o. This time, there are two word families:

- -oo-, containing cool and good
- ----, containing ally

The computer would choose the first word family, reducing its word list to cool and good, and revealing two o's in the word.

Note that if you guess a letter that does not appear anywhere in the computer's current word list, or your letter appears in the same position of every single word in the current word list, the computer will "divide" the word list into a single family consisting of all words. This really isn't a special case at all if your code is sufficiently general (as it should be).

There are two ways that the game can end. The first is that you are lucky enough to pare the computer's word list down to a single word, in which case you win the game. The second (much more common :)) case is that you exhaust all of your guesses without guessing the computer's word. In this latter case, the computer can just pick any word at random from its current list, and report this word to you. All words in the current list are indistinguishable to you, since they are all in the same word family.

Starter Code

Your first step should be to log into **MarkUs** (https://markus.cdf.toronto.edu/csc209-2019-01/) and navigate to the **a2: Structs and Dynamic Memory** assignment, which triggers the starter code for this assignment to be committed to your repository. You will be working alone on this assignment, and the URL for your git repository can be found on the right hand side of the page.

Pull your git repository. There should be a new folder named a2. All of your code for this assignment should be located in this folder. Starter code, including a Makefile that will compile your code, has been placed in this folder.

Once you have pushed files to your repository, you can use MarkUs to verify that what you intended to submit was actually submitted. The Submissions tab for the assignment will show you what is in the repository, and will let you know if you named the files correctly.

Word Families

The concept of a word family is embodied in the Family type defined in `family.h`. Each family contains a signature giving the pattern of each of the words in the family, based on a single character guess. For example, the signature could be the string --e--, representing all words in the current (five-letter) word list whose third letter is an e. The `word_ptrs` member is what points to the actual words that belong to each family. `word_ptrs` is an array, where each element is a pointer to one of the words in the current word list. The

last element in `word_ptrs` should be a `NULL` pointer.

The `num_words` member keeps track of the number of words belonging to this family. It begins at 0 for each initialized family, and increases by one each time a new word is added. `max_words` gives the maximum number of words that this family can hold. When num_words reaches `max_words`, we have no space left in the `word_ptrs` array in which to place new pointers-to-words, so we must allocate more space to `word_ptrs`. In this way, we can begin by creating a family of modest size, and only if it becomes sufficiently large, increase its memory usage by fixed increments.

The next member of `Family` is a pointer to another `Family` structure. Through next pointers, we can create linked lists of families, allowing us to traverse a whole chain of families by beginning at the head of a family list.

A skeleton of the `Family` implementation is in `family.c`. You will write much of the code to complete this module.

Master words list

The starter code comes with a file named dictionary.txt. It is a huge English word list of over 120000 words. The file `reading.h` declares functions related to reading this dictionary into memory, and associated implementations are in `reading.c`. The first thing your program will do is call `read_words` to read the entire dictionary into memory. This words list remains in memory throughout the entire program run, and is never modified. Sub-dictionaries of specific lengths are used in each round, depending on the length of word that you choose. These sub-lists, like word lists associated with families, are not copies of the master words list's words, but are pointers to these words.

`reading.c` is a complete module for which you will not write any code. It will be helpful when working on other word-list-related tasks, however, to understand how this code works.

Game Code

Finally, the code that plays the actual game is in `wheel.c`. It uses functions defined in the helper modules `family.c` and `reading.c`.

The entry point (the `main` function) performs two initializations: it reads the dictionary words into memory, and initializes the family module. It then continues to call `play_round` until the user decides not to play anymore. Lastly, it deallocates the memory used for the dictionary, and exits.

Some of this module's code has been written, but other code is left for you to write, as explained below.

Your Tasks

Programming: `family.c`

Implement the following functions in `family.c`:

- `Family *new_family(char *str)`
- `void add_word_to_family(Family *fam, char *word)`
- `Family *find_family(Family *fam_list, char *sig)`
- `Family *find_biggest_family(Family *fam_list)`
- `void deallocate_families(Family *fam_list)`
- `Family *generate_families(char **word_list, char letter)`

- `char *get_family_signature(Family *fam)`
- `char **get_new_word_list(Family *fam)`
- `char *get_random_word_from_family(Family *fam)`

Details:

- Read the comments for each function carefully.
- Some function comments mention using realloc. Like `malloc`, `realloc` is used to allocate memory. Read the man page to find out more about realloc.
- Once the initial word list has been read from the dictionary file, you will not make any copies of the words. Instead you will make lists of pointers to these words.
- Be careful: `realloc` might not be able to extend `word_ptrs` in-place, in which case it moves your `word_ptrs` array and returns the new pointer to where the new location can be found.
- When you are working on `generate_families`, remember that we reuse the words in `word_list` so do not change them.
- When working on `generate_families`, do not explicitly enumerate all the word families: a word of length n has 2^n possible families, but many of those will be empty. (For example, no English word contains three consecutive u's.)

Programming: `wheel.c`

Implement the following functions in `wheel.c`:

- `char **prune_word_list(char **words, int len, int *words_remaining)`
- `char **get_word_list_of_length(char **words, int *len)`

Testing (recommended, but not for credit)

You will find that if you try to write all the functions and only then start testing your program, you will have a very hard time debugging. Therefore, we highly recommend that you write a new program, say test_wheel.c, in which your write several test functions for almost all of the functions you need to write. (Some of the functions are really too simple for this to be necessary.) That way, you can be more confident that each function operates as expected.

When testing, you may need to create a few extra helper functions that hard-code the data structure that a function takes as input.

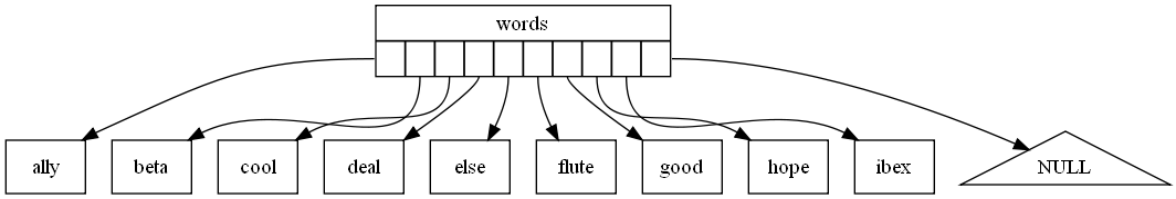
Data Structure Examples

You're working with some rather subtle data structures in this assignment, so I wanted to run through a short demonstration before letting you loose.

Step 0

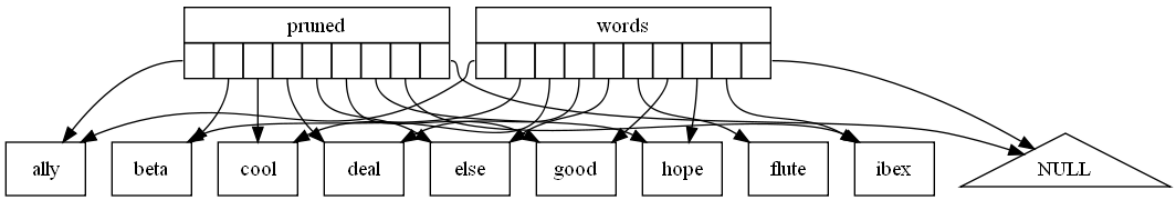
Let's say that our dictionary file contains the following words (almost the same as before, but note the word of length five): ally beta cool deal else flute good hope ibex. Starting from `main` in `wheel.c`, the first thing we do is call `read_words`. This gives us `words` set up with pointers to each of the nine words. (Actually, there is

one extra pointer in `words`: a pointer to `NULL`, which tells you when you've reached the end of the array.) `words` is passed around to many functions, but never changes again after this point.



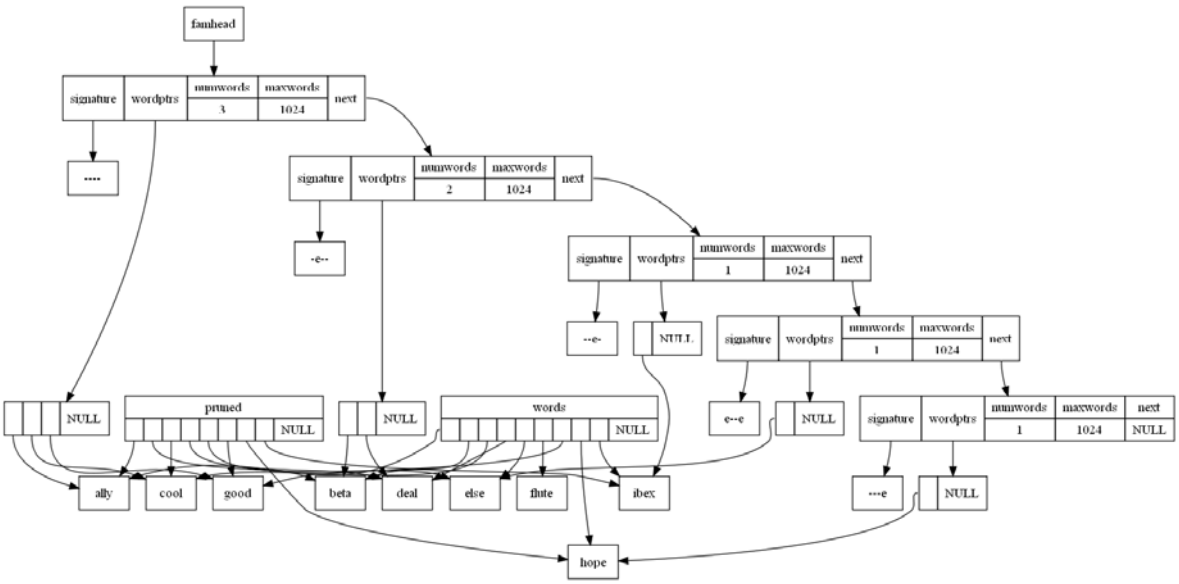
Step 1

Assume that the user wants to play with words of length 4. From within `get_word_list_of_length`, `prune_word_list` will be called with integer 4 and a pointer to the location into which it should place the number of words found. The structure returned by `prune_word_list` (we'll call it `pruned`) looks like an array of pointers to those words from words that are the appropriate length. Notice that the original dictionary is still there, unmodified. In the next game, the user might want to use words of length 5, and we can't have lost those words.



Step 2

The user's first guess is e. `generate_families` gets called with the current `word_list` and the letter e. It generates a linked list of five families; the pointer to the first is referred to as `famhead`. Note that the order of the families in this list is immaterial; this is just one possible ordering.



Marking

We will use testing scripts to evaluate correctness. As a result, it's very important that your output matches the expected output precisely. Unlike A1, which was marked only for correctness, this assignment will be

marked not only for correctness, but also for style and design. At this point in your programming career you should be able to make good choices about code structure, use of helper functions, variable names, comments, formatting, etc.

For this and future assignments, your code may be evaluated on:

- **Memory management**: your programs should not exhibit any memory leaks. This means freeing any memory allocated with `malloc` and `realloc`. This is a very important marking criterion. Use valgrind to check your code.
- **Error-checking**: library and system call functions (even `malloc`!) can fail. Be sure to check the return values of such functions, and terminate your program if anything bad happens.
- **Warnings**: your programs should not cause any warnings to be generated by `gcc -Wall`.

Submission

Please commit to your repository often. We will look at `family.c` and `wheel.c`, so those files must be pushed for us to successfully test your code. You must *NOT* change `family.h`, `reading.h`, `reading.c`, or the provided `Makefile`. We will be testing your code with the original versions of those files. We will run a series of additional tests on your full program and also on your individual functions (in `family.c` and `wheel.c`) so it is important that they have the required signatures. Do not add executables or test files to your repo.

Remember to also test your code on teach.cs before your final submission. Your program must compile cleanly (without any warning or error messages) on teach.cs *using our original* `Makefile` and may not crash (seg fault, bus error, abort trap, etc.) on our tests. Programs that do not compile or which crash will be assigned a 0.