Assignment 5 (Portfolio Assignment):
Implementation of hash map and spell checker

Portfolio Assignment

The purpose of the Portfolio Assignment is to showcase your programming skills to potential employers and colleagues. This is the only assignment in this course that is allowed to be publicly posted online (e.g. GitHub, personal website, etc. …). While this is a great opportunity to publicize your work, it is not required that you post the assignment online.

Prerequisites

There are two parts to this assignment. In the first part, you will complete the implementation of a **hash map**. In the second part, you will implement a **spell checker**.

Part 1: Hash Map

First complete the hash map implementation in hashMap.c. This hash map uses a table of buckets, each containing a linked list of hash links. Each hash link stores the key-value pair (string and integer in this case) and a pointer to the next link in the list. You must implement each function in hashMap.c with the // FIXME: implement comment.

hashMap.h is the header file which defines structs and public functions for your hash table. At the top of hashMap.h you should see two macros: HASH_FUNCTION and MAX_TABLE_LOAD.

Make sure everywhere in your implementation to use HASH_FUNCTION(key) instead of directly calling a hash function. MAX_TABLE_LOAD is the table load threshold (>=MAX_TABLE_LOAD) on which you should trigger resizing the table (double the size of the current hash table capacity).

A number of tests for the hash map are included in tests.c. Each one of these test cases use several or all of the hash map functions, so don't expect tests to pass until you implement all of them. Each test case is slightly more thorough than the one before it and there is a lot of redundancy to better ensure correctness. Use these tests to help you debug your hash map implementation. They will also help your TA grade your submission. You can build the tests with make tests or make and run them with ./tests.

Part 2: Spell Checker

There are a lot of uses for a hash map, and one of them is implementing a **case-insensitive** spell checker. All you need to get started is a dictionary, which is provided in dictionary.txt. In spellChecker.c you will find some code to get you started with the spell checker.

You are provided with a function nextWord() which takes a FILE*, allocates memory for the next word in the file, and returns the word. If the end of the file is reached, nextWord() will return NULL. It is your job to populate the dictionary with the words in dictionary.txt using nextWord().

You can build the program with this command: make spellChecker

The spellchecker program should flow as follows:

1. The user types in a word [only one word (consists of uppercase and lowercase letters only) at a time should be allowed]
2. If the spelling is correct, the following message should be outputted:
   "The inputted word .... is spelled correctly"
3. If the spelling is incorrect, the following message should be outputted:
   " The inputted word .... is spelled incorrectly". Also, 5 potential matches should be outputted like "Did you mean...?" (5 choices)
4. Continue to prompt user for a word until they type "Quit"

One way to implement a dictionary that is used for a spellchecker would be to design it with that purpose in mind from the beginning, i.e. associating a similarity for each word to some base word (maybe "abcdefghijklmnopqrstuvwyz") and then incorporating that into the hash function. However, there are better ways to establish similarity than computing the cosine of the angle between two vectors (strings) to create a list of candidates and further narrowing that list according to substring comparisons.

For example, the Levenshtein distance (https://en.wikipedia.org/wiki/Levenshtein_distance) is a good metric for calculating the distance between the misspelled word and all strings in the dictionary. After calculating the Levenshtein distance between the misspelled word and all words in the dictionary, the 5 best candidates can be determined and printed as suggestions. For this assignment, we will use the Levenshtein distance to compute the similarity between words.

Below is one example you can follow to implement your spellchecker using the Levenshtein distance:

> Step 1: Compare input buffer to words in the dictionary, computing their Levenshtein distance (https://en.wikipedia.org/wiki/Levenshtein_distance). Store that distance as the value for each key in the table.
>
> Step 2: Traverse down the hash table, checking each bucket. Jump out if you find an exact matching dictionary word. Print a message that "The inputted word … is spelled correctly".
>
> Step 3: If the input buffer did not match any of the dictionary words exactly, generate an array of 5 words that are closest matches to input buffer based on the

lowest Levenshtein distance. Print the array including the messages, "The inputted word … is spelled incorrectly", " Did you mean … ? (5 choices)".

Step 4: Continue to prompt user for a word until they type "Quit".

Scoring (100 pts)

- Hash map implementation (60 pts)
- Spell checker implementation (40 pts)

What to Turn In

Turn in the following files to both TEACH and Canvas:

1. hashMap.c
2. spellChecker.c