

Assignment 5: Implementation of hash map, concordance and spell checker

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

Prerequisites

Reading chapter 12, watching this week's lecture on hash tables with chaining, and completing worksheet 38 will better prepare you to tackle this assignment. It may also be helpful to review C file I/O with `fopen()` and `fclose()`.

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` 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: Concordance

The concordance counts how many times each word (case-insensitive) occurs in a document. You will implement a concordance using the hash map implementation from the previous part. Each hash link in the table will store a word from the document as the key and the number of times the word appeared as the value. You must finish the concordance implementation in `main.c`.

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

open the file using `fopen()`, populate the concordance with the words, and close the file with `fclose()`. The file name to open should be given as a command line argument when running the program. It will default to `input1.txt` if no file name is provided.

Your concordance code should loop over the words until the end of the file is reached, doing the following steps each iteration:

1. Get the next word with `getWord`.
2. If the word is already in the hash map, then increment its number of occurrences.
3. Otherwise, put the word in the hash map with a count of 1.
4. Free the word.

After processing the text file, print all the words and occurrence counts in the hash map. Please print them in the format of the following example.

For the input file of: It was the best of times, it was the worst of times.

```
Bucket 0 -> (best, 1) ->
Bucket 1 -> (the, 2) -> (was, 2) -> (it, 2) ->
Bucket 3 -> (of, 2) ->
Bucket 5 -> (worst, 1) ->
Bucket 6 -> (times, 2) ->
```

You can build the program with `make prog` or `make` and run it with `./prog <filename>`, where `<filename>` is the name of a text file like `input1.txt`.

Part 3: 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. It is fairly similar to the code in `main.c`.

You can build the program with `make spellChecker`. FYI:

The spellchecker program flow should be as following -

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".
If the spelling is incorrect, the following message should be outputted -
3. " 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 word until they type "Quit".

One way to implement a dictionary that's used for a spellchecker would probably 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 "abcdefghijklmnopqrstuvwxyz") and then incorporating that into the hash function. But there are better ways (https://en.wikipedia.org/wiki/Levenshtein_distance) to establish similarity than computing the cosine of the angle between two vectors (strings) to create a list of candidates and further winnowed that list according to substring comparisons. So, I would say calculating the Levenshtein distance between the misspelled word and all strings in the dictionary, create 5 best candidates and print them as suggestion.

Below is one example of the steps that you can follow to implement your spellchecker -

Step 1: Compare input buffer to words in the dictionary, computing their 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. And 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 message "The inputted word is spelled incorrectly", " Did you mean ? (5 choices)".

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

Grading Rubric:

- Hash map implementation – 40
- Concordance implementation – 20
- Spell checker implementation – 40

Submission:

Submit the following files to TEACH and Canvas

- hashMap.c
- main.c
- spellChecker.c

You will turn in the above three files via both **TEACH** and **Canvas**. 15% of the grade will be deducted if you have not done that. Make sure your programs compile well using the provided makefile on our ENGR Unix host. Zero credit if we cannot compile your programs. Finally, please don't submit in zipped format to **TEACH**.