Assignment 6: Hash Table implementation and concordance

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 **answer a number of questions** using the concordance program. There is also an **extra credit** opportunity.

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()`.

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. See `hashMap.h` and the accompanied drawing posted with this assignment for clarification. You must implement each function in `hashMap.c` with the `// FIXME: implement` comment.

At the top of `hashMap.h` you should see two macros: `HASH_FUNCTION` and `MAX_TABLE_LOAD`. You are free to change their definitions but know that the default values will be used when grading. `HASH_FUNCTION` is the name of the hash function you want to use. You will change this when answering the written part of the assignment. **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 on which you should resize the table.

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`.

Concordance

The concordance counts how many times each word 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 words and occurrence counts in the hash map. Please print them in the format of the following example above the call to `hashMapPrint()`:

```
best: 1
It: 2
was: 2
the: 2
of: 2
worst: 1
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`.

Written

Submit a pdf or text file answering the following questions:

1. Give an example of two words that would hash to the same value using `hashFunction1` but would not using `hashFunction2`.
2. Why does the above observation make `hashFunction2` superior to `hashFunction1`?
3. When you run your program on the same input file once with `hashFunction1` and once with `hashFunction2`, is it possible for your `hashMapSize` function to return different values?
4. When you run your program on the same input file once with `hashFunction1` and once with `hashFunction2`, is it possible for your `hashMapTableLoad` function to return different values?
5. When you run your program on the same input file once with `hashFunction1` and once with `hashFunction2`, is it possible for your `hashMapEmptyBuckets` function to return different values?
6. Is there any difference in the number of empty buckets when you change the table size from an even number like 1000 to a prime like 997?

Extra credit

There are a lot of uses for a hash map, and one of them is implementing a 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 as following -

1. The user types in a word
2. Potential matches are outputted Like "Did you mean...?" etc
3. Continue to prompt user for word until they type quit

The best 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 "abcdefghijklmnopqrstuvwyz") and then incorporating that into the hash function. 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 5/6 best candidates and print them as suggestion.

Grading

- Compile and style – 20
- Hash map implementation – 50
- Concordance – 20
- Written answers – 10
- Extra credit – 10

Submission

Submit the following files to TEACH and Canvas. Do not zip the TEACH submission. Do not zip the written answers in the Canvas submission.

- `hashMap.c`
- `main.c`
- `spellChecker.c` (optional)
- The written answers in a pdf or text file.