CSCI-1200 Data Structures — Fall 2024
Homework 9 — Spell Correcting Hash Table

In this assignment we will implement and compare the performance of the two strategies for collision resolution: *separate chaining* and *open addressing*. The experiments you will run are based on a simple application for spell checking and spell correction for an input text of English words. We will hash a large dictionary of English words and use simple word-frequency data to suggest replacements for 'misspelled' words from the text that are not present in the dictionary.

Please carefully read the entire assignment before beginning your implementation.

English Word Dictionary with Frequency Data

To facilitate testing for this assignment, you are provided with dictionary files in a variety of sizes between 10,000 to approximately 500,000 words. On the right is a small sample from the `words_with_frequency_10k.txt` file, which has the 10,000 most frequently used words and their frequencies relative to "`the`", the most common word.

```
...
thanks              0.003950509984
thanksgiving        0.000360839515
that                0.146959412869
thats               0.000433708400
the                 1.000000000000
theater             0.001316001550
theaters            0.000409829097
theatre             0.001473699876
...
```

NOTE: The dictionary data for this assignment was collected and combined from a few sources:

- Linux and MacOS systems have a file `/usr/share/dict/words` that contains a simple list of words.
- A Kaggle dataset based on the Google Trillion Word Corpus:
  https://www.kaggle.com/datasets/rtatman/english-word-frequency
- Data from a one billion word Corpus of Contemporary American English (COCA)
  https://www.wordfrequency.info/

NOTE: This is a challenging problem due to contractions, proper names, possessive nouns, verb conjugation, and slang. Furthermore, this data is scraped from the web with imperfect parsing, and the computed frequency skews to modern online usage and is thus not a great tool for spell-checking texts from the Elizabethan era.

A Customizable Hash Function for English Words

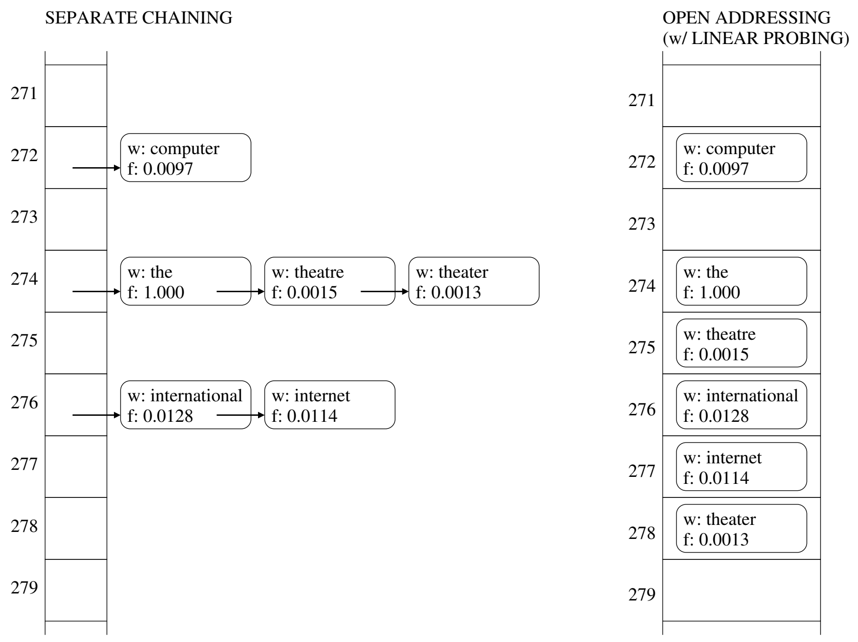
The provided code includes a functor class, `WordHashFunction`, which has a constructor that takes 2 integer arguments: `hash_prefix` and `hash_suffix`. These variables will allow us to control the number of collisions and observe the impact of these collisions on the performance of our hash table.

If we hash only a subset of the characters in the word, for example if we hash only the first 5 letters of the word (by specifying that `hash_prefix = 5`), we are guaranteed to have collisions in the hash table. In our 10k dataset, the most common 5 letter prefix is `inter` – the top five words when sorted by frequency are shown on the right. Can you guess the most common prefix of length 3? Or 5? Or other small integer values? What about the most common suffixes (`hash_suffix`)? What is the most common combined prefix and suffix?

```
maximum bucket contains:
  0.01277840  international
  0.01140123  internet
  0.00519855  interest
  0.00256346  interface
  0.00233390  interesting
 ...
```

Hash Table with Different Collision Resolution Strategies

To compare the two methods for collision resolution, you will complete the `HashTable` class, which can be configured for either the separate chaining *or* open addressing methods of collision resolution. The constructor for the `HashTable` class takes in multiple arguments: an integer `table_size`, an instance of the `WordHashFunction` functor class, and booleans `use_open_addressing` and `use_quadratic_probing`. *NOTE: quadratic probing is extra credit.* We discussed both of the collision methods in lecture. Below is a diagram illustrating the plausible result from hashing the first three letters of each word:



With *separate chaining*, the hash table is a simple array of pointers, and all items that hash to the same location are stored as a linked list. You may implement this with the STL `list` class or with a custom singly-linked node class. For *open-addressing* the hash table is an array directly storing the data. With *linear probing* the collisions are resolved by spilling into the next slots in the array. When nearby hash values have larger quantities of items mapped to them, the collisions overlap and may have significant impact on the performance. In this small illustration, the open-addressing example has two *non-empty sequences*, one with length 1, and one with length 5.

Measuring the Performance of a Hash Table Configuration

To understand the severity and impact of collisions for different configurations, your program will print out to `std::cerr` simple statistics about the hash table. Here are two sample command lines and corresponding error/information stream output:

```
./main.out words_with_frequency_10k.txt --hash_prefix 5 --table_size 30000
./main.out words_with_frequency_10k.txt --hash_prefix 5 --table_size 30000 --open_addressing
```

```
Hash Table Statistics:
Using Separate Chaining
# entries                          =      10000
# buckets                          =      30000
# empty buckets                    =      24301 (81.00%)
# single entry buckets             =       3795 (12.65%)
average bucket count               =      0.333
maximum bucket count               =         31
maximum bucket contains:
  0.01277840  international
  0.01140123  internet
  0.00519855  interest
  0.00256346  interface
  0.00233390  interesting
  0.00226986  internal
  0.00221657  interested
  0.00169600  interests
  0.00156698  interactive
  0.00140175  interview

hash table creation time           =      0.020 seconds
maximum resident set size (RSS)    =      2.490 MB
```

```
Hash Table Statistics:
Using Open Addressing
# entries                          =      10000
# locations                        =      30000
# empty locations                  =      20000 (66.67%)
# non-empty sequences              =       3402
longest non-empty sequence         =        186
average non-empty sequence length  =      2.939

hash table creation time           =      0.011 seconds
maximum resident set size (RSS)    =      2.212 MB
```

The *maximum resident set size (RSS)* is a measurement of the peak total memory usage of your program.

Once the functionality of your `HashTable` implementation is debugged, you should explore the configuration options and the impact on running time and memory usage. What does or does not match your expectations? How could configuration tuning have real-world impact on applications that use hash tables?

NOTE: For this assignment we will not implement automatic table re-sizing. If the `table_size` requested on the command line is too small for open addressing, your program should exit with an error.

Application: Spell Checking & Word Replacement Suggestions

We can use this hash table to check the spelling of every word in an input text and flag words that are not in the dictionary. The "Alice in Wonderland" input file and thousands of other public domain classics are available from Project Gutenberg (https://www.gutenberg.org/). A portion of the output of a command line checking such an input file is shown below left. Note that the detailed output of the misspelled words is sent to `std::cout`, so we can separate it from the performance data.

```
./main.out words_with_frequency_200k.txt
      --table_size 300000 --check_spelling alice_in_wonderland.txt
```

```
Total mispelled words              = 89
Unique mispelled words             = 47

<snip>
MISPELLED: draggled 1 time(s)
MISPELLED: drawling 3 time(s)
MISPELLED: duchesss 3 time(s)
MISPELLED: eaglet 3 time(s)
MISPELLED: footmans 1 time(s)
MISPELLED: forepaws 1 time(s)
MISPELLED: hearthrug 1 time(s)
MISPELLED: hjckrrh 1 time(s)
MISPELLED: html 1 time(s)
MISPELLED: inkstand 1 time(s)
MISPELLED: jurymen 4 time(s)
MISPELLED: maynt 1 time(s)
MISPELLED: morcar 2 time(s)
MISPELLED: muchness 3 time(s)
MISPELLED: neednt 3 time(s)
MISPELLED: ootiful 4 time(s)
MISPELLED: pattering 3 time(s)
<snip>
```

```
MISPELLED: capering 1 time(s)
  1 0.0005894878 catering
  1 0.0000137156 tapering
  1 0.0000015527 papering
MISPELLED: drawling 3 time(s)
  1 0.0008673666 drawing
  1 0.0000929780 crawling
  1 0.0000104411 trawling
  1 0.0000039800 brawling
  1 0.0000022236 drawline
  2 0.0004311697 drawings
  2 0.0000387441 rawlings
MISPELLED: pattering 3 time(s)
  1 0.0000178061 patterning
  1 0.0000109871 battering
  1 0.0000028581 mattering
  1 0.0000021605 nattering
  1 0.0000013379 puttering
  1 0.0000013051 spattering
MISPELLED: muchness 3 time(s)
  no replacement suggestions
```

Beyond simply detecting misspelled words, we can make suggestions about replacement words. How can we efficiently find words that are close in spelling? If we have created the hash table using a `hash_prefix` *and* if we believe the spelling error is at the end of the word (after the prefix), then the correct spelling of the word will hash to the same value, so we can simply look through all words in that bucket to find the most similar word. However, this strategy does not work if the word is short or if the spelling error is within the prefix region. Also this strategy has a downside of creating a hash table with a large number of collisions.

Instead, when the `--skip_letters_while_hashing` argument is specified, we will proactively place every dictionary word into multiple buckets, anticipating potential misspellings. Which misspellings? We will loop over the letters in the word, and one at a time drop or skip each letter, computing the hash of the modified words. For example, we will also insert the correctly spelled dictionary word "`patterning`": at the hash locations of "`atterning`" "`ptterning`" "`paterning`" "`pattrning`" "`pattening`" "`pattering`" "`patternng`" "`patternig`" and "`patternin`".

Furthermore, when we search for replacement options, we can not only search for the misspelled word, but also one-by-one drop each letter of the misspelled word, hash those options, and search in all of those buckets. Here is a sample command line we will use to run this algorithm:

```
./main.out words_with_frequency_200k.txt
      --table_size 2000000 --skip_letters_while_hashing
      --check_spelling alice_in_wonderland.txt --suggest_replacements
```

A portion of the output for this command is shown above right. Note that because we are inserting every word multiple times with the `--skip_letters_while_hashing` command, we will want to use a much bigger hash table, especially if we are using the open addressing collision resolution strategy.

When the `--suggest_replacements` command line argument is included, these candidate words will be collected and organized first by the *edit distance*, and secondarily by frequency. The `EditDistance` function is in the provided code for this assignment. It calculates how many neighboring letter *swaps* (e.g., "recieve" → "receive"), letter *replacements* (e.g., "seperate" → "separate"), letter *inserts* (e.g., "mispelled" → "misspelled"), and letter *deletes* (e.g., "whereever" → "wherever") are necessary to transform the misspelled word into the dictionary word.

Assignment Requirements, Hints, and Suggestions

- The provided code for this assignment includes command line argument parsing, input parsing, the hash function, and edit distance code. You may edit any of the provided code, but your program should still match the expected input and output formats.
- A significant portion of this homework is about experimentation, observation, data collection, and writeup. Answer the questions posed in this handout in your README.txt and create and answer your own questions as your work through the assignment.
- Be sure to make up new test cases to fully test your program. Use the template `README.txt` to list your collaborators, your time spent on the assignment, and any notes you want the grader to read.