# Spellchecker application

Itay Knaan-Harpaz
knaan.harpaz@gmail.com

July 10, 2019

## 1 Abstract

In this homework assignment, we had to program an application that reads a dictionary file and an input file, and finds all words in the input file that does not exists in the dictionary, and prints them in lexicographical order. The application suggests corrections for words that are not a part of the dictionary based on a few heuristics.

The algorithm works by creating a hash tale of all words in the dictionary, then reading an input file, and placing all words in it in a red-black tree. After reading the input file, the program iterates over the tree inorder and deletes all words in it that are not a part of the dictionary. Finally, the program iterates over the tree again inorder, prints every word it encounters (those words are not in the dictionary) and attempts to find the word the user meant to write instead of the one written.

## 2 Application flow

1. The application creates an empty hash table for the dictionary.
2. The application reads all words from the dictionary and inserts them into a hash table, ignoring duplicates (there are duplicates as the dictionary implementation is case insensitive and ignores `.` in acronyms, etc.).
3. For each input file:
   (a) The application creates an empty hash table for the input words (see discussion in the algorithm section)
   (b) The application creates an empty lexicographically-sorted-string-keyed red-black tree for unique input text words.
   (c) The application reads all words from the input file. If a word is in the input-file hash table, ignore it. Otherwise, insert it into the input hash table and the red-black tree.
   (d) Iterate over all words in the red-black tree inorder, and delete those that appear in the dictionary's hash table.
   (e) Iterate over all remaining words in the red-black tree inorder, and attempt to suggest correction, checking against hash table.

## 3 Algorithm analysis

We will analyze average-case performance of the program (analyzing worst-case when using hash tables is meaningless). Assume all words are of length up to $w$. Hashing the word is an $O(w)$ operation, and we assume that the result of the hashing algorithm is uniform in the co-domain of our hash function (see the section discussing the hash function). Let $s$ be the size of the hash table, and let $n_{dict}$ be the number of words in the dictionary and $u_{dict}$ be the number of unique words in the dictionary. The average time complexity of insertion into the hash table is $O(w + 1 + \frac{u_{dict}}{s})$. Hash collisions are implemented by a linked list at each entry, as this was the simplest to implement, and the various ways of handling hash collisions does not change the asymptotic complexity. The number of words in our dictionary is around 460K words, so by choosing a hash table size of 512K we ensure an average collision amount of less then half per hash table entry, and the entry insertion is $O(w)$.
As we insert all words from the dictionary into the hash table, the total asymptotic time complexity of dictionary reading is $O(n_{dict} \cdot w)$.

We use a red-black tree with strings as keys. The time complexity of lexicographically comparing 2 strings is proportional to the number of common initial characters - comparing "airplane" with "encyclopedia" is much quicker then comparing "accommodate" and "accompany", as the comparison has to detect the first different character. Assuming all words has a common letters distribution, for every fixed number of layers - $l$ - in the red-black tree, there is on average another common character between the inserted word and the comparisons that needs to be done to insert it into the tree. Therefore, as there are $O(\lg n)$ comparisons when inserting a word, and $O(\lg n)$ layers in a red-black tree, the asymptotic complexity of inserting a word into a lexicographically keyed red-black tree is $O(\lg n \cdot (1 + \frac{\lg n}{l})) = O((\lg n)^2)$.

Let there be $n_{text}$ words in the input text, and $u_{text}$ unique words in it. Attempting to inserting a word that was already in the tree into it again, results in no difference to the resulting tree, but is still the same complexity as inserting a new word into the tree. Therefore, the asymptotic time complexity of inserting a whole text into the tree is $O(n_{text} \cdot (\lg u_{text})^2)$. In large texts, the amount of unique words can be quite high, and as an optimization I added a second hash table to check whether a word was already seen before inserting it into the tree. This turns the asymptotic time complexity into $O(n_{text} \cdot w + u_{text} \cdot (\lg u_{text})^2)$ (same hash table analysis as previous paragraph). In practice this gives us a large performance boost when processing large texts, as there are many more words than unique words, and the $O(w)$ time to check whether a word is in the hash table is much quicker then the $O(\lg u)^2$ process of finding whether it is in the tree.

Iterating over a binary tree is an $O(n)$ process, checking whether a word is in the hash table is an $O(w)$ process, and deleting a word from the hash table is an $O(\lg n)$ process. Therefore the time complexity of the third stage - iterating over the binary tree and deleting words that are in the hash table, if of asymptotic time complexity $O(u_{text} \cdot (w + \lg u_{text}))$.

Let there be $n_{bad}$ words in the text that are not in the dictionary - this is the amount of words in the tree at this stage. For every words, there are $O(w)$ correction suggestions, the creation of each is an $O(w)$ process (see heuristics subsection), and each of whom is checked against the hash table at an $O(w)$ time complexity. Therefore, the time complexity of finding correction suggestion to all words is $O(n_{bad} \cdot w^3)$.

To sum up, the time complexity of all program stages together is:

$$O(n_{dict} \cdot w + n_{text} \cdot w + u_{text} \cdot (\lg u_{text})^2 + u_{text} \cdot (w + \lg u_{text}) + n_{bad} \cdot w^3)$$

$$= O(w \cdot (n_{dict} + n_{text} + u_{text}) + u_{text} \cdot (\lg u_{text})^2 + w^3 \cdot n_{bad})$$

## 4 Hash function

I use the following hash function as a hash function: Initialize an accumulator to the value $0x99999999$ as a base (having a good mix of 1's and 0's), and given a string $s$ - iterates over the characters of $s$, perform the following operation per character - multiply the accumulator by a large prime number (1000003, the first prime number over a million was chosen), xor the least significant byte of the hash result with the top byte of every dword in the accumulator (to move useful entropy bits from the high-end unused part of the hash result to the low byte), and add the next character. Using debug prints shows there are not too many hash collisions, but I do not know how to prove it...

The hash algorithms discussed in the book assume the input is a giant number - what would require us to use a bignum library, and I wanted to avoid having to do that. Also the XOR and MUL operations are very efficient on words in the processors, making the hash calculation quite fast.

As the loops iterates over characters from the input string, the time complexity of running the hash function is $O(w)$ where $w$ is the length of an input word.

## 5 Suggestion heuristics

We consider the following heuristics as a decreasing priority when suggesting corrections:

1. Words that are the same excpet for a letter appearing twice
   (e.g. 'businness' $\rightarrow$ 'business').

2. Words that are the same except for a swapped pair of letters
   (e.g. 'buisness' $\rightarrow$ 'business').

3. Words that are the same except for a double letter written once
   (e.g. 'busines' $\rightarrow$ 'business').

4. Words that are the same except for one homophonic letter
   (e.g. 'buciness' $\rightarrow$ 'business').

Each of these operations require scanning through the string a fixed amount of times, and create $O(w)$ new strings as suggestions. The creation of each suggestion is $O(w)$ as it requires copying almost all characters from the original string into a new string, and performing some $O(1)$ change in the middle. checking each suggestion against the dictionary is an $O(w)$ operation. Therefore, running the suggestion heuristics on a single word is $O(w^3)$ operation.

Suggestions heuristic in action as seen when running the program on the sample sentence:

```
The following words are not in the dictionary:
caled
Did you mean: 'called'?
haerd
Did you mean: 'heard'?
ideea
Did you mean: 'idea'?
kar
Did you mean: 'kra'?
lookking
Did you mean: 'looking'?
mayby
Did you mean: 'maybe'?
polise
Did you mean: 'polies'?
woried
Did you mean: 'worried'?
```

Turns out `kra` is an English word, and according to the priority of stated heuristics, it is prioritized higher than `car`...

## 6 Software design

As a separation of data structure logic from hashing and comparisons logic, the data structures receive these functions as parameters at construction time.

The implementation of the data structure is independent of the type stored in it, therefore they are implemented as class templates, with the stored data type as template parameter.

The hash table is implemented using a vector container inside, as it preallocates the specified amount of entries (hash table size), and accessing an element of it is $O(1)$ as specified in the C++ language standard. Hash collisions are implemented by chaining the different keys with the same hash in a list from the standard library, as it admits the same complexity as linked lists discussed in hash table collisions treatment in the book.

The red-black tree was implemented using smart pointers as described in the C++ language standard. This way of constructing it allows us to move nodes around the tree in $O(1)$ complexity - same as setting a pointer - and also allows us not to manually allocate and free memory for nodes inserted into the

tree. Each node has a weak pointer to it's parent and to the tree object - these pointers does not hold these objects in RAM, so when a parent is deleted for e.g., it will be freed even through it's kids are holding pointers to it. A parent node holds a shared pointer to it's child nodes, keeping his child in RAM as long as it is in the RAM.
The initialization of such data structure requires a creation of shared pointer to it at object creation, and passing it's pointer around to sub objects. This requirement lead me not to allow manual creation of these objects by the user (by having a privately declared parameter type in constructor), and allowing their creation only via a static function that creates a smart pointer to them and passes it in at creation time.

## 7 Performance results

The program can load a dictionary containing all words of the English language and scan the whole text of the classic book frankenstein in 2.7 seconds on my 5 years old PC, seems fast enough ☺.

Getting this performance required adding a hash table to identify unique words before inserting into the red-black tree (see algorithm section) and some tweaking of optimization flags...
The English language dictionary is taken from
`https://raw.githubusercontent.com/dwyl/english-words/master/words.txt`
The Etext of frankenstein is taken from
`http://umich.edu/~umfandsf/other/ebooks/frank10.txt`

## 8 Toolchain

The application is written in C++17, and was tested using the GNU Compiler Collection (GCC), version 8.3.0, invoking the 'g++' command. The application uses features from the C++17 standard and would not compile on non standard-compliant compilers.

The program uses the following optimization flags to achieve optimal performance:
```
-flto
-fwhole-program
-Ofast
-march=native
```
Those flags enable same-file software structure trans-formative optimizations (`-Ofast`), CPU-specific optimizations (`-march=native`) and link time optimizations (`-flto, -fwhole-program`).

The application is warning free using the following warnings flags:
```
-Wall
-Wextra
-Wshadow
```

```
-Wstrict-aliasing
-pedantic
-Wc++17-compat
-Wduplicated-branches
-Wduplicated-cond
-Wempty-body
-Wtautological-compare
```

To ensure software correctness and standard compliance.

The `-std=c++17` flag is used to specify the version of the language standard used, and the flag `-DNDEBUG` is used to turn off debug assertions in the standard library that slows the program down.

The application is built using the `make` tool, tested on version 4.2.1 (but should probably work on most versions of it from the last 20 years as no advanced features are used).

## 9 Building and usage

In order to build the application, invoke the `make` command in the `spellChecker` directory. This command will compile each of the `.cpp` files in the directory and then link them into an executable named `spellChecker`.

The application takes command line argument, the first of whom is a path to the dictionary to be used (a dictionary is included in project), an the rest are paths of files to be checked.
For e.g., to check the files `file1.txt, file2.txt, file3.txt` against the dictionary `english-dict.txt`, run the following command:

```
./spellChecker english-dict.txt file1.txt file2.txt file3.txt
```

The sample output files use the `time` command as well to clock the runtime it takes the program to complete it's execution.

## 10 Libraries used

All external used code is part of the standard library of the C++ language. At times there exists standard libraries that perform tasks written in the application as programming the hash table and red-black tree data-structures was part of the assignment.

The following libraries are used:

```
algorithm
exception
fstream
functional
iostream
list
memory (shared_ptr, unique_ptr, weak_ptr)
```

```
sstream
string
tuple
utility
vector
```

## 11 Appendix: Description of files in package

`Readme.pdf` - this file.
`Readme.tex` - sources for this file.
`App.cpp` - The file containing the logic of the application.
`App.h` - Header of application logic.
`Autocorrect.cpp` - Implementation of suggestions heuristics.
`Autocorrect.h` - Header of suggestions heuristics.
`english-dict.txt` - A dictionary containing all words of the English language.
`Exceptions.h` - Header defining exceptions thrown in application.
`FileReader.cpp` - Implementation of file reading, word splitting and processing.
`FileReader.h` - Header for file reading, word splitting and processing.
`frankenstein-run-output.txt` - Result and timing of running application on Project Gutenberg Etext of Frankenstein.
`frankenstein.txt` - Project Gutenberg Etext of Frankenstein.
`hash.cpp` - Implementation of hash function.
`hash.h` - Header of hash function.
`Hashtable.h` - Header of hash table definitions.
`Hashtable.hpp` - Header of hash table template implementation.
`main.cpp` - Program entry point, argument parsing and invoking of the application.
`Makefile` - Build automation definitions used by the `make` command.
`RBTree.h` - Header of red-black tree.
`RBTree.hpp` - Header of red-black tree template implementation.
`Readme.md` - Readme explaining app building and using.
`sample-sentence-run-output.txt` - Output and timing when running the program and checking the given sample sentence (from the homework assignment).
`sample-sentence.txt` - sample sentence (from the homework assignment).