Comparing data structures for searching words

Akmalkhon Khashimov

August 8, 2021

Contents:

1. Description

In this section different data structures will be discussed and the specifics/details of their implementation.

1.1 Linear list

Taken from C++ standard library, this data structure is a simple linked list with **O (1)** insertion time complexity, however **O(n)** traversal time.

1.2 Balanced Binary Search Tree

This data structure optimizes the search time by balancing the tree data structure it uses for storing strings with time complexity of **O (log n)** for insertion/deletion/search.

1.3 Trie

Trie is an efficient information re***Trie***val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well-balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in **O(M)** time. However, the penalty is on Trie storage requirements.

The trie in this project was implemented as simple as it is with only two functions available: insert and search.

1.4 Hash map

A simple data structure that uses the principle of hashing to store strings. The hash function implemented here is the same as in Rabin Karp algorithm. As for the choice between linear probing and linked list hash map, I chose linked lists, for the sake of efficiency. Complexity: **O(n)** for calculating hash, where n – size of the string) and **O(m)** the average length of the linked lists in the hash map.

2. Methodology

2.1 Alphabet

For the size of the alphabet, I used 26. For this reason, I implemented the code to convert all the characters in the text to lowercase and if the character is not a letter then convert it to space.

2.2 Wordlist and Text

As for the wordlist, I downloaded it from the internet/GitHub (link in the references below). The text is taken from one of the Harry Potter books.

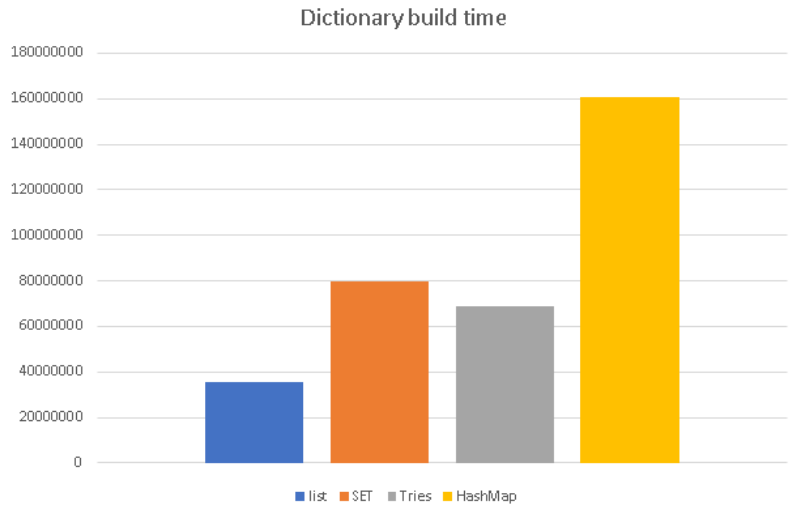
2.3 Time measurements

For the dictionary creation, the time was measured overall, not each time refreshing during the insertion, but rather total time taken for the whole process, which is getline() function from wordlist and insertion.

As for the search, it was specifically measured with refreshments each time before search and summing it up for each word with the total number of words **4152**.

3. Results

Dictionary build time

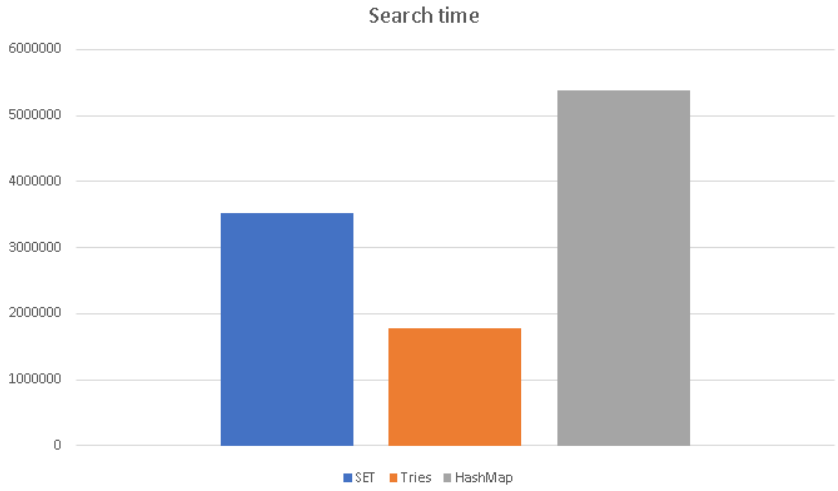


The bar graph above demonstrates the time (in nanoseconds) it took for each of the data structure to be built.

As it is obvious, the hash map takes too much time to be built, however the linear list takes the least amount of time to be built, with BBST and Tries in the middle and tries

having slight domination over BBST. In summary, the hash map data structure build time is essentially high.

Search time



In the bar chart above, the vertical axis represents the amount of time (in nanoseconds) it took to search 4152 words in the corresponding data structure.

The linear list data structure was not included into the graph, as the numbers of it distorts the graph and makes it incomparable, because it takes too much time to search in linear list.

To summarize, the tries data structure is leading in search time, with BBST and hash map to follow up accordingly.

It is also worth to mention that each of these data structures have their own best applications in different cases and these numbers are just generalization.

Success rate

I also measured the success rate for each of the algorithms, and the numbers turned out to be the same. There are 196 fails out of 4152, which is not significant. This is possibly connected with the fact that there are names/slang/non-typical words in the text.

4. References

https://github.com/jeremy-rifkin/Wordlist