A Comparison of Spell Checking Implementations

Mohamed Samine

September 01, 2021

1 Introduction

The purpose of this paper is to explore the spell-checking problem. The problem is to find all misspelled words in a text.

In this paper, we compare four algorithms:
Brute Force (Naive), Balanced Binary Search Tree (AVL Tree), Trie (Prefix tree), and Hash tables.

1.1 Brute Force

For constructing the dictionary, the naive implementation converts the list of words into an array, it achieves its goal of finding the misspelled words in the text by comparing each word encountered in the text with all words in the dictionary array, if the word doesn't exist in the dictionary array it means that it is indeed misspelled.

Time complexity

To make sure that a word is misspelled, naive implementation loops over the dictionary array to make sure that it doesn't exist in the dictionary, which takes $O(n)$ with n being the number of words in the dictionary. Considering the comparison of the word we are spell checking with each word in the dictionary, which is taking $O(m)$ with m being the length of the word, the overall complexity of the naive implementation will be $O(n*m)$.

1.2 Balanced Binary Search Tree (AVL Tree)

Binary Search Tree

A binary search tree is a binary tree that allows maintaining a sorted list of items. As its name is a binary search tree, each node in the tree can have a maximum of two children. For a binary tree to be considered a BST, the tree should have the following properties:

1. All nodes of the left sub-tree are less than the root node.
2. All nodes of the right sub-tree are more than the root node.
3. Both sub-trees of each node are also BSTs.

AVL Tree

AVL tree belongs to the family of self-balanced binary search trees, the balance is enforced by rotations that enforce the AVL property which is:

- No sub-tree's height differs by more than one from its sibling.

Thus, when we insert an item, we re-balance the sub-tree containing the new item with one of its ancestor's siblings.

We perform a right rotation when the balance factor in a node is -2, and the balance factor in its left node is -1.

We perform a left rotation when the balance factor in a node is +2, and the balance factor in its left node is +1.

We perform a left rotation on the left child then a right rotation on the node when the balance factor in the node is -2, and the balance factor in its left node is +1.

We perform a right rotation on the right child then a left rotation on the node when the balance factor in the node is -2, and the balance factor in its left node is +1.

Spell Checking Implementation

BBST constructs the dictionary by adding the words to a balanced binary tree using their alphabetic order.

To find out if a word is misspelled the algorithm search for it in the balanced binary tree, if it is not found it means that the word is misspelled.

Time complexity

The BBST implementation spell checks a word by traversing the dictionary tree to make sure that this latter doesn't exist, as we already know that an AVL tree is a self-balanced binary search tree, which means the height is guaranteed to be $O(\log (n))$ where n is the number of words in the dictionary. The overall complexity of the BBST spell checking implementation is $O(\log (n)*m)$ where m is the length of the word we are spell checking.

1.3 Trie / Prefix Tree

To construct the dictionary, The **prefix tree** also known as a **Trie,** takes each word from the list and creates a new node for each character of that word, with a link between the node/character and the next character's node which is considered a child node. The node registered for the last character of the word is marked as the end of the word.

The implementation of prefix tree is using a BST container to save the child nodes, each node in the prefix tree has a value, a BBST as a dynamic container to store its child nodes, an end of word flag, a left node which is alphabetically inferior to the node and a right node which is superior to the node.
To spell-check a word, the prefix tree goes through the word character by character and checks at each level if the character exists in the BBST tree, if it does it moves to check the next character in the BBST of the found node, until it either finds the node matching the last character with an end of word flag or fails to.

Time complexity

As we are using a BBST as a dynamic container for child nodes, the time complexity of searching for a node in BBST is $O(\log(s))$ where s is the number of child nodes (bounded by the number of alphabets used). Additionally, we are checking the word character by character, that is $O(m)$ where m is the word length. As a result, the overall time complexity of the implementation is $O(m*\log(s))$ where m is be the average word length and s is the number of child nodes that a node have (bounded by the number of the alphabets).

1.4 Hash Table

A hash table is a data structure that stores elements in an associative manner. In a hash table, data is stored in an array format, where each value has its unique index value.

As a result, insertion and search operations are very fast irrespective of data size. Hash Table uses an array as a data container and a hash function to generate the index where the element should be inserted or located in case of search.

The hash table implementation constructs the dictionary by inserting all the words in the list, it uses the multiplication method to generate the hash of the inserted words with the constant $A = (\sqrt{5} -1)/2$.

For collisions handling, the implementation uses open addressing with double hashing. The secondary hash function used is relatively prime to the array capacity, which is why it is always returning an odd value while the capacity of the array is always a power of 2.

To examine the spelling of a word, the implementation converts this latter into an integer using the sum of each character ASCII code multiplied by its position and a constant prime number, taking that integer value as a key, the hash function returns the index of the word. With the index generated by the hash function the implementation checks if the word exists at

that position, if it points to a value different than the word searched for, the hash table uses the secondary hash function to generate the step size to use for looping through the array. At each iteration a check is conducted to verify if the value at the index generated is equal to the word being spell checked, if at any point of the search the value at the index position is NULL it means that the word doesn't exist in the hash table, and is indeed misspelled based on the dictionary fed to the algorithm.

Time complexity

Finding a word in a hash table takes $O(1)$ in the best case and $O(n)$ in the worst case where n is the number of elements in the hash table.

2 Methodology

The algorithms were implemented using the JavaScript programming language.

A list of 370103 English words was provided to the algorithms as a dictionary.

The comparison was conducted on an increasing text length starting with 2000 characters up to 57586 characters.

Each algorithm was given the same list of words and text to spell check, and each test was repeated 5 times, the average time of the repeated tests was logged into a .csv file for each algorithm.

3 Results

The chart in **Figure 1** shows the average time it took each implementation to build the dictionary using a list of 370103 English words. The slowest is the Hash Table implementation this can be explained by the need for the Hash Table to rehash the elements when the threshold is reached, which is an expensive operation, we can also not ignore collisions that take time to be handled as well. The second slowest is the Prefix Tree, as it processes the words character by character and creates a node for each, and adds it to the BBST of the previous character's node or the root node in case of the first character.

The time BBST took to construct the dictionary is due to its need to build the binary tree and ensure it stays balanced. As for the Naive implementation, it is the fastest in building the dictionary because it's simply filling the array with values without extra manipulation or computation.

Figure 2, the chart shows us a representation of the time it took each implementation to find all the misspelled words in an increasing text, it can be observed from the chart that the Naive implementation is by far the slowest in terms of spell checking, that is explained by the need

to compare each word being spell checked with all the words in the dictionary to be able to ensure that the word is indeed misspelled.

In **Figure 3**, the naive implementation was omitted to be able to compare the remaining implementations. We can observe from the chart that Hash Table is the fastest implementation to find the misspelled words, this can be explained by its search time complexity of $O(1)$ in the best case, which is giving it a clear advantage over the other implementations. Next in the list, we find the Prefix Tree, which is performing better than BBST, this is due to its technique of reading the word each character at a time, and checking if the node for that character exists in the root's node BBST or the previous character's node BBST in case of characters at a position > 1, as in a Trie the BBSTs in each node has a height bounded by the number of alphabets $\log(s)$, this allows Prefix Tree implementation to conclude without many comparisons that a word is misspelled if it is the case, which gives it an advantage over the BBST implementation that has to do $\log(n)$ comparisons to make sure that a word is misspelled, where n is the number of words in the dictionary.

The slowest implementation from the list is the naive implementation, and this is due to its need to compare each word being misspelled to all dictionary words before it can conclude that the latter is indeed misspelled, which is not efficient.

4 Conclusion

Based on the test conditions presented above, we can conclude that among the implementations presented in this paper, the most optimal one for the spell checking problem is the Hash Table implementation. Furthermore, it should be mentioned that if features such as auto-completion or suggestions are to be considered the Trie implementation should be the way to go.

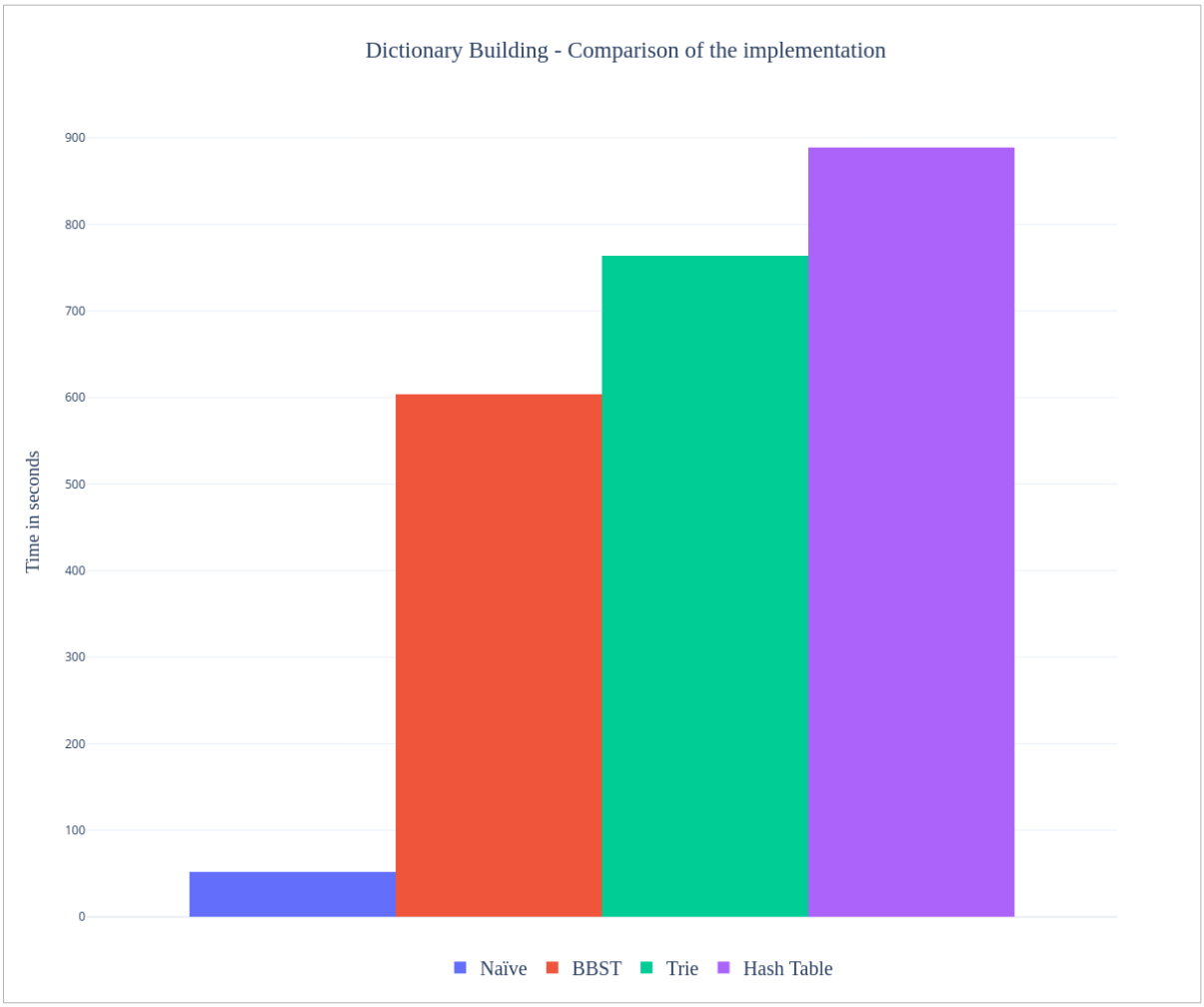

Figure 1: Comparison of the average time each implementation took to build the dictionary (370103 word).

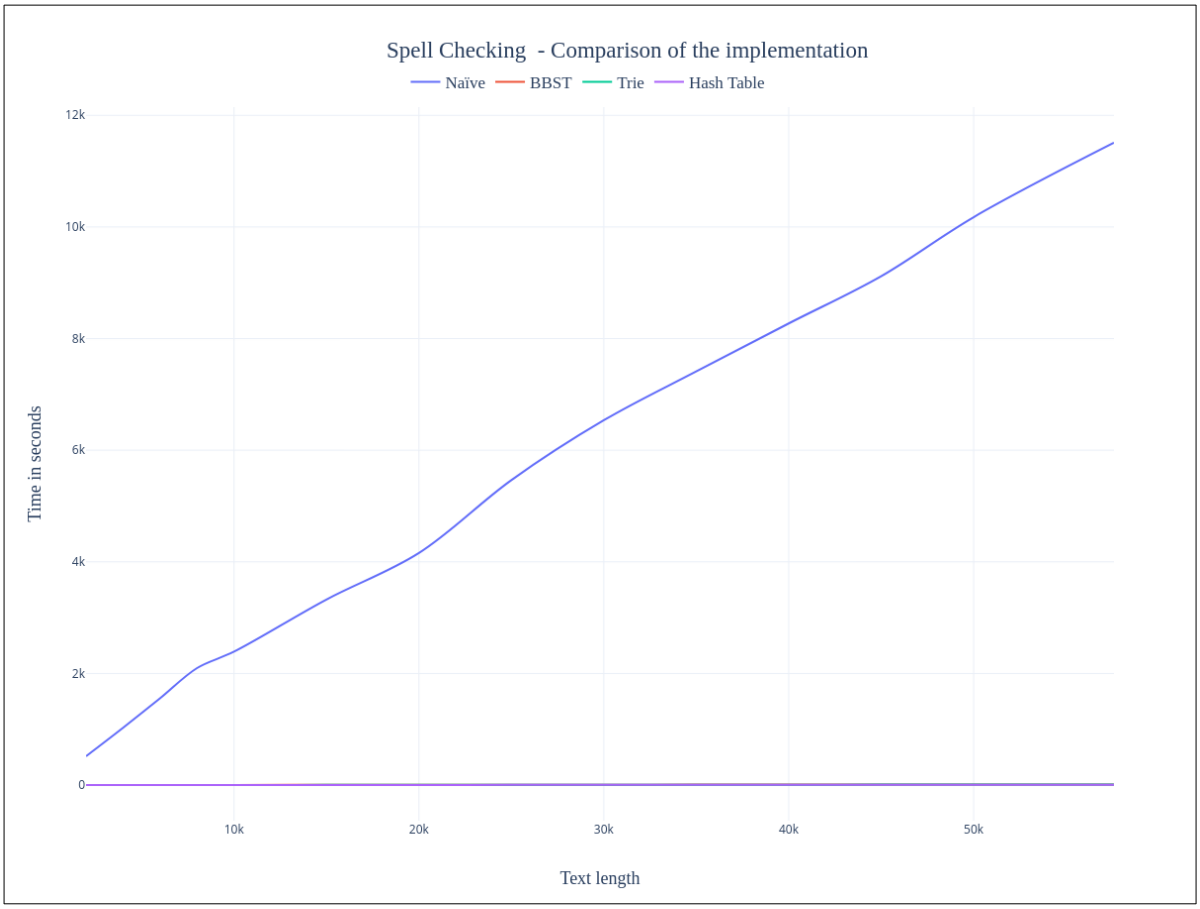

Figure 2: Comparison of the average time each implementation took to spell check an increasing text (2k – 57k).

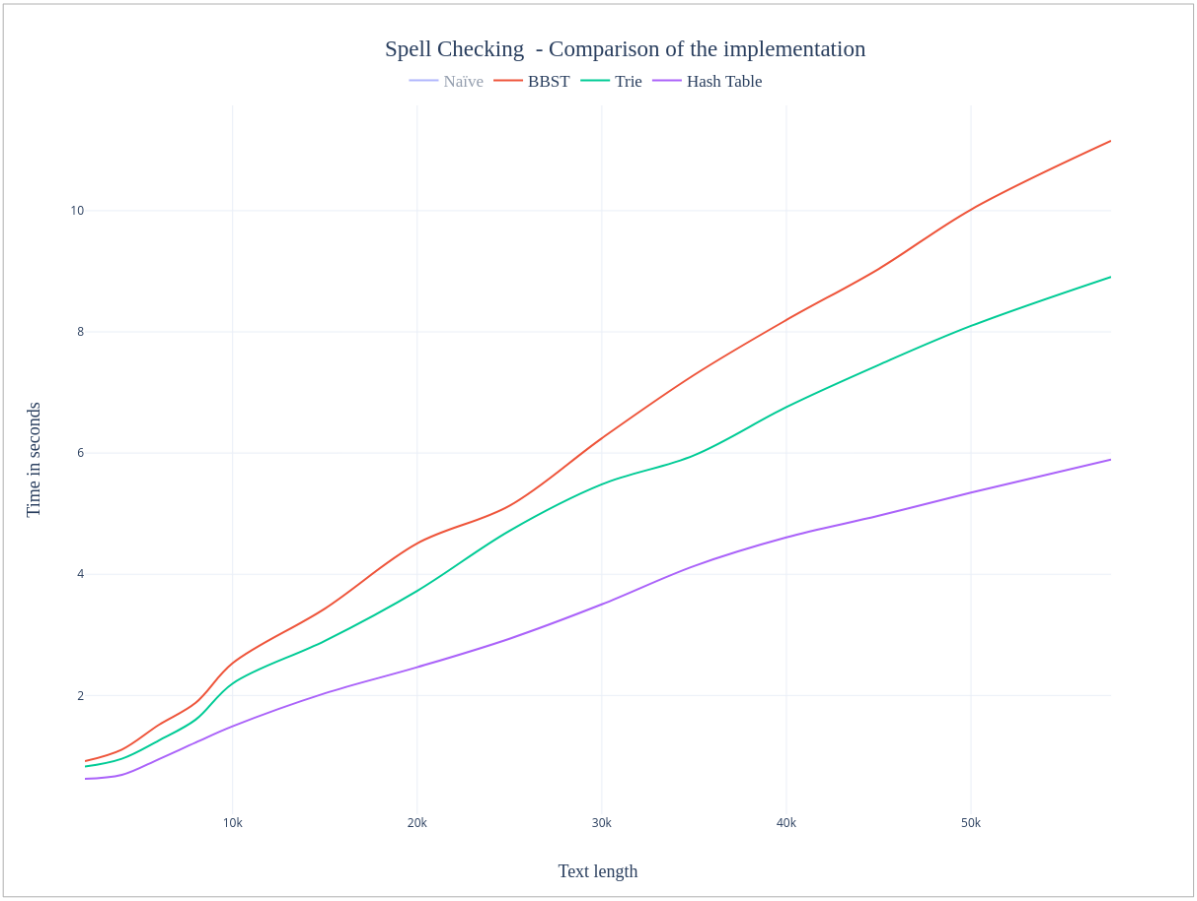

Figure 3: Comparison of the average time each implementation (except Naive) took to spell check an increasing text (2k – 57k).