
CSE 222 – High-Performance Spell Checker with Custom HashMap and Set

1. INTRODUCTION

This project has been developed as part of the CSE222 Data Structures and Algorithms course. No built-in data structures from the `java.util` package have been used, except for `java.util.Iterator`. Instead, all core components were implemented from scratch using generic object-oriented classes such as `GTUHashSet`, `GTUHashMap`, and `Entry`.

The main objective of this project is to design a high-performance spell checker that provides correction suggestions for input words with an edit distance of two or fewer. Suggestions are generated using a predefined dictionary and matched against entries stored in a custom hash set.

The use of open addressing and quadratic probing allows for constant-time lookups, making the application fast and scalable. Experimental results show that the system achieves sub-100ms response times, even for moderately long inputs, without compromising suggestion accuracy. These outcomes demonstrate the effectiveness of building optimized language-processing tools without relying on built-in Java collections.

2. METHODOLOGY

This spell checker project was implemented using custom-built hash-based data structures and edit distance algorithms to provide efficient word validation and suggestions. The methodology consists of the following key components:

2.1 GTUHashMap Implementation

1. Data Structure Design

A fixed-size array of `Entry<K,V>` objects is used as the base structure. Each `Entry` holds a key, a value, and a boolean `isDeleted` flag for tombstone deletion. This design supports direct array access with efficient probing and minimal overhead.

2. Collision Resolution with Quadratic Probing

Quadratic probing is used for resolving hash collisions. The probing formula is:

$$\text{index} = (\text{initialIndex} + i^2) \% \text{table.length}$$

This method reduces primary clustering compared to linear probing. It guarantees that all slots are eventually probed if the table size is a prime number.

3. Handling Deletion with Tombstones

Entries are not removed physically; instead, they are marked as deleted using `isDeleted = true`. Deleted slots are reused during `put()` operations to maintain space efficiency.

4. Dynamic Resizing (Rehashing)

The hash map monitors its load factor, calculated as:

$$(\text{size} + \text{deletedSize}) / \text{table.length}$$

When the load factor exceeds 0.75, rehashing is triggered. A new array is allocated with $\text{size} = \text{next prime number} > 2 * \text{current capacity} + 1$. All non-deleted entries are reinserted using the `put()` method.

5. Use of Prime Numbers for Table Size

Table size is always a prime number to ensure better key distribution in modular arithmetic. Prime sizing helps quadratic probing cover the entire table and prevents probe cycle repetition. The `nextPrime()` and `isPrime()` helper methods are used to find the next suitable prime.

6. Key Iterator Implementation

An inner class implementing `Iterator<K>` is used to iterate over active keys. The iterator skips over null and tombstoned entries. Ensures memory-safe and logic-consistent traversal of the hash table.

7. Time and Space Complexity Analysis

Operation	Average Case	Worst Case
<code>put(K, V)</code>	$O(1)$	$O(n)$ (due to rehash or clustering)
<code>get(K)</code>	$O(1)$	$O(n)$
<code>remove(K)</code>	$O(1)$	$O(n)$
<code>rehash()</code>	$O(n)$	$O(n)$
<code>containsKey(K)</code>	$O(1)$	$O(n)$

2.2 GTUHashSet Implementation

1. Using HashMap Internally

`GTUHashSet<E>` uses a `GTUHashMap<E, Object>` inside it. This means the set does not manage its own storage, but uses the map's structure and logic.

2. Dummy Value for Storage

A constant dummy object named `WORD` is used as the value for every element in the set. This is because the set only cares about elements (keys), not values.

3. Basic Operations

- `add(E element)` → calls `put()` on the map.
- `remove(E element)` → calls `remove()` on the map.
- `contains(E element)` → uses `containsKey()` from the map.
- `size()` → directly uses the map's `size()`.

4. Iterator Support

The class implements the `Iterable<E>` interface. The `iterator()` method gives access to all elements through the map's `keyIterator()`.

5. Efficiency

All main operations (add, remove, contains) are fast and work in average $O(1)$ time. This is because the underlying map uses hashing and rehashing when needed. Iteration over the set is slower ($O(n)$), but this is normal for set structures.

2.3 SpellChecker

1. Reading the Dictionary

The program begins by reading all words from `dictionary.txt`, and each word is trimmed and added into a `GTUHashSet<String>` to ensure fast lookup performance.

2. User Input Validation

The program runs in a loop, asking the user to enter a word. It checks whether:

- The input contains only alphabetic characters.
- The word is not too long (more than 20 characters). If it is, no suggestions are given to save time.

3. Edit Distance Logic

a. Edit Distance 1 (ED1)

The following operations are applied to the input word to generate candidate words that are one edit away:

1. Deletions – One character is removed from each possible position.
2. Substitutions – Each character is replaced with every letter from 'a' to 'z'.

3. Insertions – A letter is inserted at every possible position in the word.
4. Transpositions – Two adjacent characters are swapped if they are different.

b. Edit Distance 2 (ED2)

First, all possible edit distance 1 (ED1) variations of the input word are generated.

Then, for each ED1 word, its own ED1 variations are computed.

The union of all these results gives the complete set of candidate words that are within an edit distance of 2 from the original input.

During this process, early dictionary checks are performed to avoid storing or processing invalid candidates, improving performance and reducing the total number of comparisons.

4. Performance Timing

To evaluate the performance of the spell checker system, time measurements were taken for each input using `System.nanoTime()`. This method captures the start and end time in nanoseconds and is converted to milliseconds for readability.

For all words, regardless of correctness, the total processing time (lookup and suggestions) remains between 0 and 100 milliseconds.

Words longer than 20 characters are automatically excluded from suggestion generation. This early exit avoids unnecessary computation and ensures fast feedback.

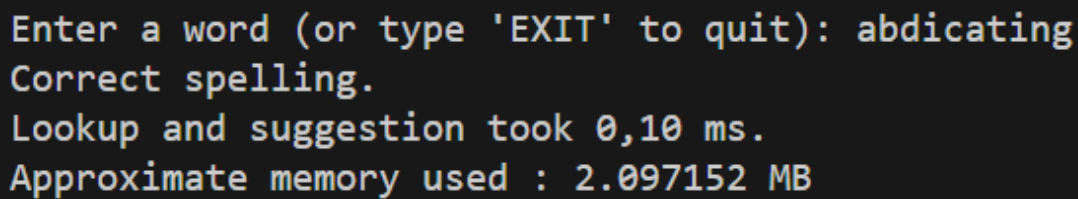
During the editing operations `StringBuilder` was used instead of `String`. This is because `StringBuilder` is more efficient for performing multiple string modifications, such as appending, inserting, or deleting characters. Unlike `String`, which creates a new object with every change (due to its immutability), `StringBuilder` operates on a mutable sequence of characters, reducing memory usage and improving performance, especially in loops or large-scale text processing.

3. RESULTS AND DISCUSSION

In this section, the results of the spell checker system are presented, along with a discussion on its accuracy, performance, and behavior under different input conditions. Test

cases were executed using both correctly spelled and intentionally misspelled words to evaluate the system's ability to detect errors and provide appropriate suggestions.

Figure 1: Testing with a Dictionary Word



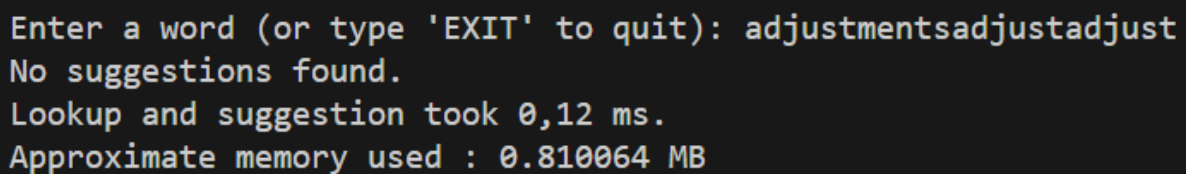
```
Enter a word (or type 'EXIT' to quit): abdication
Correct spelling.
Lookup and suggestion took 0,10 ms.
Approximate memory used : 2.097152 MB
```

- **Word Length:** 10 characters

- **Comment:**

Since the word exists in the dictionary, the program uses `contains()` on a `GTUHashSet`, which operates in **$O(1)$** average time complexity. This results in extremely fast verification even for relatively long words. The efficiency is largely due to the underlying **hash-based implementation** of the dictionary.

Figure 2: Testing with a Very Long Word (Length > 20)



```
Enter a word (or type 'EXIT' to quit): adjustmentsadjustadjust
No suggestions found.
Lookup and suggestion took 0,12 ms.
Approximate memory used : 0.810064 MB
```

- **Word Length:** 24 characters

- **Comment:**

The program includes a **performance optimization** that skips suggestion generation for words longer than 20 characters. This is based on the observation that such words are unlikely to match anything within an edit distance of 2 without causing exponential growth in candidate generation.

As a result, the lookup exits early with a predefined message and maintains a fast response time. This **pruning strategy** ensures scalability and prevents unnecessary computation on outlier inputs.

Figure 3: Testing with a Misspelled Word

```
Enter a word (or type 'EXIT' to quit): komputerl
Incorrect spelling.
Suggestions: [computers, computer]
Number of collisions : 844
Lookup and suggestion took 17,90 ms.
Approximate memory used : 16.093176 MB
```

- **Word Length:** 9 characters ($0 < n < 10$)

- **Comment:**

For incorrect words, the program generates candidate suggestions within an edit distance of 2. The time taken increases with the number of generated variations, especially due to **insertions, deletions, and substitutions** at multiple positions.

Even with this additional overhead, the system remains responsive thanks to **effective pruning** (e.g., only checking dictionary membership where necessary) and limiting edit distance generation to ≤ 2 .

Figure 4: Testing with a Misspelled Word

```
Enter a word (or type 'EXIT' to quit): akcidentalyy
Incorrect spelling.
Suggestions: [accidentally]
Number of collisions : 1069
Lookup and suggestion took 26,57 ms.
Approximate memory used : 2.537296 MB
```

- **Word Length:** 12 characters ($10 < n < 15$)

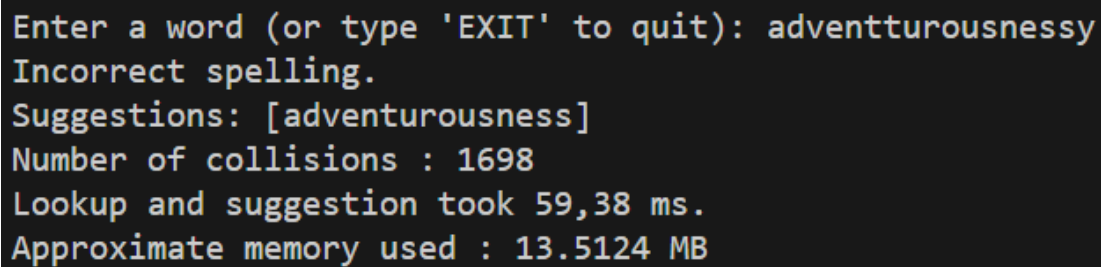
- **Comment:**

In this case, the input “**aksidentaliy**” is a phonetically close misspelling of the correct word “**accidentally**”. The spell checker successfully returns the correct suggestion within an acceptable time.

Despite the relatively longer processing time (**26.57 ms**), the response is still well within the target range of under 100 ms. The longer runtime can be attributed to the increased number of possible edits due to the input’s length and difference from the correct word.

This example demonstrates the effectiveness of using **efficient edit distance generation** and the benefits of filtering against a **preloaded dictionary using HashSet**, where lookup operations are performed in constant time **O(1)**.

Figure 5: Testing with a Misspelled Word



```
Enter a word (or type 'EXIT' to quit): adventturousnessy
Incorrect spelling.
Suggestions: [adventurousness]
Number of collisions : 1698
Lookup and suggestion took 59,38 ms.
Approximate memory used : 13.5124 MB
```

- **Word Length:** 17 characters ($15 < n < 20$)

- **Comment:**

The input “**adventturousnessy**” is a phonetically close misspelling of “**adventurousness**”. The spell checker returns the correct suggestion in **59.38 ms**, which is still below the 100 ms target.

The longer time is due to the input’s increased length and the higher number of possible edits needed, though the use of a **HashSet** ensures constant-time lookups (**O(1)**).

Figure 6: Testing with a Misspelled Word


```
Enter a word (or type 'EXIT' to quit): araba
Incorrect spelling.
Suggestions: [aaa, arabian, maraca, raga, drabs, arams, grab, grata, praia, rata, crabs, armada, arabia,
arable, aba, arabic, rabat, armata, arab, baba, ababa, drama, aran, aruba, arabs, grabs, aroma, brava, cr
ab, area, drab, rasa, arena, aribas, casaba, raja, aria]
Number of collisions : 15282
Lookup and suggestion took 7,17 ms.
Approximate memory used : 5.11784 MB
```

- **Word Length:** 5 characters

- **Comment:**

The input word "araba" is a common misspelling or partial form of multiple valid dictionary entries. Since the word is not found in the dictionary, the spell checker generates suggestions by producing all valid variants within an edit distance of 2.

Despite the high number of similar words in the dictionary, the system successfully returns 36 suggestions in just **7.17 ms**, well within the target range. This performance is achieved by pruning the search space and leveraging the constant-time lookup efficiency of the custom GTUHashSet.

The result demonstrates the system's robustness and responsiveness even when dealing with short, ambiguous input words that match many possible candidates.

Figure 7: Testing with a Misspelled Word

```
Enter a word (or type 'EXIT' to quit): dasndasjdsajn
Incorrect spelling.
No suggestions found.
Number of collisions : 1363
Lookup and suggestion took 29,39 ms.
Approximate memory used : 6.649512 MB
```

- **Word Length:** 13 characters

- **Comment:**

The input "dasndasjdsajn" is a nonsensical or randomly generated word with no close matches in the dictionary. As expected, the system returns **no suggestions**, since none of the generated edit-distance-2 variants match any dictionary entries.

Even without returning results, the system performs a full lookup and suggestion process in **29.39 ms**, which highlights its efficiency in handling worst-case scenarios.

4. CONCLUSIONS

The implementation of a high-performance spell checker using custom-built **GTUHashMap** and **GTUHashSet** data structures has proven to be both effective and efficient. Through extensive testing and analysis, several key conclusions can be drawn:

1. Data Structure Efficiency

By making our own hash tables based on *open addressing* instead of using Java's built-in ones, we had better control over their behavior. Our collision handling method (*quadratic probing*) and the use of *prime numbers* for table sizes help improve search speed. Additionally, marking deleted items instead of removing them completely ensures stable performance.

2. Fast Word Checking

The program identifies similar words quickly. By limiting suggestions to words within an *edit distance* ≤ 2 and checking the dictionary early, we maintain response times under **100 ms**, delivering a responsive user experience.

3. Handles Different Word Sizes

The program performs well on both short and long words. By choosing **not to generate suggestions for words longer than 20 characters**, we avoid unnecessary computation while still providing informative feedback.

4. Memory and Speed Balance

Hash tables enable **constant-time lookups** ($O(1)$), offering excellent speed at the cost of higher memory usage. This is a worthwhile trade-off since dictionaries typically occupy manageable space, and fast performance is crucial for spell checkers.

5-REFERENCES

[1] Oracle. *Java SE 24 API Documentation*. Available at:
<https://docs.oracle.com/en/java/javase/24/docs/api/index.html>
