Spell Checker Benchmark Report

By Ivan Malishevskyi & Oleksii Savelich
All the code can be found here

Objective:

The goal of the task was to implement a simple spell checker that decides whether each individual word in a text file is spelled correctly, using various data structures. The evaluation was twofold:

1. Implement and test the spell-checking functionality using different approaches.
2. Compare running times for **dictionary building** and **spell checking** on a large piece of text.

Implemented Approaches:

Four spell-checking strategies were implemented, each relying on a different underlying data structure:

Method	**Description**
LinearCheck	Naive linear search over a list of dictionary words
BstCheck	Uses a balanced binary search tree (e.g., `SortedSet`)
HashSetCheck	Employs a hash-based set for constant-time lookups
TrieCheck	Implements a trie (prefix tree) to match words by traversing character paths

---

Experimental Setup

- **Dictionary Source:** A standard English word list (~100,000 words).

- **Text Sources:** Three input texts different in size, even though task suggested using "large piece of text", it wasn't specified what exactly word "large" means, so I thought I might use a couple of relatively large texts just in case.
    1. `small.txt` (~10 KB)*
    2. `medium.txt` (~100 KB)*
    3. `AliceInWonderland.txt` (~150 KB)

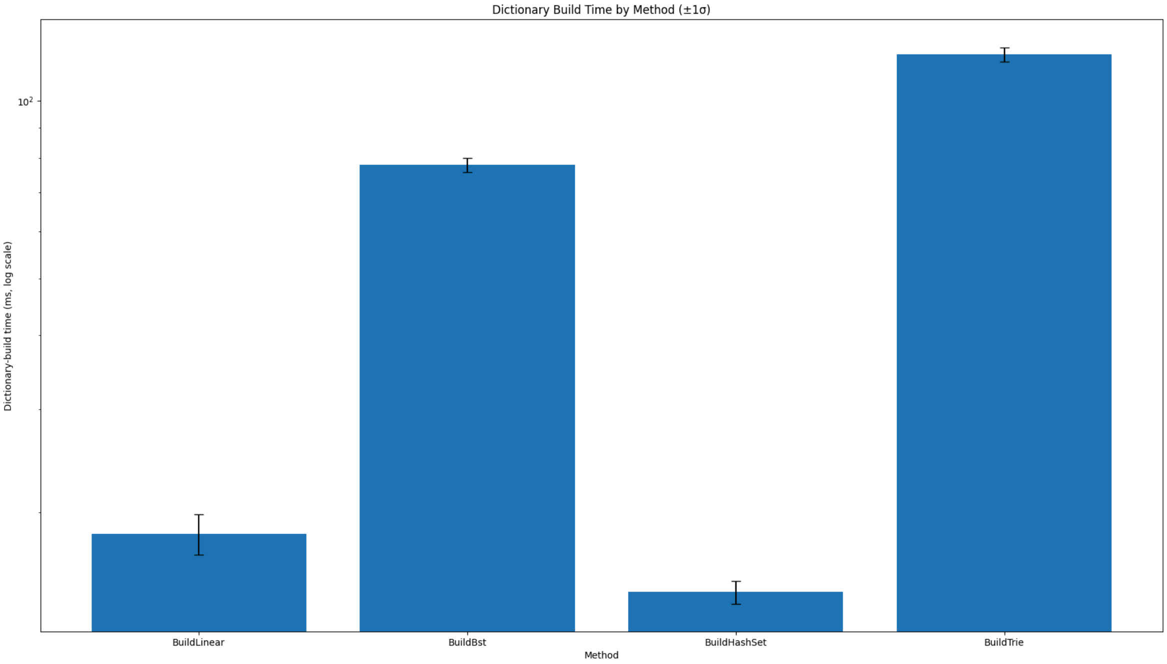
*marked files were made through modifying AliceInWonderland.txt by striping different amount of symbols at the end of the file

Each spell checker was benchmarked for:

1. **Dictionary build time** (structure construction from word list)
2. **Spell-check time** (checking a text file for misspelled words)

Visual benchmark results followed by observations

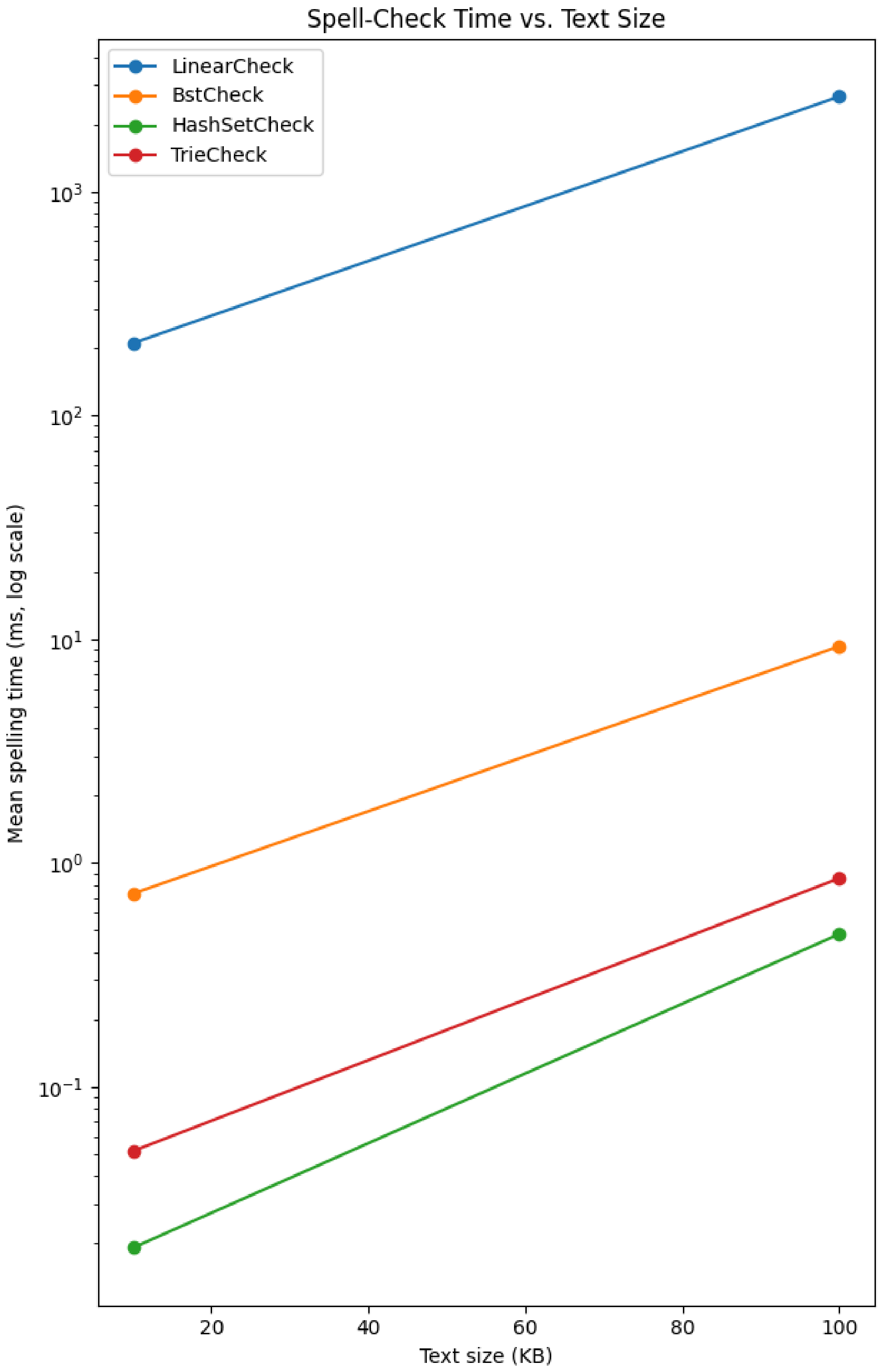
Dictionary build time



Observation:
HashSet is the fastest structure to build, followed closely by Linear. The Trie takes the

longest to build, likely due to its per-character node structure. BST sits in the middle but is significantly slower than HashSet.

Spellcheck time



Spell-Check Time vs Text Size

Method	10 KB (ms)	100 KB (ms)	150 KB (ms)	Time Complexity
LinearCheck	460.79	3563.11	5389.57	**O(N × D)** — linear scan through the dictionary
BstCheck	15.57	134.85	203.71	**O(N × log D)** — binary search in balanced BST
HashSetCheck	0.72	6.39	9.73	**O(N)** — average-case constant lookup
TrieCheck	1.64	12.49	18.83	**O(N × L)** — prefix tree traversal per word

Legend:

> **N** = number of words in the input text
> **D** = number of dictionary words
> **L** = average word length

Summary of Results

- **LinearCheck** is the slowest and scales poorly — shouldn't be used for large inputs.
- **BstCheck** improves performance with logarithmic lookups but still grows with input size.
- **HashSetCheck** is the fastest overall — great build time and constant-time lookups.
- **TrieCheck** performs nearly as well and is useful for prefix-based operations.

Conclusion

This project clearly demonstrates how the choice of data structure directly impacts both the speed of building a dictionary and the efficiency of spell checking. The differences were especially noticeable when processing larger texts. But on a smaller scale texts it could be clearly seen too.

In summary, **hash-based and trie-based approaches should be preferred** for building high-performance spell checkers. They scale well with large texts and provide fast, predictable behavior.

> **ⓘ Tools used:** ›
>
> - Obsidian *, to generate this fancy report and export it into pdf*
> - BenchmarkDotNet *, to measure closest to raw performance of the algorithms*
> - C# & .NET *, the language of choice and environment to implement algorithms*
> - Project Gutenberg library *, the input text was downloaded from there*
> - Matplotlib *, library used for visualization output of benchmarks*

> **ⓘ Machine specs** ›
>
> Windows 11 (10.0.22631.5189/23H2/2023Update/SunValley3)
> 12th Gen Intel Core i5-1240P, 1 CPU, 16 logical and 12 physical cores
> .NET SDK 8.0.408
> [Host] : .NET 8.0.15 (8.0.1525.16413), X64 RyuJIT AVX2
> DefaultJob : .NET 8.0.15 (8.0.1525.16413), X64 RyuJIT AVX2

Easter egg

I'm sure that nobody's gonna read my report till the end but in case you will I just wanna let you know that I love pizza ❤️