Word Frequency Analysis

Muneeb Azfar Nafees

April 2025

Contents

1 Abstract

I explored how the choice of data structure affects large-scale text analysis by counting word frequencies in two different corpora: Reddit comments from 2015 and Shakespeare's complete works. To do this, I implemented both a Binary Search Tree (BST) map and a Hash Table map, used each to build frequency tables, and measured build times, structural depth, and top-word lists. I analyzed core computer science data structures, comparing the $O(\log n)$ performance of binary search tree inserts to the $O(1)$ runtime of hash-based inserts to understand their trade-offs. My key findings show that hash maps are up to five times faster than BSTs on large corpora due to their shallower bucket depths and that the most frequent words differ dramatically between modern social-media text and Early Modern English.

2 Results

2.1 Analysis 1: Top 10 Words

2.1.1 Experimental Setup

I used the `WordCounter` class backed by the `BSTMap` implementation to process two corpora: the 2015 Reddit comments dataset (`CLEANED_reddit_comments_2015.txt`) and the complete works of Shakespeare (`CLEANED_shakespeare.txt`). For each corpus, I called `readWords()` to tokenize the text into words, then `buildMap()` to tally word frequencies. Finally, I extracted the full entry set, sorted it in descending order by count, and selected the top ten entries.

2.1.2 Metrics

My key metric for this analysis was *word count*, the total number of occurrences of each word in the corpus. Table 1 presents the ten most frequent words for each dataset.

Rank	Reddit Comments	Count	Shakespeare	Count
1	like	180,784	thou	5,485
2	dont	137,848	thy	4,032
3	would	130,045	shall	3,591
4	get	124,499	thee	3,178
5	one	123,123	lord	3,059
6	people	111,374	king	2,861
7	think	98,193	good	2,812
8	really	85,492	sir	2,754
9	time	82,538	come	2,507
10	good	80,540	well	2,462

Table 1: Top 10 most frequent words in the 2015 Reddit comments vs. Shakespeare's complete works.

2.1.3 Observations

- Both lists are dominated by high-frequency "function words" or common terms of address: in Reddit, words like "like," "dont," and "would"; in Shakespeare, archaic pronouns and titles such as "thou," "thy," and "lord."
- Only one word, "good," appears in both top-10 lists, confirming that modern social-media language and Early Modern English share very little in their highest-frequency vocabulary.
- The magnitude difference is striking: Reddit's top word appears over 180,000 times, whereas Shakespeare's most common word ("thou") appears fewer than 6,000 times.

This reflects not only the size of the datasets but also the relative redundancy in social-media posts.

2.1.4 Interpretation

These results align with my expectations. In a large, conversational corpus like Reddit comments, everyday verbs and adverbs ("like," "get," "really," "time") dominate, often serving as colloquial fillers. Shakespeare's corpus, by contrast, features formal pronouns and titles befitting its genre and era. The presence of "good" in both lists hints at its enduring role as both a common adjective and a normative term of praise across centuries.

2.2 Analysis 2: Build Time Comparison

2.2.1 Experimental Setup

I measured the time taken by the `buildMap()` method on two datasets, 2015 Reddit comments and Shakespeare's complete works, using two data structures (BSTMap and HashMap). For each combination, I ran the map construction 10 times, cleared the map between runs, and recorded the elapsed time via `System.currentTimeMillis()`.

2.2.2 Metrics

The primary metric is the *average build time* (in milliseconds) over 10 runs for each dataset and data structure.

2.2.3 Results

Data Structure	Reddit (ms)	Shakespeare (ms)
BSTMap	7190.8	147.5
HashMap	1272.8	26.3

Table 2: Average time (ms) to build the word-frequency map over 10 iterations.

2.2.4 Discussion

The HashMap implementation outperformed the BSTMap by a substantial margin on both datasets. On the large Reddit corpus, HashMap was over five times faster, reflecting its average O(1) insert runtime versus BSTMap's O($\log n$). Even on the smaller Shakespeare dataset, HashMap built the map in under 30ms on average, compared to nearly 150ms for BSTMap. These results confirm that hash-based lookups scale more efficiently with increasing data volume.

2.3 Analysis 3: Max Depth Analysis

2.3.1 Experimental Setup

I modified the main method to call `maxDepth()` on each `WordCounter` immediately after `buildMap()` finished for both the Reddit and Shakespeare datasets, using both BSTMap and HashMap implementations.

2.3.2 Metrics

The key metric is the *maximum depth* of the data structure after all words are inserted:

- For BSTMap: the length of the longest root-to-leaf path.
- For HashMap: the size of the largest bucket (chain length).

2.3.3 Results

Data Structure	**Reddit maxDepth**	**Shakespeare maxDepth**
BSTMap	38	35
HashMap	6	5

Table 3: Maximum depth of BSTMap vs. HashMap after building the word-frequency map.

2.3.4 Discussion

These depth measurements help explain the timing results from Analysis 2. The BSTMap develops a maximum depth near 40, meaning each lookup or insert may traverse up to 38–35 nodes in the worst case. In contrast, the HashMap's deepest bucket contains only 5–6 entries, so it rarely traverses more than half a dozen pointers for a single check. This difference in worst-case path length directly contributes to HashMap's superior build times on both large and small corpora.

3 Extensions

3.1 BSTMap: Balance-Factor Analysis

3.1.1 What I did and why

To quantify how unbalanced my plain BSTMap becomes, I implemented a new method, `numberOfUnbalancedKeys()`, which traverses every node, computes the height of its left and right subtrees, and counts how many nodes have an absolute balance factor greater than 1. I first validated this on a degenerate tree (inserting 1,2,3,4,5,6 in order), where four of the six nodes should be unbalanced.

3.1.2 Outcome

After the validation, I inserted random integers in batches, 10, 20, 30, and 100 extra keys, in the same BST to see the effects on number of unbalanced keys. Table 4 shows the resulting tree size versus the number of unbalanced nodes.

Tree Size	Unbalanced Nodes
10	3
30	13
60	15
158	55

Table 4: BSTMap size versus count of unbalanced nodes (absolute balance factor > 1).

Without any self-balancing, the BST grows increasingly skewed as more keys are added. By defining "unbalanced" as nodes whose balance factor exceeds 1 in absolute value, we see that a simple metric, counting those nodes, captures how far the tree deviates from ideal balance. This analysis highlights the need for a self-balancing strategy: as the tree's size increases, the proportion of unbalanced nodes quickly rises, degrading search and insertion performance.

3.1.3 How to replicate

Include the `maxDepth(Node)` method and `unbalancedKeys(Node)` recursion helper in your `BSTMap` class. After building the tree with `put()` calls, invoke `numberOfUnbalancedKeys()` to retrieve the count of unbalanced nodes.

3.2 HashMap: Custom Polynomial Rolling Hash

3.2.1 What I did and why

I created a new class `HashMapExt` by copying the original `HashMap` and replacing its `hash(K key)` method with a polynomial rolling hash. I then updated `WordCounter` to allow choosing between the standard `HashMap` and `HashMapExt`. Finally, I measured each

map's average build time and maximum bucket depth on both corpora to assess how the custom hash spreads keys.

3.2.2 Outcome

Implementation	Reddit Time (ms)	Shakespeare Time (ms)	Reddit Depth	Shakespeare Depth
HashMapExt (rolling)	1500.9	35.4	6	5
Built-in `HashMap`	1117.2	23.8	6	5

Table 5: Average build time and max bucket depth for custom vs. built-in hash functions.

Both implementations exhibit identical bucket depths, suggesting that the polynomial hash did not significantly alter collision patterns on these corpora. The custom hash performed slightly worse in build time, likely due to the extra multiplications per character. Overall, the built-in `hashCode()` remains preferable for performance in this context.

3.2.3 How to replicate

1. Copy your original `HashMap` class to a new `HashMapExt`.
2. Replace `hash(K key)` with:

```
public int hash(K key){
    int h = 0;
    int a = 31;

    for(int i = 0; i < key.toString().length(); i++){
        h = (a * h + key.toString().charAt(i)) % capacity();
    }
    return h;
}
```

3. Add a `"HashMapExt"` option in your `WordCounter` constructor.
4. Run the main analysis on both Reddit and Shakespeare datasets with each implementation, recording `buildMap()` times and calling `maxDepth()`.

3.3 Map Creation using ArrayList

3.3.1 What I did and why

I implemented `ArrayListMap`, which keeps all `KeyValuePair` objects in a single `ArrayList` and uses a linear scan for each `put`, `get`, and `remove`. I then updated `WordCounterExt` to allow choosing `ArrayListMap` and measured map-building times for BSTMap, HashMap, and ArrayListMap on both corpora.

3.3.2 Results

Data Structure	Reddit (ms)	Shakespeare (ms)
BSTMap	6770	133
HashMap	1203	24
ArrayListMap	339558	5410

Table 6: Average time to build the word-frequency map over 10 runs.

The ArrayListMap is orders of magnitude slower than both BSTMap and HashMap, taking over 300 000 ms on the Reddit corpus. This result matches the theoretical $O(m \cdot k)$ runtime, compared to $O(m \log k)$ for BSTMap and $O(m)$ for HashMap, where m is the number of total words and k is the number of unique words.

3.3.3 How to replicate

1. Add the `ArrayListMap` class implementing `MapSet` by maintaining an `ArrayList<KeyValuePair<K,V>>` and scanning it for each operation.
2. Modify `WordCounterExt` to accept the option `"ArrayList"` and instantiate `ArrayListMap` when chosen.
3. Run the existing main harness on both datasets with each map type, record `buildMap()` times, and report the averages.

4 Acknowledgments

1. String hashing using Polynomial rolling hash function - GeeksForGeeks
2. Function Word - Wikipedia
3. Peers: Alize Warraich, Fahad Chaudhry, Ibraheem Waheed & Muhammad Ibraheem