Spell Checker And Suggest

August 26, 2024

Shravan Jindal (2022CSB1124) ,
Siddharth (2022CSB1125) ,
Siddharth Verma (2022CSB1126)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Soumya Sarkar

Summary: The C-based spell checker project employs a combination of advanced algorithms and data structures for efficient spell checking. It utilizes Trie data structure for dictionary storage and retrieval. To enhance performance, it incorporates a Bloom Filter to quickly eliminate non-existent words. The Levenshtein edit distance algorithm and Jaro Winkler distance algorithm help suggest corrections by measuring string similarity. An LRU Cache is implemented for optimizing the speed spell checking for least recently checked words. Additionally, multithreading is used to parallelize and expedite the spell checking process, resulting in a robust, high-speed spell checker with accurate suggestions.
The culmination of these elements results in a powerful and efficient spell checker that not only corrects misspelled words but also provides accurate suggestions in a timely manner. The spell checker's performance has been rigorously tested, demonstrating its effectiveness in a variety of scenarios.
This project represents a holistic and robust solution, combining several data structures and algorithms to create a high-performance spell checking system, laying a strong foundation for future enhancements and applications.

---

1. Introduction

Within the realm of computational linguistics, accurate text processing is paramount. This project endeavors to present a comprehensive and efficient spell checker developed in the C programming language. In today's digitally reliant environment, ensuring precise spelling and grammar is fundamental to effective communication. This project amalgamates a range of sophisticated algorithms and data structures—such as Trie, Bloom Filter, Levenshtein edit distance, Jaro Winkler distance, LRU Cache, and multithreading—to create a robust spell checking system. The objective of this endeavor is to enhance the accuracy and speed of error detection and correction in textual content, catering to the demands of various applications and software systems. All the used references [1] [5] [4] [3] [6] [8] [2] [7] can be found at the last of this report.

2. Data Structures

This section gives a concise and detailed working of the various data structures used in the project.

2.1. Tries

Tries, also known as prefix trees, are tree-like data structures primarily used for efficient retrieval and storage of strings or words. Each node in the trie represents a character, with the root node typically denoting an empty string. The structure's design allows for rapid search and insertion operations, making it ideal for dictionary-related applications, such as spell checkers. Tries excel in minimizing the time complexity of string-related operations, enabling quick prefix-based searches and dictionary implementations. By leveraging the shared

prefixes among words, tries significantly reduce the time and memory requirements for tasks like autocomplete, spell checking, and searching for words in large collections. With their structured hierarchy and efficient traversal, tries offer a powerful solution for organizing, retrieving, and manipulating textual data in an optimized and scalable manner.

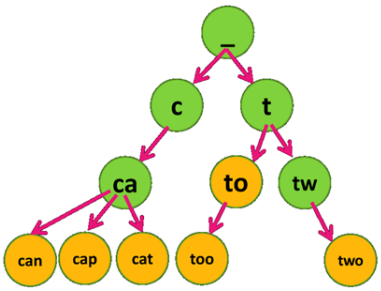

Figure 1: Trie storing words

Basic Tries Operations

	Time Complexity	Space Complexity
Insertion	O(m)	O(m)
Search	O(m)	O(1)

Table 1: 'm' being the length of word

2.2. Bloom Filter

A Bloom filter is a probabilistic data structure designed for efficient membership query operations, particularly for large datasets, where memory optimization is crucial. It works by using a bit array and a set of hash functions to store and quickly query the existence of an element within a collection. When adding an item to the Bloom filter, it undergoes multiple hash functions, setting the corresponding bit positions in the array. During a lookup, if all the bits accessed through the hash functions are set, the element is likely present in the dataset, but false positives can occur due to potential hash collisions. Bloom filters are highly space-efficient as they require minimal memory compared to other data structures. Their main advantage lies in their speed and memory savings, making them well-suited for applications that can tolerate a small probability of false positives, such as spell checkers and web caching systems, where quick queries for potential element existence are vital.

Formulas for bloom filter

if n=total words to be added in boom filter
if p=false positive rate
Filter Size (m) can be determined using:

$$m = \frac{-n \ln(p)}{\ln(2)^2}$$

Number of Hash Functions (k) can be determined using:

$$k = \frac{m}{n} \ln(2)$$

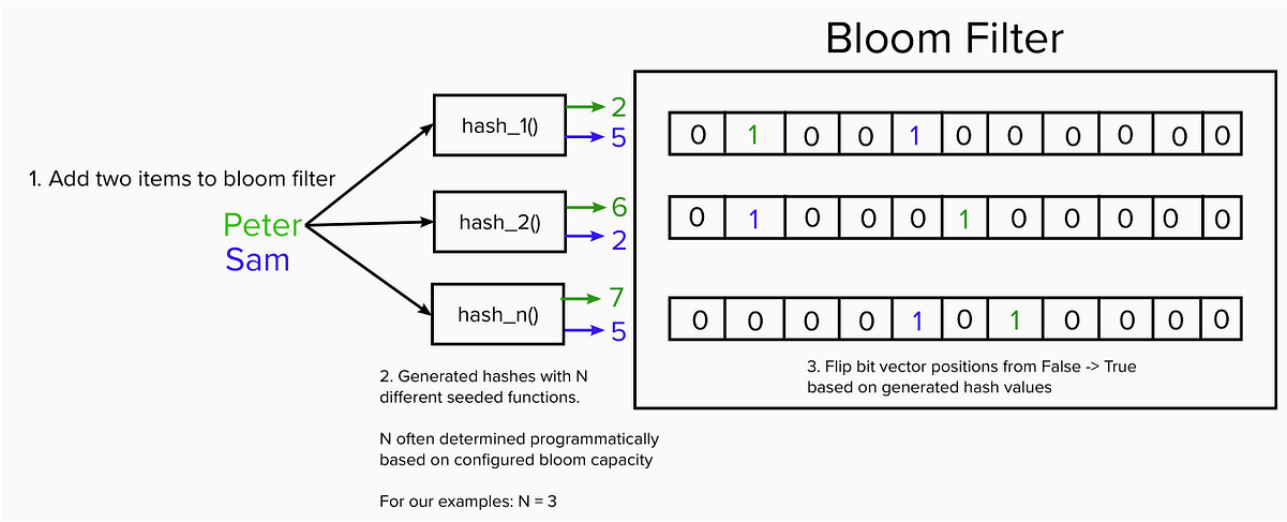

Figure 2: Bloom filter to store words

Basic Bloom Filter Operations

	Time Complexity	Space Complexity
Insertion	O(k)	O(1)
Search	O(k)	O(1)

Table 2: 'k' being number of hashes used

2.3. LRU Cache

An LRU (Least Recently Used) cache is a memory structure that efficiently manages data by retaining the most recently accessed items while discarding the least recently used elements when the cache reaches its capacity. This policy ensures that the most relevant data remains readily accessible, enhancing overall performance. The LRU cache operates on the principle that recently accessed items are more likely to be accessed again in the near future. It typically employs a data structure, such as a doubly linked list or a hashmap, to facilitate quick insertion, deletion, and retrieval operations. When an item is accessed, it moves to the front of the cache, ensuring its retention. Conversely, when the cache is full and a new item needs to be added, the least recently used item, positioned at the end of the cache, is evicted. LRU caches are widely used in various computing systems, including databases, web browsers, and operating systems, to optimize performance by strategically managing the most frequently accessed data and minimizing access times to enhance overall efficiency.

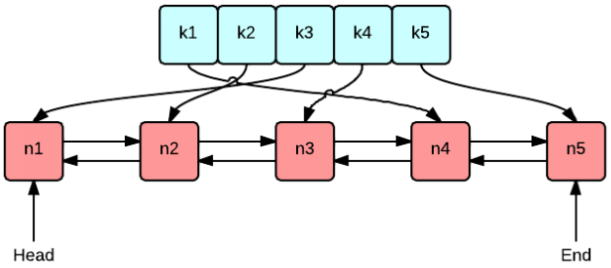

Figure 3: LRU Cache with Hash Map and Queue

Basic LRU Cache Operations

	Time Complexity	Space Complexity
Put	O(1)	O(1)
Get	O(1)	O(1)

Table 3: Note:- This is average case complexity. In worst case it might be linear

3. Algorithms

This section gives the pseudo codes of important algorithms used in the project along with their working and complexities(space and time).

3.1. DJB2 Hash

The djb2 hash, created by Daniel J. Bernstein, is a simple and effective non-cryptographic hash function. It converts strings into a hash value (an unsigned 32-bit integer). The algorithm traverses each character of the input string, incorporating it into the hash value using basic bitwise operations and addition.

3.1.1 Time Complexity

The time complexity of the djb2 hash function is considered to be linear or O(n), where 'n' represents the length of the input string being hashed.
The function processes each character of the string one by one, performing simple operations (such as additions and bitwise shifts) on each character. Consequently, the time taken for the hash calculation increases linearly with the length of the input string.

3.1.2 Space Complexity

The space complexity of the djb2 hash function is considered to be constant or O(1). This is because the algorithm uses only a fixed and finite amount of memory regardless of the input size. The memory usage does not grow with the input size; it remains constant as it works on a character-by-character basis without using additional memory allocations or increasing space requirements.

Algorithm 1 djb2(const char* string)

```
hash = 5381; // Initial value of hash
int c; // Integer to store each character of the string
while (c = *string++) do
    hash = (hash « 5) + hash + c; // DJB_2 hashing: hash * 33 + character
end while
return hash % FILTER_SIZE; // Modulo operation to fit within the range of the filter size
```

3.2. Jenkins Hash

The Jenkins hash function, or more specifically, the Jenkins One-at-a-Time hash, is a non-cryptographic hash function created by Bob Jenkins. It is designed for hash table implementations and other applications that require hash values. This hash function processes data one byte at a time and accumulates a hash code by mixing the bits in a sequence of simple arithmetic and bitwise operations.
It's widely known for its simplicity, speed, and reasonably good avalanche effect, where even small changes in the input data lead to significant changes in the resulting hash value, thereby contributing to its use in hash table implementations.

3.2.1 Time Complexity

The Jenkins hash function, especially the "One-at-a-Time" (OAT) variant, exhibits a linear time complexity, O(n), where 'n' represents the length of the input data being processed.
This hash function operates by iterating through each byte of the input data, performing simple arithmetic and bitwise operations for each byte. The time required for the computation increases linearly with the amount of input data, as each byte is processed in a sequential manner.
As a non-cryptographic hash function, its speed and simplicity make it well-suited for hash table implementations and applications requiring moderate to good distribution of hash values.

3.2.2 Space Complexity

The space complexity of the Jenkins hash function, particularly the "One-at-a-Time" (OAT) variant, is constant, denoted as O(1).
The space required for the hash function's operation remains consistent and does not increase with the size of the input data. Regardless of the input length, the algorithm's memory usage stays constant, representing a fixed amount of memory necessary for its operations.

Algorithm 2 jenkin(const char* str):

```
hash = 0 // Initialize the hash value
while *str is not NULL do
    hash+ = *str // Add the ASCII value of the character to the hash
    hash+ = (hash << 10)
    hash^=(hash >> 6)
    Move to the next character in the string
end while
hash+ = hash << 3 // Further hash transformation with bit shifting
hash^=hash >> 11
hash+ = hash << 15
return hash%(FILTER_SIZE) // Return the hash value modulo the filter size
```

3.3. Levenshtein Edit Distance

The Levenshtein distance algorithm, also known as the Edit Distance algorithm, is used to measure the difference between two strings. It determines the minimum number of single-character edits required to change one string into another. These single-character edits include insertion, deletion, or substitution.

3.3.1 Time Complexity

The time complexity of the Levenshtein algorithm is O(m * n), where m and n are the lengths of the input strings. As the algorithm fills in the (m+1) x (n+1) matrix, each cell's computation involves simple operations (comparisons and minimum calculations), resulting in a time complexity that scales quadratically with the length of the strings.

3.3.2 Space Complexity

The space complexity of the algorithm is also O(m * n) because it utilizes an (m+1) x (n+1) matrix to store the intermediate values during computation. Therefore, the space required is proportional to the product of the lengths of the input strings.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 4: Levenshtein algorithm

Algorithm 3 LevenshteinDistance(s, t):

```
n = length of string s
m = length of string t
if n equals 0, return m
if m equals 0, return n
declare dp[n+1][m+1] // Declare a 2D array for dynamic programming
for i from 0 to n do
    dp[i][0] = i // Initialize the first column from 0 to n
end for
for j from 0 to m do
    dp[0][j] = j // Initialize the first row from 0 to m
end for
for i from 1 to n do
    for j from 1 to m do
        if s[i-1] = t[j-1] then
            cost = 0
        else
            cost = 1
        end if
        dp[i][j] = smallest of (dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + cost)
    end for
end for
return dp[n][m] // Return the Levenshtein distance
```

3.4. Jaro Winkler Distance

The algorithm starts by computing the Jaro distance, which measures the similarity between two strings based on the number of matching characters and transpositions.
1. Jaro Distance: It calculates the number of matching characters and the number of transpositions (when two characters are not in the same order but close together) to determine a similarity ratio. It considers matches within a certain range or window size determined by the formula.
2. Jaro-Winkler Modification: The Jaro distance is then modified using the Jaro-Winkler formula, which boosts the score for the initial characters that match. It adds a scaling factor to the Jaro distance based on the prefix similarity of the strings.
The final result is a similarity score between 0 and 1, where 1 indicates a perfect match and 0 indicates no similarity.

3.4.1 Time Complexity

The time complexity of the Jaro-Winkler algorithm is O(m * n), where m and n are the lengths of the input strings. The algorithm iterates through both strings, performing character comparisons and transposition checks to compute the similarity score. However, it tends to perform better than the Levenshtein distance for longer strings due to its characteristic matching window size.

3.4.2 Space Complexity

The space complexity of the Jaro-Winkler algorithm is O(1), as it does not require additional memory proportional to the lengths of the input strings. The algorithm calculates the similarity score without using extra storage, resulting in constant space complexity.

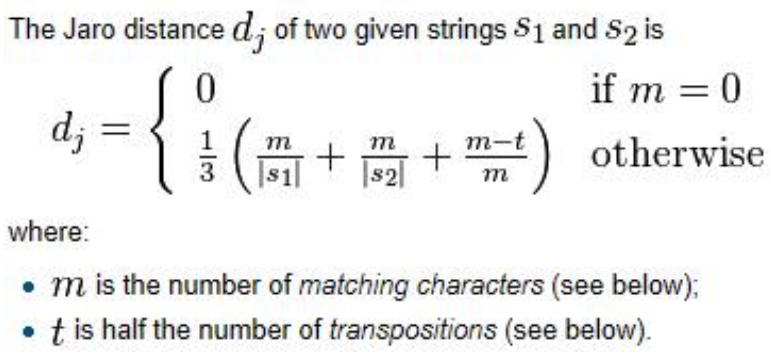

Figure 5: Jaro winkler algorithm

Algorithm 4 JaroWinklerDistance(s1, s2):

```
if s1 equals s2 then
    return 1.0 // If strings are equal, return 1.0 as similarity is 1.0
end if
len1 = length of s1 // Length of string s1
len2 = length of s2 // Length of string s2
max_dist = floor(greater(len1, len2) / 2) - 1 // Calculate the maximum allowed matching distance

match = 0 // Initialize match count
declare hash_s1[len1] and hash_s2[len2] // Arrays to mark matching characters
for i from 0 to len1-1 do
    for j from max(0, i - max_dist) to min(len2, i + max_dist + 1) do
        if s1[i] = s2[j] and hash_s2[j] = 0 then
            hash_s1[i] = 1 // Mark characters as matched
            hash_s2[j] = 1
            match = match + 1 // Increment match count
            break // Exit inner loop
        end if
    end for
end for
if match equals 0, return 0.0 // If no matches found, return 0.0 as similarity is 0.0
t = 0 // Initialize transposition count
point = 0 // Pointer for s2
for i from 0 to len1-1 do
    if hash_s1[i] equals 1 then
        while hash_s2[point] equals 0 do
            point = point + 1 // Find a matched character in s2
        end while
        if s1[i] does not equal s2[point++] then
            t = t + 1 // If characters don't match, increment transposition count
        end if
    end if
end for
t = t / 2 // Calculate half of the transpositions
similarity = (match / len1 + match / len2 + (match - t) / match) / 3.0 // Calculate Jaro-Winkler
similarity
return similarity // Return Jaro-Winkler similarity
```

3.5. Suggestion algorithm

The suggest algorithm is designed to offer recommended corrections for potentially misspelled words. It operates by comparing the input word to a dictionary of words stored in a file. Using Levenshtein and Jaro Winkler distance metrics, it assesses the similarity between the input word and each dictionary word. The algorithm updates an LRU cache with words exhibiting the lowest Levenshtein distance or, in case of a tie, the highest Jaro Winkler similarity. After scanning the dictionary, it prints all the elements in LRU Cache i.e. the top suggestions present in the dictionary.

3.5.1 Time Complexity

The time complexity of the Suggestion algorithm is O(number of words in dictionary) since it traverses through the whole dictionary to find the best top MAX_SUGGESTIONS words and the LRU Cache only takes O(1) complexity to insert each better suggestion in Cache.

3.5.2 Space Complexity

The space complexity of the Suggestion algorithm could be attributed to Levenshtein(O(mn)), Jaro Winkler (O(1)) and the LRU Cache (O(MAX_SUGGESTIONS)).

Algorithm 5 Suggest(word, obj, suggestions)

```
declare dict_word[MAX_LENGTH + 1] // Temporary variable to hold a word from the dictionary

declare tempJaroWinklerValue = 0 // Temporary Jaro-Winkler value
declare tempLevenshteinValue = INFINITE // Temporary Levenshtein distance value
file = open("dictionary.txt", "r") // Open the dictionary file for reading
while not end of file (EOF) do
    read a word from the file into dict_word
    distance = levenshteinDistance(word, dict_word) // Calculate Levenshtein distance
    jaroWinklerValue = jaroWinklerDistance(word, dict_word) // Calculate Jaro-Winkler distance

    if distance < tempLevenshteinValue then
        tempLevenshteinValue = distance // Update Levenshtein distance value
        lRUCachePut(obj, dict_word) // Add word to the LRU cache
    else if distance = tempLevenshteinValue and jaroWinklerValue >= tempJaroWinklerValue
    then
        tempJaroWinklerValue = jaroWinklerValue // Update Jaro-Winkler distance value
        lRUCachePut(obj, dict_word) // Add word to the LRU cache
    end if
end while
close the file // Close the dictionary file
print the elements of LRU Cache which are the suggestions
```

4. Program Flow

The program starts by loading the Tries and Bloom Filter with the dictionary text file containing approximately 370k words. It uses multi-threading to speed up this process. It then offers the user with 3 modes:-

1. Spell Checking mode for spell checking few sentences.
2. Analysis mode for comparing the accuracy and speed of Tries and Bloom Filter based on 2 variables(number of hashes in bloom filter and the size of bit array).
3. The optimisation mode where a large text input file is read and all the incorrect words along with their suggestions are displayed with the help of Multithreading and LRU Cache which speeds up the process of spell checking and displaying the suggestions for incorrect words.

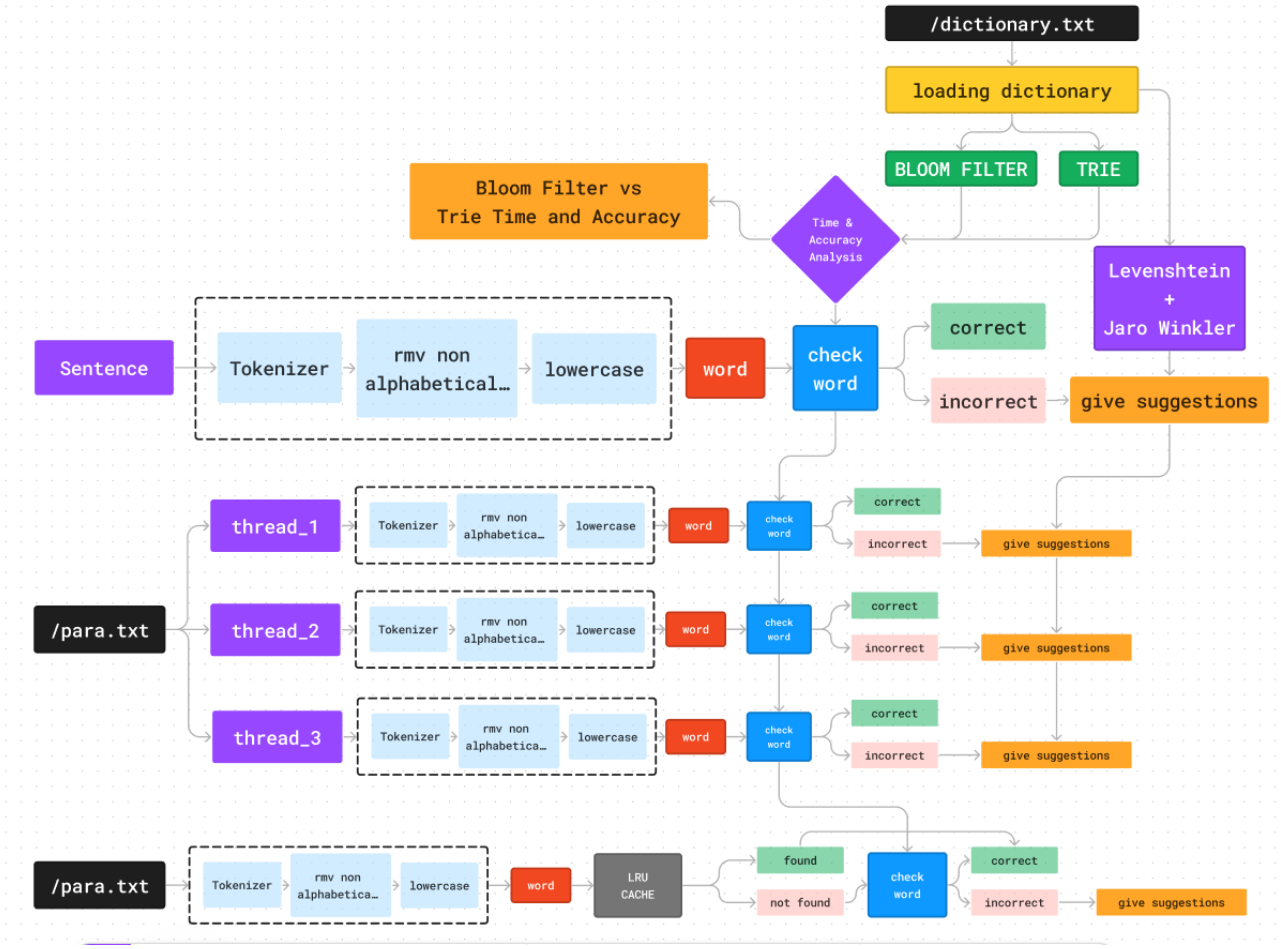

Figure 6: Program Flow Block Diagram

5. Analysis Report

Following are the major conclusions we have drawn from the Spell Checker project:-

1. Using a bigger bit array and more hash functions for bloom filter can result in better accuracy of spell checking but it comes with the cost of more memory and more time complexity respectively. User can compare the speed vs accuracy of bloom filter in the analysis mode.
2. Multithreading improves the speed of loading dictionary and reading the input file for spell checking. More threads means higher speed but leads to racing of threads where threads can print output one over another and also lead to data corruption where dynamic data structures.
3. LRU Cache is very useful in assisting of spell checking of correct words which are frequent or least recently used by giving the membership in O(1) average time complexity.

6. Conclusions

In summary, our C-based spell checker project has successfully combined advanced techniques and tools to create an efficient and accurate spell checking system. We used clever data structures and algorithms like Tries, Bloom Filters, Levenshtein and Jaro Winkler distances, LRU Caches, and multi-threading to make the spell checker work really well. It doesn't just find and fix misspelled words, but it also gives you suggestions quickly and accurately. We've tested it thoroughly in different situations, and it's proven to be effective. This project lays a strong foundation for future improvements and applications in the world of language processing.

7. Bibliography and citations

Acknowledgements

Special mention to Dr. Anil Shukla Sir and our TA Soumya Sarkar Sir for helping and guiding us in our project. We had a great learning and fun in practically applying the data structures and programming fundamentals on a real project scale.

References

[1] Portfolio Courses. Multithreading.

[2] Geeks for geeks. Levenshtein.

[3] Geeks for Geeks. Lru cache.

[4] Geeks for Geeks. Tries.

[5] Gaurav Sen. Bloom filter.

[6] York University. Djb2.

[7] Wikipedia. Jaro winkler.

[8] Wikipedia. Jenkins hash.