Java Text Indexing System: Implementation and Comparison of Search and Indexing Algorithms

Vinicius S. Boldan[1], **Nicolas D. D. Rodrigues** [2]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul - UFMS
CEP 79070-900 – Campo Grande – MS – Brazil

{boldan.vinicius, nicolas.dourado, bruno.nogueira}@ufms.br

Abstract. *This study presents a text compression and word indexing system leveraging Trie and Huffman Trie data structures. We aimed to evaluate the efficiency of data structures explored in-class. Java and Python were utilized for object-oriented development and PDF text extraction, respectively. Key findings include process execution time analysis and memory consumption estimation.*

Resumo. *Aqui desenvolvemos um sistema de compressão de textos e indexação de palavras baseado nas estruturas de dados Huffman Trie e Trie. Buscamos analisar a eficiência das estruturas estudadas e desenvolvidas em aula. Utilizamos principalmente a linguagem Java, devido ao seu paradigma orientado a objetos, e Python para extrair textos de artigos científicos em PDF. Conseguimos analisar os tempos de execução dos processos e estimar o consumo de memória.*

1. Introduction

Compressing data is essential for many reasons; raw data is usually large, which leads to slowness on data transmission, writing and reading; so along the story scientists like David Huffman had to find out how to store data more efficiently. Compression is one of the methods, and consists basically on removing any redundant information but in a way a receiver can decode the original information.[1]

1.1. Huffman Algorithm

The Huffman algorithm is a fundamental method for compressing text because it uses lossless compression. This means we can reproduce the exact text information as it was before compression, which is essential for text comprehension. In other formats, some *lossy* compression can often be allowed because the information remains comprehensible despite minor losses.

In Huffman's algorithm, information is represented using only 1's and 0's, and the size of the code for a symbol is inversely proportional to its frequency. Additionally, it employs prefix-free encoding, meaning no code is the prefix of another.

We use this algorithm to compress a collection of text files. It operates using a Trie tree, where each path to a symbol represents a specific codification. Moving to the left in the tree adds a 0 to the code, while moving to the right adds a 1. The tree is built from the bottom up, starting with the least frequent symbols. Their frequencies are summed to create a new node, and this process is repeated for the remaining nodes using a priority queue. This ensures the essential operations of the Huffman algorithm. The system also indexes word from these texts using a Trie.

1.2. Trie Structure

A Trie is a specialized tree data structure designed to store strings efficiently, where each node represents a character in a word, and a path from the root to a leaf represents an entire word. This makes the Trie particularly suitable for tasks like word searches, auto completion, and dictionary-like operations, as it allows quick lookup, insertion, and deletion of strings.

But we added a key feature for word searching; every node that represents the end of a word also stores a linked list with the titles of the texts that contains the word we are searching for.

1.3. Hash Tables

One common element in Trie and Huffman structure is the use of key-value pairs, what takes us to Hash Tables. Hash tables store data in key-value pairs, where each key is unique and maps to a specific value. Hash functions convert keys into indices, allowing for fast access. With average time complexities of O(1) for search, insert, and delete operations. However the worse case for these operations is O(n), that happens if many values are mapped to the same position.

2. Implementation

Java has its own structures in form of interfaces and concrete classes that we can use, and we use some of them in this study, but we want to test our own implementations.

2.1. Texts

For the tests, we downloaded 30 articles from *arXiv* and used *PyMuPDF*, a Python library, to extract text from the PDFs and write it into .txt files, enabling us to read them using default Java libraries. To minimize extraction errors, all the articles are single-column format.

In the indexing phase, we aim to search for 100 terms. To ensure relevance, all the articles are in English and pertain to computing. This allows us to identify terms that are present and absent, including terms like Portuguese words. Of the 100 terms, 90 appear in at least one article, while 10 do not appear in any.

2.2. Huffman Code

We implemented the main logic of Huffman algorithm to compress the data; so we map the frequencies of each symbol, store the key-value pairs (frequency-symbol) then bottom-top build the tree with the nodes according to the frequencies (from less frequent to the most frequent). But it uses helper structures, like Hash Map to temporarily store the key-value pairs, then a temporary priority list to order the nodes according to their frequencies, and finally we build the tree with those nodes; The helper structures used were java default. After writing the entire system we compresses the texts and measured the time it took to compress everything.

2.3. Hash Table

After compressing the text from an article we get its title and its compressed content and map using our own hash table. To get the positions a user could select between two hashing algorithms: hash by division or DJB2 hash[2]. Here we measure the distribution of the table after inserting all the compressed content. As a Hash Table has a search time of O(1) in best case and as we used the titles as keys the insertion and search time of our table is negligible. What can makes a hash table less efficient is the distribution of the content. If we have a table where only one position is occupied by many elements then the search time for that table is O(n).

To get a better distribution we first set the size of our table to 31, a prime number; so there is 31 positions for 30 articles. We put the distributions for both hashing algorithms in a graphic on the results section. After that we wanted to compare with different sizes, so we made another 2 tests with prime number for size, 29 and 37.

3. Trie Indexing

. We implemented a version of Trie tree that also stores a list of documents that contains the words. Each node is hash map where the key is the character and the value is another node. Adding another word to the the Trie means adding another path to a leaf node; leaf nodes have a boolean flag representing the end of the word. In our case the end word node also have a java default linked list, with the titles of the articles containing the word. Therefore when we add a word already stored it simply appends a new title to this list if its also not there yet. The time for that indexing is very interesting and its in the results section.

3.1. Measuring time and Memory

We have specific methods to do the compression, indexing and search. So we just get the current time with java default method getCurrentTimeMilis() before and after the execution of the methods and calculate the difference. We executed each part 5 times to calculate average time. To get memory consumption we used java default method getRuntime().getfreeMemory() that returns in bytes the current available memory, again we calculate the difference. There was a helper method to write the collected data in txt files on each execution.

The source code and and terms we used are on github, the link will be at appendix section.

4. Results and Comparison

After making the whole code works fine we wrote a shell script to run the code automatically, 5 times each part, except the distribution of our hash table, because its always the same for each hash algorithm, so we had to get the distribution only once for each.

4.1. Hash Distribution

The figure 1 shows a graphic comparing the amount of values per position of each hashing algorithm.

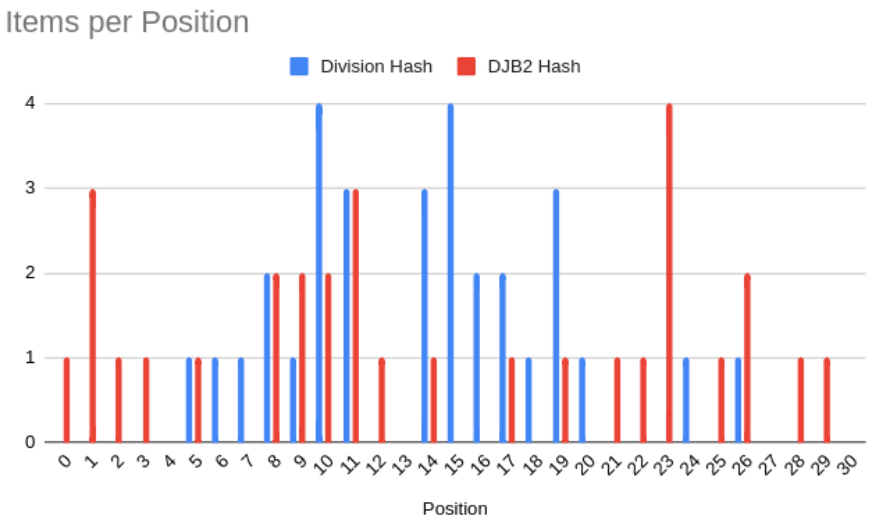

Figure 1. Comparison between distribution of two hash algorithms

The DJB2 algorithm was slightly better that hashing by division. DJB2 occupied 20 positions while the other occupied only 17, resulting in worse distribution, what leads to worse performance when trying to search values stored in positions like 10 and 15. At that point we were convinced that DJB2 have a better distribution, but we needed more evidence.

4.1.1. Different Table Sizes

To make sure of the superiority of DJB2 over division we made two more tests using table sizes of 29 and 37, both results are in figure 2 and 3 respectively.

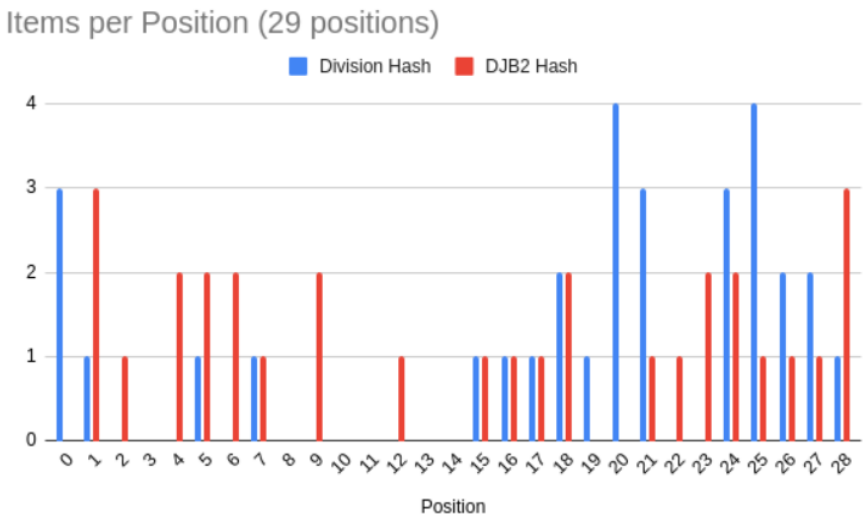

Figure 2. Comparison between distribution of two hash algorithms for 29 positions

Again the max number of items per position was 4. This time division occupied 16 positions while DJB2 occupied 21 positions, i.e. DJB2 made a better use of table space.

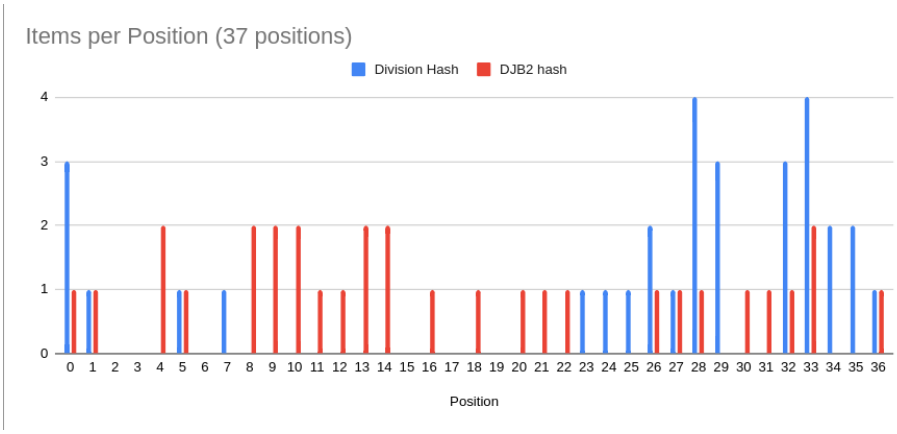

Figure 3. Comparison between distribution of two hash algorithms for 37 positions

In this case we can clearly see the difference in the graphic. Division occupied only 16 positions while DJB2 occupied 24.

4.2. Compression Time

The values store in our hash table were the compressed texts; while mapping them was fast and easy compression took longer to finish. We made the table 1 with the time and estimated memory consumption of each execution.

	Time (minutes)	Memory (Mb)
	2,749	267,092
	2,757	222,819
	2,754	167,902
	2,775	191,063
	2,780	298,899
Average	2,757	222,819

Table 1. Indexing time and memory consumption of each execution and is averages

The average time was about 2,727 minutes, or 2 minutes and 45 seconds or 5,5 seconds per file. Or course the time per article vary because they have different sizes and symbols. But the problem here was to estimate memory usage. The SO manages the memory of the processes and the java method we used returns an estimative; furthermore the free memory depends on many factors; that is why the table shows a very different values of memory for each execution. This is not a precise measurement and the real value of memory the Huffman Trie and its compressed texts consumes is probably very different from the values we measured. The same problem comes when we indexed using Trie.

4.3. Indexing and Search Time

Measuring the time was quite easy for any process, but we did not expect the Trie to be so fast as we show in the table 2.

	Indexing Time (ms)	Memory (Mb)	Search time (ms)
	862	2,134	1
	828	4,134	0
	845	7,020	0
	1093	3,866	0
	944	7,980	0
Average	862	4,134	

Table 2. Indexing time and memory consumption of each execution and is averages

In every execution we read the file with the terms to search, store in a list and throws each of the terms to the trie.search() method. We get the time right before we search the first term and right after we search the last one. Even during the tests of the code the search time was 0 ms is most executions. It is quite impressive but not actually a surprise, since we now this structure uses hash map to go to the next characters of the word, and the time to get each character and go to the next nodes is ridiculously short, basically a sum of O(1) to search an entire word..

5. Conclusion

However we implemented the structures and were able to measure the execution time of the data structure operations and compare them. We also measured the distribution of our hash table comparing two algorithms. But the memory usage of the structures was not precise, this is the first improvement we could make if this project continued. Another thing that could be done is to do not use any java default structures at all, implement everything on our own and see what happens. By building this project we were able to get a better understanding of the data structures studied in class.

6. References

.

[1]Sanjeev R. Kulkarni. Information, Entropy, and Coding. 2002. url: https://www.princeton.edu/ cuff/ele201/kulkarni$_text/information.pdf$

[2]Filip Stanis. Djb2 Hash. 2024. url: https://theartincode.stanis.me/008-djb2/

7. Appendix

Github link: https://github.com/ViniciusDSB/dataStructure-project