DS 4300 - Practical 01 Analysis Report

Spring 2025

Chairlift Chillers

	Name (in GradeScope)	NU Email Address
Member 1	Heidi Eren	eren.h@northeastern.edu
Member 2	Lily Hoffman	hoffman.li@northeastern.edu
Member 3	Mihalis Koutouvos	koutouvos.m@northeastern.edu

Additional Data Structure Implemented: **List**

4-Person Team Additional Data Structure or Task: **N/A**

Introduction

Our goal is to evaluate the performance of various in-memory indexing data structures—Binary Search Tree, AVL Tree, Hash Table, and List—in building an efficient search engine over financial news articles. This exercise mirrors the role of indexes in a Relational Database Management System (RDBMS), critical for speeding up data retrieval. By comparing these data structures, we aim to recommend the most effective method for optimizing search operations, similar to RDBMS indexes. This practical helps us prepare for real-world database management and highlights the significance of choosing the right structure for performance optimization in data-heavy environments.

Data Structure Implementations

In our practical, we worked with four data structures: AVL Tree, Binary Search Tree (BST), List, and HashMap. We implemented three of these structures, while the BST was provided by Professor Fontenot, and we did not modify his implementation. For the AVL tree, Professor Fontenot provided partial code with gaps in the functions that we needed to complete. To implement this, we created the left and right rotation functions, allowing the tree to rebalance itself when encountering any of the four imbalance cases. We then implemented the recursive insert function, which identified imbalance cases using conditional statements and applied any rotations. Then, our team implemented a HashMap using separate chaining. A key feature of our implementation is that it preserves the order in which keys are added, allowing us to retrieve them in sequence. Finally, the List indexes documents through sequential insertion and search operations. Each word extracted from a JSON file is stored as a key, with the filename stored as the corresponding value. These key-value pairs are maintained as a list of lists for all words found across files. The search function iterates through each key-value pair. If a match is found, it returns the result; otherwise, it continues searching. Finally, we crawled through the folders of news articles to obtain metadata, including the title, source URL's domain name, and the author's last name (if available).

Experimental Methodology

The input data set for the indexing data structures is based on the US Financial News Article dataset from Kaggle. It contains a folder from each month between January to May 2018 with around 60,000 JSON files of news articles. Each file contains metadata about the article, including the author, url, and preprocessed text that removes stop words, digits and decimal places, and lemmatizes the full text. search_function.py constructs each indexing structure with the input data set. By pulling the title, author, url, and preprocessed text from each JSON file, they are tokenized to ensure that there are no whitespaces, capitalization, stop words included. These strings are then separated by word and are inserted into each indexing structure using the .insert(word, filename) method, where the key is the word and the value is the filename. To be able to access after indexing once, each constructed indexing structure is then converted into a pickled format.

To test the performance of such indexing structures, eight search datasets were generated in the search_data_set.py and saved into one JSON file to be able to access for

experiments. The process of generating these search datasets involve multiple components and functions. Component A creates a random sample of n terms or tokens from the index, where n is a multiple of 4 and at least 4000. Component B adds an additional set of (n/4) two- and three-word phrases, formed by randomly selecting 2 or 3 tokens from Component A. Component C introduces n randomly generated strings of characters that are unlikely to appear in the index, while Component D adds another set of (n/4) 2- and 3-word phrases from Component C. These components are then shuffled together to form a search dataset. In the search_data_set.py, the multi_phase function creates the phrases used for Component B and D. The generate_n function randomly creates the value of n based on the provided constraints. The create_search_data_set function generates the dataset by compiling words from all components and shuffling them into a list of words. The load_index function opens the pickled indexing structure, and the reformat_dataset compiles and saves a dataset as a dictionary in a JSON file, along with metadata like the size of n, indexing structure, dataset number, and runtime. The main() function iterates through this process, ensuring each search indexing structure is randomly selected and that random words are pulled accordingly to generate 8 different search datasets.

Figure 1: Example of first search dataset

Using the 8 generated search data sets, in each experiment, we used our own timer to compare how long each data structure took to index each dataset using the time library. We subtracted our end time from our start time to return the search_time. The find_search_data_sets function reads a JSON file with news articles and pulls out specific datasets and their corresponding "n" values. It creates empty lists to store these datasets, reads the entire JSON file, and goes through each dictionary to get and add the dataset and "n" values to the lists. The function then returns these lists, giving us a well-organized collection of the data for further analysis.

In Experiment 1, we set up a clear process to measure how well our search index worked with eight different datasets. We created several datasets, each with different sizes and vocabularies, to cover a wide range of test scenarios. The experiment_searching function automated the search process by going through each dataset, searching for words in the index, and tracking how many documents and tokens were indexed. We carefully measured the time it took for each search operation, allowing us to evaluate how efficient the indexing was. We then summarized the results in a DataFrame, giving us a snapshot of how each dataset performed.

In Experiment 2, we developed a method to check how long it took to search for words

that were not in the index. The main goal was to count how many words in the datasets weren't found in the index and how long those searches took. Each dataset, with different sizes and vocabularies, was processed to ensure a complete range of tests. The experiment_missing_words function automated this process by going through each dataset and searching for words in the index. For each word that wasn't found, a counter was increased to track how many words were not indexed. We recorded the start and end times for each search to measure how long it took. The results were then put into a summary data frame.

In experiment 3, we developed a method to check how long it took to search for phrases containing either two or three words in them, separated by a space (' '). The main goal was to count how many phrases in the datasets were found in the index and how long those searches took. The function for this, find_phrases_in_datasets, followed similar procedures as the others, as it took found and checked two- to- three-word phrases from the datasets across the rest of the articles. The method times and results were then placed in a data frame.

Results

Index Characteristics

To evaluate whether the assertion of each index was properly constructed, we ran the indexing structure on a smaller dataset called 'P01-verify-dataset'. Once indexing structured pickles were constructed, we searched for the same set of three words from each indexing structure: 'northeastern', 'beanpot', 'husky'. By using the .search method in each indexing structure, we were able to identify the same set of news articles for each word, as shown below. This was an indication that each index was properly constructed, as each word was tied to the correct news article as a key-value pair.

(Figure 2: Example of the each index being properly constructed)

```
['preproc-news_0034466.json', 'preproc-news_0002807.json', 'preproc-news_0001728.json', 'preproc-news_0056098.json', 'preproc-news_0023198.json']
['preproc-news_0034466.json', 'preproc-news_0023198.json']
['preproc-news_0034466.json', 'preproc-news_0002807.json', 'preproc-news_0001728.json', 'preproc-news_0056098.json']
```

Ideally, all indexes should have the same number of keys because they were all preprocessed in the same fashion. Each unique word should be its own key and then be linked to its corresponding filename as the value. We indexed the same set of articles for each structure, so it would make sense that we are storing the same terms and words as keys. When comparing the data structures indexed with the verified dataset, there were 979 nodes for both AVL and BST indexing while 979 keys in list and hashmap indexing. Because all the indexing structures have the same number of nodes or keys, when searching for the same specified set of words to locate which articles contain that word, it makes sense that the articles located would be the same.

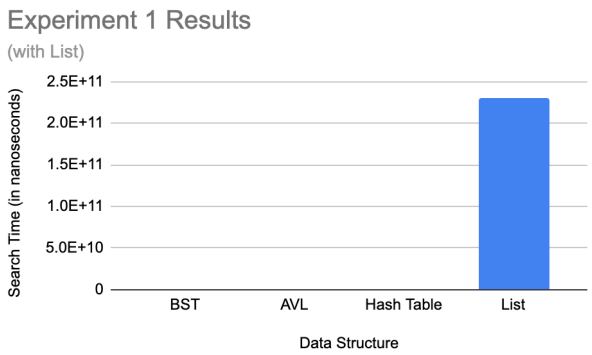
```
Index loaded from /Users/Heidi/Downloads/final_pickles/index-verify-bst.pkl
Index loaded from /Users/Heidi/Downloads/final_pickles/index-verify-avl.pkl
Index loaded from /Users/Heidi/Downloads/final_pickles/index-verify-hash.pkl
Index loaded from /Users/Heidi/Downloads/final_pickles/index-verify-list.pkl
Number of nodes in BST index: 979
Number of nodes in AVL index: 979
Number of keys in list index: 979
Number of keys in hash map index: 979
```

Figure 3: Output depicting size of indexing structure

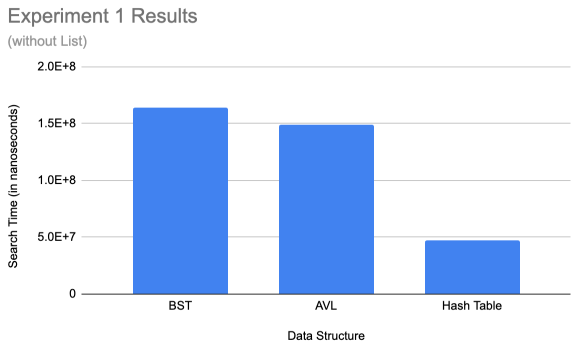
Experimental Results

We start with Figures 1 and 2 below, which highlight the speed in which the function relates to searching for words according to an index. According to Figure 2, the hash table had the fastest search time (which we predicted), followed by the AVL and binary search trees. However, the reason we included two graphs, with one lacking the list data structure, is because a list takes way too long for our search function to go through.

(Figure 4)

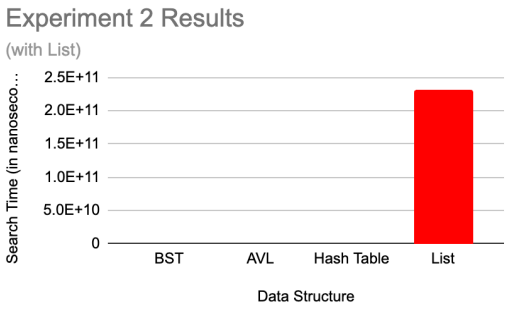


(Figure 5)

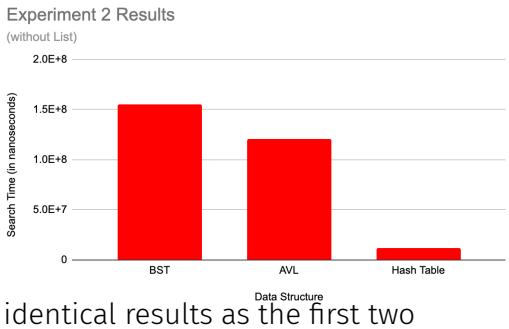


For our second experiment (see Figures 3 and 4 below), we had very similar performance results, with the hash table being the most time efficient, followed by an AVL and binary search tree, and capped off with the list. It is worth nothing, though, that the hash table took a slightly longer time to go through, but it ended up the quickest data structure nonetheless.

(Figure 6)

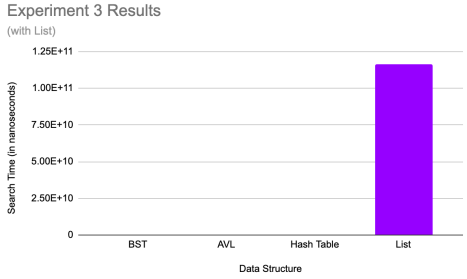


(Figure 7)

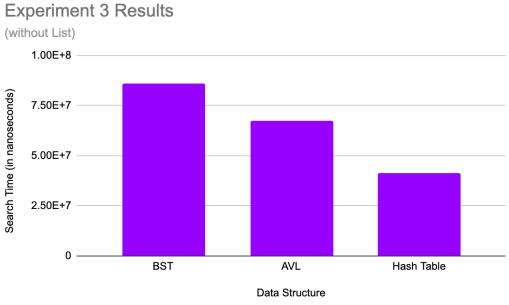


For our third and final experiment, we obtained almost identical results as the first two figures (See Figures 5 and 6 below). However, the hash table, although being the most time efficient once again, took the most amount of time during that experiment compared to the other ones. Such similar data performance indicates that there is some consistency within the results.

(Figure 8)



(Figure 9)



Discussion

We gleaned a ton of information about the data our group collected. Firstly, the data structure that proved to be the most time efficient is the hash table, as it remained the fastest data structure for indexing among the others. However, the hash table started pretty fast, then dropped to an insanely fast time, and then jumped up higher than the original time when it came to searching for phrases with multiple words.

For the most time inefficient, our group noticed it to be the list. One of the reasons why we had double the graphs was because we decided to focus on the time (in nanoseconds) of the other three data structures, which were significantly faster than the list (Figures 5, 7, and 9 do not include the list, while Figures 4, 6, and 8 do). Performance-wise, it did the best during experiment 3 compared to its other runs, but it was still nowhere near the other structures.

The order for the most time efficient data structures also remained the same, albeit with search times adjusting marginally depending on the task except for the list, which doubled in time in experiments 1 and 2 compared to experiment 3.

We were most shocked by the AVL tree being faster than the BST. We assumed that due to the rebalancing that needed to happen when one of the four imbalance cases appeared, the search time would increase. However, our results proved otherwise.

Conclusion

In our experiment, the data structure that took the longest to process was the list, which showed slower indexing times compared to others. On the other hand, the hash table proved to be the quickest in indexing. Key takeaways from the experiment suggest that while some data structures excel in speed, they may also be limited by factors such as scalability and memory usage. A few limitations in our implementation include experimental reproducibility issues, as machine-specific variations (like CPU speed and RAM availability) slightly impacted results. For instance, Lily's (Intel Core i5) laptop took longer for Experiment 1 compared to Mihalis's (he has an Apple M2 chip). Additionally, the experimental design had limitations, such as a limited number of trials, which could have introduced statistical noise and a static dataset that did not allow for real-time indexing tests. We also did not incorporate the timer decorator, but we used our stopwatch to measure indexing times for each article in the specific data structures.