DS 4300 - Practical 01 Analysis Report

Spring 2025

GMC

	Name (in GradeScope)	**NU Email Address**
Member 1	Grace Kelner	kelner.g@northeastern.edu
Member 2	Claire Mahon	mahon.c@northeastern.edu
Member 3	Milena Perez-Gerus	perez-gerus.m@northeastern.edu
Member 4 (optional)		

Additional Data Structure Implemented: **Sorted List**

Introduction

Digital information continues rapidly expanding, therefore efficient data retrieval is essential. This project explores different in-memory indexing data structures to determine the most effective and efficient approach for building a specialized search engine. The goal is to identify the best data structure for quick searching through a large corpus of financial news articles, retrieving relevant results, and optimizing search performance.

From a broader perspective, this project simulates how databases optimize search queries through indexing. By comparing various data structures, we can gain insights into how different approaches impact search speed and efficiency. Understanding and analyzing these trade-offs is crucial for developing scalable and high-performance information retrieval systems. The results will offer recommendations for selecting the best indexing approach, based upon the search speed in different indexing structures.

Data Structure Implementations

Binary Search Tree: Our binary search tree (BST) implementation supports insertion, searching, and traversal. Each node contains a key, a list of associated values (e.g., multiple document IDs per term), and pointers to left and right children. The BST Index class manages the tree's root and provides methods for inserting, searching, and utility functions. Insertion compares the key with the current node's key, navigating left for smaller keys or right for larger ones, appending values to the list if the key exists. Searches follow a similar recursive approach. The tree also supports in-order traversal to retrieve keys in ascending order and offers additional methods for counting nodes, calculating tree height, and finding the average length of value lists.

AVL Tree: The AVL Tree implementation with inherited classes from the BST implementation is a self-balancing binary search tree (BST), to index keys and their associated values. When inserting a key-value pair, the tree is checked for imbalance (the height difference between the left and right subtrees of a node must be at most 1). If any node becomes unbalanced during insertion, the tree performs rotations (left or right) to restore balance. Right Rotation (performed when a left-heavy subtree is detected) or Left Rotation (performed when a right-heavy subtree is detected). The tree calculates the balance factor of each node by comparing the heights of its left and right subtrees. After each insertion, the tree checks the balance factor of the node and performs the necessary rotations to maintain balance.

Hash Map: This implementation provides a hash-map based index using a hash table, which allows efficient insertions, deletions, and searches. The core data structure of the hash map is a hash table, which is an array of "buckets". Each bucket is a list of key value pairs where the key is a term and the values are a set containing document IDs that referenced that term. The hash table is resized when the load factor, which is the ratio of the number of elements and to the table size, exceeds 0.9. When the table grows, the rehash function redistributes the existing elements into a new, resized table.

Sorted List: This implementation (in the SortedListIndex class) is a simple index that uses a sorted list to store key-value pairs, where the key is either words or character strings and the value is a list of associated file names. The class maintains the list in sorted order so that insertion and searches can be done efficiently using binary search. This allows for a quicker search time than an unsorted list, as the search is able to split and evaluate where a term would be in its sorted order.

Experimental Methodology

Dataset EDA and Preprocessing: The dataset consists of 306,242 finance-related news articles from January to May 2018, organized into five subfolders corresponding to each month. Each article is stored as a JSON file containing both metadata (title, source domain, author) and preprocessed text. The text has been tokenized and cleaned for indexing. Terms from the preprocessed text were indexed using four data structures: a binary search tree (BST), AVL tree, hash table, and sorted list.

Experimental Search Sets: To evaluate search performance, we generated search datasets by extracting terms from the index and creating random search queries (via the Searching Set class). These queries were designed to test index efficiency with varying search set sizes. The experimental variable was the index structure, with searches performed on each structure using different dataset sizes.

Measuring Index Performance: Search efficiency was measured using a timer decorator, and the time taken to retrieve results was recorded using the search_index_for_experiments function. Queries were split into individual words, and results were identified by intersecting document sets for matching terms. If no documents matched, an empty list was returned. Using the searching sets, we were able to validate each algorithm had correctly searched for and found the same intended values. All results were stored in timing_data.csv. The experiment was repeated using the run_experiment function, which initialized datasets, performed searches on all index structures, and recorded the search times, allowing for a performance comparison across structures.

Results

Index Characteristics

- The Binary Search Tree was constructed by indexing a total of 306,242 news articles. The BST had 308705 nodes and a height of 55. The average list length was 130.
- Next, our AVL Tree was constructed by indexing a total of 306,242 news articles. The AVL tree had 308706 nodes and a height of 22.
- The hash map was constructed by indexing a total of 306,242 news articles. The hash map resulted in having 662,601 terms indexed. Each token corresponds to a set of document IDs. The hash map resulted in having 786,432 buckets.
- Finally, our Sorted List was constructed by indexing a total of 306,242 news articles, resulting in a sorted list length of 308705.

Experimental Results

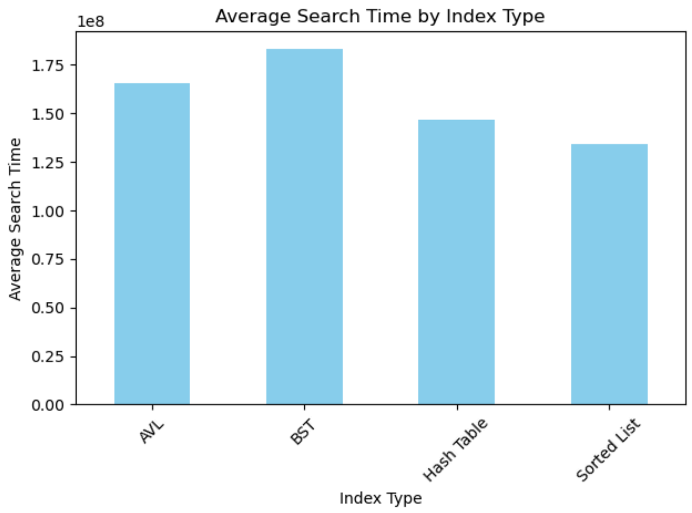

Figure 1.

The sorted list had the lowest search time (in Nanoseconds), followed by the hash table. Since the sorted list has the lowest average search time, followed closely by the hash table, this suggests that these two index types are the most efficient in terms of search time. The BST tree had the highest average search time, followed by the AVL tree.

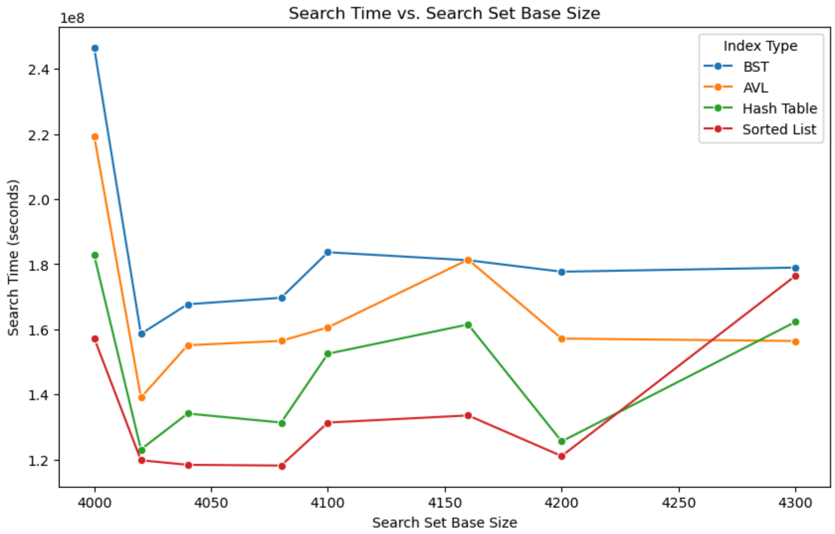

Figure 2.

Figure 2 shows that the sorted list, followed by the hash table, had generally the lowest search time for every search set base size besides 4300, showing that as the dataset grows, the searching becomes slower. The BST tree had the highest search time, but as the search dataset grew to the maximum of 4300, its search time was very similar to the search time of the sorted list. Additionally, although the sorted list and hash table generally had the lowest search time, the AVL tree had the lowest search time out of all three index types at the maximum search set base size of 4300.

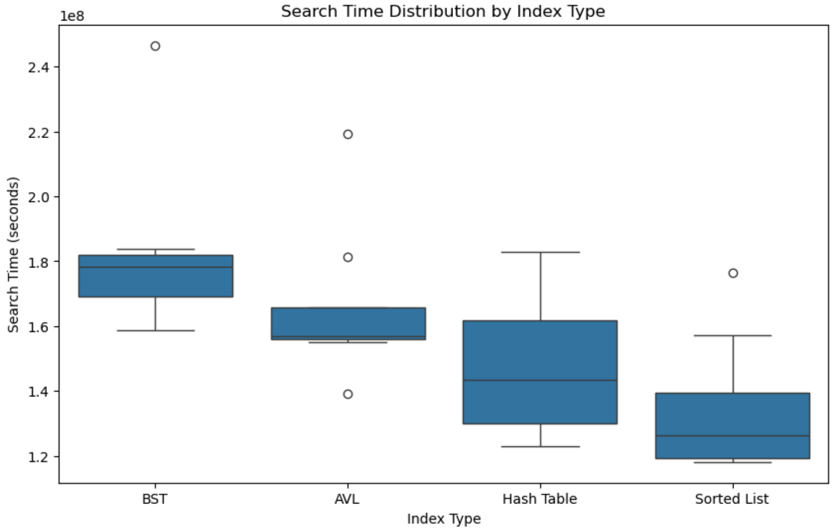

Figure 3.

Figure 3 shows the median variation in search times by index type. Sorted list has the lowest search time and the BST tree has the highest search time. Hash table follows behind sorted list for the lowest search time, however the hash table has the greatest amount of variance in search times, demonstrating inconsistency in performance with varying dataset sizes. Additionally, although the BST tree had the highest search time followed by the AVL Tree, these two had the least amount of variation in search times, demonstrating a more stable and predictable search time with varying dataset sizes.

Discussion

The results indicate that the sorted list consistently demonstrated the most efficient search times across the different index types. However, as search set sizes increased, its performance advantage diminished, particularly at the largest search set size, where its search time converged with that of the BST. The AVL tree, while initially slower, became more competitive at larger search set sizes and even outperformed the hash table at the largest search set size. The BST maintained the highest search times throughout, but its performance became comparable to the sorted list at larger dataset sizes.

One of our key findings is that while hash tables generally provide fast search times, their advantage weakens as dataset size scales up. This could be attributed to increased computational power needed as a hash table becomes larger -> more need for resizing when buckets become full. The variability in search time for the hash table, as observed in Figure 3, further supports the idea that hash-based indexing may not provide consistently predictable performance across different dataset sizes.

The AVL tree's stability in search time performance is noteworthy. Despite having a higher average search time than the sorted list and hash table for smaller datasets, its performance remained steady across dataset sizes, making it a reliable choice for potential applications where consistency is crucial. The BST, while generally the slowest, exhibited similar stability in search times, suggesting that balanced tree structures provide more predictable performance.

From a scalability perspective, BST and AVL trees demonstrated increasing search times as expected, but they did not degrade as rapidly as anticipated. The sorted list performed well even at higher search set sizes, likely due to its efficient binary search mechanism. Hash tables, while excelling in small-to-medium-sized datasets, did not maintain performance at larger dataset sizes, potentially due to increased overflow within buckets and large computations for rehashing.

Conclusion

Overall, the results highlight that different indexing structures offer distinct advantages depending on dataset size and performance requirements. Initially, sorted lists performed the best, but by the largest dataset size, all index types exhibited similar search times. This suggests that while hash tables are highly efficient for small datasets, their advantage narrows as datasets grow, making AVL trees and sorted lists viable alternatives for large-scale applications.

One limitation of the implementation is that the dataset was restricted to a fixed set of financial news articles from 2018. Expanding the dataset to more diverse domains could provide additional insights into index performance. Additionally, while search times were measured, we did not analyze memory usage or insertion efficiency in depth. Future experiments could incorporate these factors to gain a more comprehensive understanding of the trade-offs involved in choosing an indexing structure.

In terms of experimental design, while different search set sizes were tested, additional experimentation with varied query complexities could offer further insights. Investigating how different indexing structures handle complex multi-word queries or phrase searches could provide valuable findings for real-world applications. Finally, while our experiment focused on search efficiency, incorporating user experience metrics, such as query latency across different computers (ours were unfortunately all the same type of computer and system) and environments, would give a more in depth evaluation of indexing strategies.