

Group Project: Search

**BST-ARS Bears**

Harrison Baker

Manuel De La Rosa M

Jane Nim

University of Central Arkansas

CSCI 2320: Data Structures

Professor Talley

Nov 18, 2024

## Introduction

The objective of this project is to deepen our understanding of key concepts in computer science, particularly those related to search Abstract Data Types (ADTs), algorithm complexity, and collaborative software development. By focusing on search ADTs, we aim to gain insights into the structure and functionality of data handling methods, which are essential for efficient data retrieval and manipulation. This project also explores the impact of algorithm complexity, particularly Big O notation, on search performance—a critical factor when working with large datasets or real-time applications. We will be testing on five search ADTs: Linear Search using Linked List (`std::list`), Binary Search Tree (BST), AVL, Hash Table, and B-Trees.

## Method Experiment #1:

After developing multiple ADTs, we implemented a random array of 10,000 numbers, we ran the array through each of the search ADTs and compared the time it took to search through the array.

## Results Experiment 1:

ADT	Insert time (ms)	Successful Inserts	Failed Inserts	Search Time (ms)	Successful Searches	Failed Searches	Delete Time (ms)	Successful Deletes	Failed Deletes
Linked List	2	10000	0	28	1000	100	25	1000	100
BST	4	10000	0	0	1000	100	0	1000	100
AVL	4	10000	0	0	1000	100	1	1000	100
Hash	1	10000	0	0	1000	100	0	1000	100
B-Tree	2	10000	0	3	1000	100	4	1000	100

**Method Experiment #2:**

After developing multiple ADTs, we implemented a random array of 1 million numbers, we ran the array through each of the search ADTs and compared the time it took to search through the array.

ADT	<b>Insert time (ms)</b>	Successful Inserts	Failed Inserts	<b>Search Time (ms)</b>	Successful Searches	Failed Searches	<b>Delete Time (ms)</b>	Successful Deletes	Failed Deletes
Linked List	191	1000000	0	3,299	1000	100	3524	1000	100
BST	709	1000000	0	2	1000	100	1	1000	100
AVL	1745	1000000	0	2	1000	100	3	1000	100
Hash	218	1000000	0	0	1000	100	0	1000	100
B-Tree	129	1000000	0	174	1000	100	171	1000	100

**Method Experiment #3:**

After developing multiple ADTs, we implemented a random array of 10 million numbers, we ran the array through each of the search ADTs and compared the time it took to search through the array.

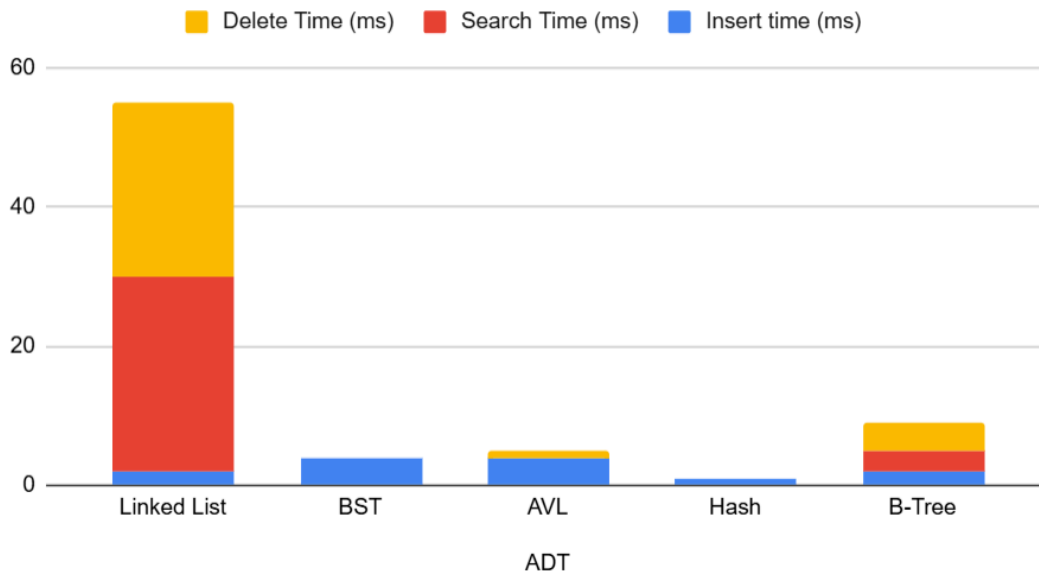
ADT	<b>Insert time (ms)</b>	Successful Inserts	Failed Inserts	<b>Search Time (ms)</b>	Successful Searches	Failed Searches	<b>Delete Time (ms)</b>	Successful Deletes	Failed Deletes
Linked List	2182	10000000	0	35739	1000	100	37311	1000	100
BST	12319	10000000	0	2	1000	100	3	1000	100

ADT	Insert time (ms)	Successful Inserts	Failed Inserts	Search Time (ms)	Successful Searches	Failed Searches	Delete Time (ms)	Successful Deletes	Failed Deletes
AVL	16950	10000000	0	2	1000	100	4	1000	100
Hash	2767	10000000	0	0	1000	100	1	1000	100
B-Tree	1561	10000000	0	1745	1000	100	1738	1000	100

## Graph Analysis Section

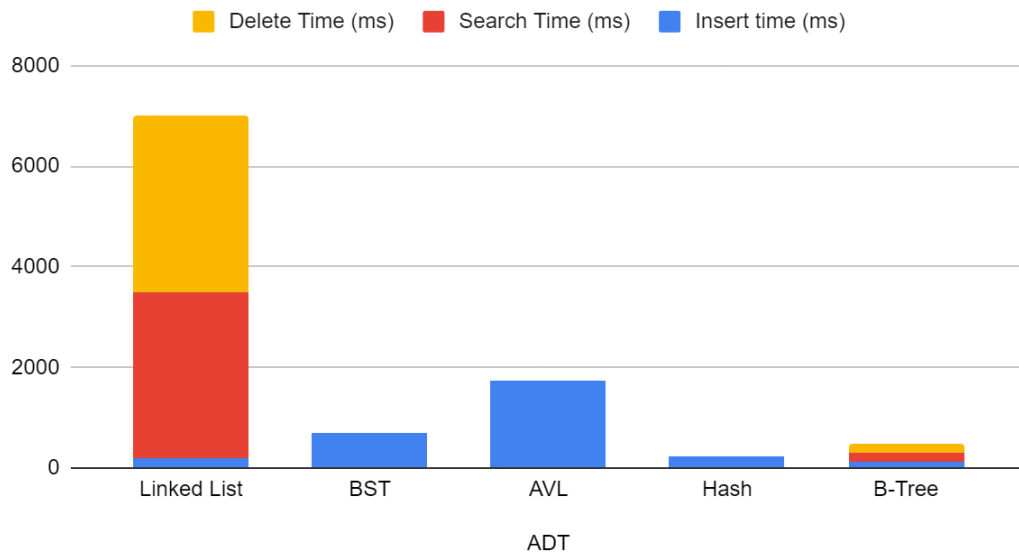
*10k dataset:*

10k Data set

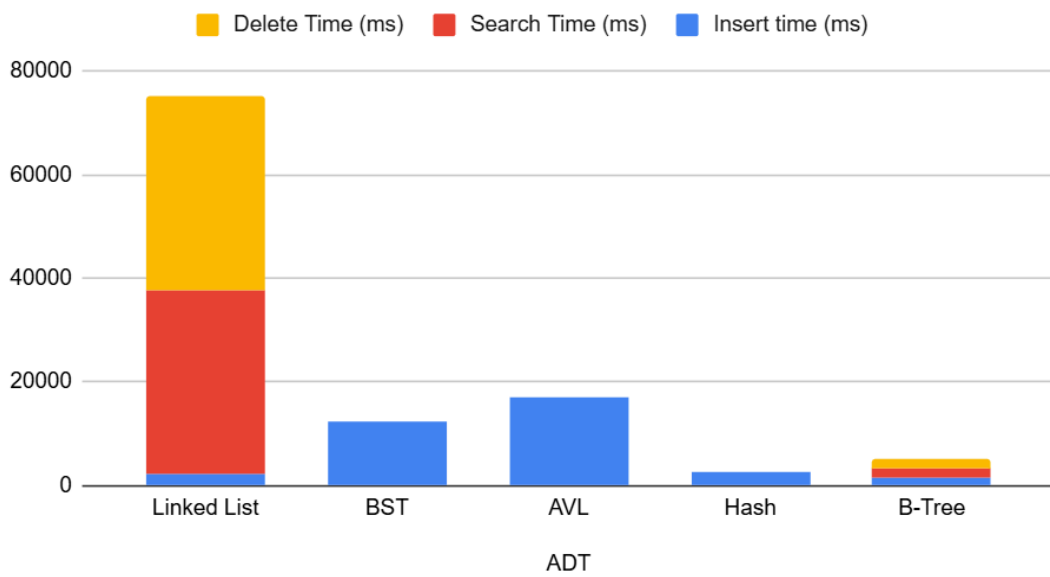


***1 million dataset:***

1 Million Data set

***10 million data set:***

10 Mil. Data Set



## **Conclusion for the graph and data analysis.**

To begin with, as a disclaimer, all the data was collected from a single computer to avoid external confounding variables, such as differences in system performance. Although an attempt was made to analyze a dataset of 100 million values, it could not be completed, so only the other three datasets were used for analysis. In this analysis, the data is being added between all the experiments and then taken into consideration unless specified otherwise.

From the data obtained, it is evident that the "Hash ADT" consistently demonstrated the fastest performance across all cases, regardless of dataset size. Specifically, for smaller datasets in this implementation, the "BST ADT" was the second fastest, closely followed by the "AVL ADT" and the "B-Tree ADT." As the dataset size increased, the time difference between the "B-Tree" and the "BST" narrowed, with the "BST" eventually surpassing the "B-Tree" in total time across the three datasets (making it slower than the "B-Tree" overall).

In summary, the "B-Tree ADT" resulted as the second fastest, and the "BST ADT" was the third. The "AVL ADT" was the fourth fastest. Although the "AVL" had faster delete and search times than the "B-Tree" in all experiments, its slower insertion time made it slower overall when considering the total time for all operations. The slowest ADT in every case was the "Linked-List ADT." While it had one of the fastest insertion times, its performance was significantly hampered by the time taken to delete and search for values, making it the slowest overall when considering the combined time for insertion, deletion, and search operations.

## **Fibonacci Hashing method**

According to *OpenGenusIQ*, The Fibonacci hashing method is a technique used in hash functions, particularly for calculating the hash value of a key. It uses a multiplier derived from the golden ratio (often denoted as  $\phi$ , approximately 1.6180339887) to calculate the index in a

hash table. The golden ratio is used because it has desirable mathematical properties, specifically for ensuring a uniform distribution of hashed values.

The primary reason why Fibonacci hashing tends to perform well is because of the **irrational nature of the golden ratio**. This irrationality ensures that the multiplication of any integer key with  $\phi$  results in a non-repeating fractional part. This prevents clustering of hash values and provides better **uniform distribution** of keys across the table. Uniform distribution reduces the number of collisions (which are the root cause of performance degradation in hash tables).

In comparison, in other methods (like division-based hashing like mod hashing), poor distribution can lead to a higher number of collisions, especially if the table size is not chosen carefully. Fibonacci hashing generally produces a more even spread of values, reducing clustering.

Also according to *OpenGenusIQ*, “Depending on the probability of collisions, if less, then it will have an average of  $O(1)$  We should also consider the size of the table. if the table is large then it will have less collisions, if small then there is a larger probability for collisions.”

## **B-Tree**

According to GeeksforGeeks, B-Trees can handle massive data with ease, B-Trees are notable for their ability to store many keys within a single node, which is why they are sometimes referred to as "large key" trees. With multiple keys in each node, B-Trees achieve a higher branching factor, resulting in a shorter tree height. This reduced height minimizes disk I/O operations, leading to quicker search and insertion processes (GeeksforGeeks).

B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always  $O(\log n)$ , regardless of the initial shape of the tree.

Algorithm	Time-complexity
Search	$O(\log n)$
Insert	$O(\log n)$
Delete	$O(\log n)$

## References

GeeksforGeeks, & GeeksforGeeks. (2023, December 28). Introduction of B-tree.

GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-of-b-tree-2/>

Roño, Cara. “Fibonacci Hashing.” *OpenGenus IQ: Learn Algorithms, DL, System Design*,

OpenGenus IQ: Learn Algorithms, DL, System Design, 22 Aug. 2022,

[iq.opengenus.org/fibonacci-hashing/](https://iq.opengenus.org/fibonacci-hashing/).

## Time Log

Person	Description	Time Took
Jane	Created the BST + main driver	1 hour 30 min
Jane	Wrote the Draft of the Experiment Report and the Time Table	2 hours
Jane	Wrote the description for new algorithm	30 min
Jane	Did research and outline the new algorithm	2 hours
Jane	Created Time Log	20 mins
Jane	Created outline of presentation	30 mins
Manuel	Created Linked List implementation + main driver.	1 hour
Manuel	Created the Hash implementation with Fibonacci method + main driver.	2 hour
Manuel	Created the AVL implementation + main driver	1 hour 30 mins
Manuel	Tested all previous implementations with the different data sets.	4 hours.
Manuel	Wrote an explanation for the fibonacci method of hashing.	30 minutes.
Harrison	Collected the data on AVL	30 minutes
Harrison	Setup avl ADT to report time for insert,search,delete functions	1 hour
Harrison	Setup list ADT to report time	45 min

	for each insert, search, delete	
Harrison	Collected data for list	30 min
Harrison	Setup hash table ADT to report time for insert,search,delete functions	25 min
Harrison	Collected data for hash table	25 min
Jane	Imported data from all ADT to graphs	30 min
Harrison	Added timers to BST insert, search and delete	45 min
Harrison	Collected data from BST timers	25 min
Manuel	Added fibonacci method section to Slides	10 min
Manuel	Analyzed the data and created a conclusion taking into consideration the data.	40 min.