An Empirical Study of Three Search Structures
James Murphy & Guolong Luo
Team stayHome
CSCI 323 - Analysis of Algorithms
Semester: Spring 2020

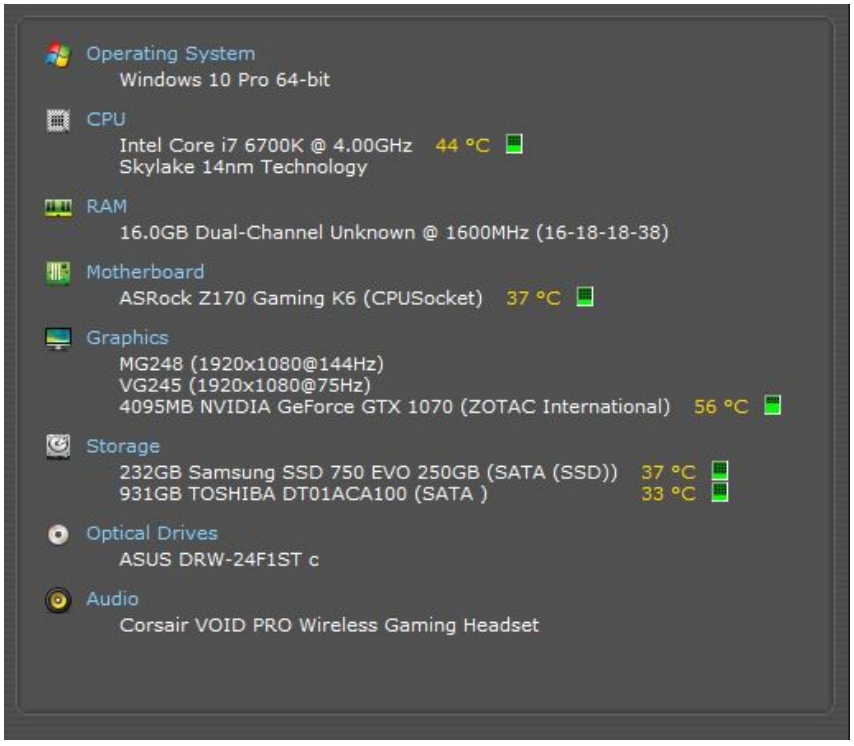
Introduction

A short paragraph describing what you sought to do in this project - list the three (or however many) algorithms you were comparing, what you compared them for, etc.

In this project I attempted to track how many operations (comparisons) each data structure took on various datasets to find out how it compares to their run times. The 3 data structures that I used were a Binary Search Tree, a Hash table with separate chaining and an AVL tree. The data sets I used are a randomly generated array of 1000000 integers, 10000 randomly generated words and dataset3.

Implementation Considerations

For this project we used Eclipse and wrote the program in Java. The machine that executed the code used in the tables was on Windows 10 and its specs are :



Code for the BST and AVL tree were modified. If a String was used, the value stored in a BST / AVL tree was the built in hashCode() function in Java, with the node class being edited to hold the data of the non-hashed String since hashCode() is a one way function. For the implementation of the third dataset, on a hashmap the key is either the hashcode of the String if that was selected or the Integer that was selected is both the key and the value.

The comparisons or operations tracked for each data structure is as follows :

Binary Search Tree : Each comparison of the key of the node that's trying to be inserted against the key of the node in the tree as the algorithm traverses. Recursive implementation. To store Strings in the BST, I had to use the String Builder class to turn the string into an ASCII Representation and save that as the key with a String variable inside the node.

Java's Implemented HashMap Class : Code found online for HashMaps in Java broke under various circumstances. Instead of using online code we decided to use the HashMap data structure already built into Java. The comparisons were tracked by a line of code that if the map already contains the key, keep track of that and increment a counter based on the amount of comparisons made.

AVL Tree : Like the BST, the comparisons are against the key of each node against another node until it finds its proper location to be inserted. However, there are more comparisons in the AVL tree than the BST due to the various rotations that the tree has to make to balance itself.

Data Considerations

Data Set 1 : A data set of 1000000 randomly generated integers using the java.util.Random library

Data Set 2 : A copy and pasted plain text data set of 10000 randomly generated words with duplicates allowed from this website: **https://www.randomlists.com/random-words?dup=true&qty=10000**

Data Set 3 : A dataset of n elements generated upon running the program. A randomly selected integer 0 or 1 is selected and if 0 is selected take the data from the 1000000 randomly generated integers array and assign that to index i. If 1 is selected, randomly select an index to take a string from the randomly generated string array and assign that to index i. It's a combination of both datasets already used.

Algorithm Overview + Tracking of Operations

Binary Search Tree

Overview :

A Binary Search Tree is a tree that is made up of a root, a left subtree and right subtree. The left subtree contains keys in it's node that are less than the key of the root node and the right subtree contains keys in its node greater than the key in the root node. Each element in the tree is a node and a node contains a key, sometimes data and 2 pointers to the next nodes in the tree : the left node and right node. If both pointer values are null, then the tree is either just a root or a leaf or both. Each subtree is also a binary search tree and must maintain the same rules to be a BST. The time complexity of access, search, insertion and deletion is in the average case O(log n). However, if the tree isn't balanced it can turn into a linked list which gives it a worst case time of O(n) for each of those operations.

Tracking of Operations :

Size (n)	Random Integers	Randomly Generated Words	Randomly Generated Words / Integers
10	19	27	27
100	654	583	621
1000	11162	9914	10655
10000	148793	117892	148400

Hash Table With Separate Chaining

Overview :

A Hash Table or Hashmap stores data in a <Key, Value> pair. If you want to retrieve an element from the table, you have to know the key. A hash table uses a hashing function that when run on the data stores it into a specific location. A collision occurs when 2 values are being placed into the same location by the hash function. It could be due to a bad hash function, or a duplicate value. There are different ways to handle this but in the implementation used in this project it is done through Separate Chaining. In that location there is a pointer to a linked list that stores any data / value that is put into the same location twice. Hash Tables are very efficient with a good Hash function, and in the best case with an evenly distributed hash function has a look up, deletion and insertion time of O(1). However, when you have a collision that changes into O(n).

Tracking of Operations :

Size (n)	Random Integers	Randomly Generated Words	Randomly Generated Words / Integers
10	0	0	0
100	0	1	0
1000	1	161	52
10000	40	7569	3109

AVL Tree

Overview :

An AVL tree is a modified Binary Search Tree. The general concept remains the same, however to avoid the worst case of a BST which is O(n) the tree self balances itself to make sure that never happens and maintains a worst case of O(logn). The way that the tree self balances itself is that it makes sure that the height of the left and right subtree is never more than 1 of each other. If it is, the tree then balances itself using these four different rotations : Left Left, Left Right, Right Right, Right Left. The tree selects a pivot and executes one of those rotations to maintain that height difference of a maximum of 1.

Tracking of Operations :

Size (n)	Random Integers	Randomly Generated Words	Randomly Generated Words / Integers
10	43	53	55
100	1265	1195	1283
1000	19203	17966	18907
10000	259471	192665	225340

Time Complexity
Time Look Up :
Binary Search Tree

Size (n)	Random Integers (Nanoseconds)	Randomly Generated Words (Nanoseconds)	Randomly Generated Words / Integers (Nanoseconds)
10	313	95	158
100	60756	160	185
1000	633	389	346
10000	771	451	566

Hashmap

Size (n)	Random Integers (Nanoseconds)	Randomly Generated Words (Nanoseconds)	Randomly Generated Words / Integers (Nanoseconds)
10	554	711	243
100	326	172	148
1000	307	299	131

10000	459	422	274

AVL Tree

Size (n)	Random Integers (Nanoseconds)	Randomly Generated Words (Nanoseconds)	Randomly Generated Words / Integers (Nanoseconds)
10	492	135	66
100	442	186	151
1000	658	327	288
10000	606	526	509

Conclusion / Analysis :

Operations :

Due to the nature of how a HashMap works, the only time there will be a collision is if the element already exists in the Map. It's expected that it by far had the least amount of operations since the program would need to randomly generate multiple of the same elements and words from the data sets chosen. An AVL Tree has much more operations than the BST due to the self balancing nature of the data structure. An AVL tree has a longer insertion time based on balancing and does so to avoid the downfall of turning into a linked list. However, the time added on in insertions is made up on lookups.

Time Analysis :

There were some weird anomalies within some executions where the first iteration of a hashmap search would have an abnormally long time compared to other look-ups. However, the results are pretty consistent and expected. In most cases the HashMap time is generally pretty consistent and doesn't have that much variance which is expected of an O(1) look up. A BST and AVL Tree both have a look up time of O(logn), and they both show the same behavior in terms of how many more nanoseconds it takes to look up the next iteration of n (10, 100, 1000, 10000). The AVL tree is a bit faster in almost all trials than the BST, but that should be expected since it's self balancing and provides a much better distribution of nodes than a BST does.

Sources :

Hashmaps
https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/

BST
https://www.geeksforgeeks.org/binary-search-tree-data-structure/

AVL Tree
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/