**INTRODUCTION**

Due to the nature of data organization between an AVL tree and a hash map, theoretically, the hash map should perform most operations faster than the AVL tree. This is because searching, inserting, and deleting in a hash table should be O(1) due to a hash table simply hashing a key and inserting it while an AVL tree must traverse down the tree until the proper location is reached. The same goes with searching too: an AVL must traverse the tree in O(log(n)) operations rather than the O(1) operations it takes for hash map to find a value.

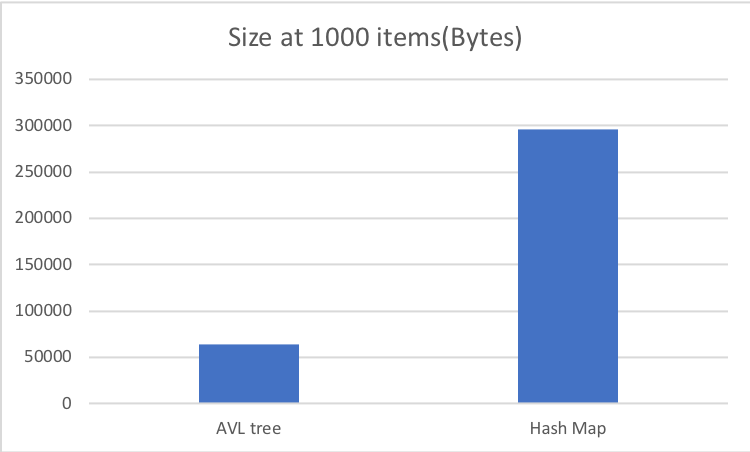
The most important part that makes a hash table O(1) is the hashing function. A proper hashing function minimizes collisions (when two keys return the same hash value, this requires chaining at the end of each index). Assuming that a hash map uses chaining when it encounters a collision, a hash table search has best case O(1) and worst case O(n). With an improper hashing function that implies worst-case-scenario, the AVL tree (which has best AND worst-case O(log(n))) will perform faster in every operation since the AVL tree only needs to traverse down in order while the hash table must iterate through every item in a single chain.

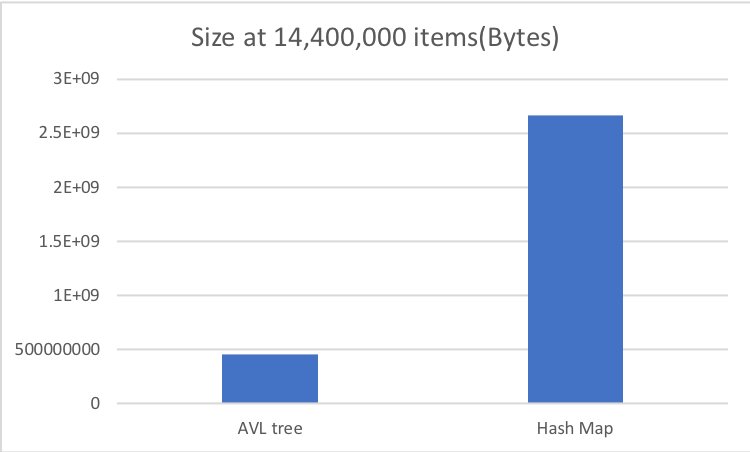
Although hash tables perform faster in most basic operations, there is one major advantage the AVL tree has. The AVL tree takes significantly less memory to maintain. The AVL tree is simply sized to the amount of nodes that have been inserted. Therefore, memory capacity is optimized and AVL trees should be preferred for crucial low memory applications. The hash table contains a table size designated by the programmer which will be resized once a high percentage of keys have been inserted. However, when the hash table needs to be resized, a temporary object multiple orders of magnitude larger needs to be created to replace the old table to minimize collisions; the risk of collisions from hash maps and resizing the table leads to copious memory usage. Even in my own testing, my hash map utilized all of my RAM, so I had to change my own resize function. However, the AVL tree will always hold the minimum amount of memory to operate efficiently.

For testing throughout this paper, I used three different word files. One containing 1000 unique words, one containing 400,000 unique words, and one containing 14,400,000 unique passwords. I repeated each experiment 10 times for each data structure and plotted the results into a box-and-whiskers plot. I wanted to use words to test the power of the STL hashing function, and if it truly provides enough randomization in a hash map to boast an O(1) search time.

**MEMORY**

Memory is where the AVL tree shines. Every node that the AVL tree occupies is contained with a data payload and pointers to other objects, compared to a hash map which is a large contiguous array that must be much larger than the total amount of keys to guarantee a proper search time of O(1). In my testing, I came to the same conclusion. The AVL tree occupied 5 times less space than the hash map in every data set.

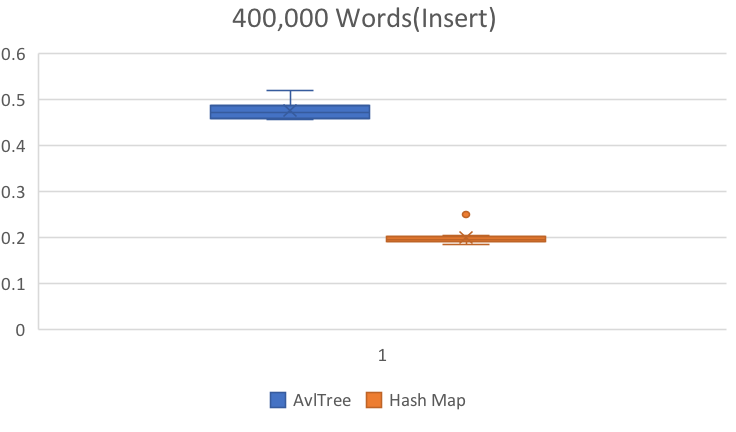
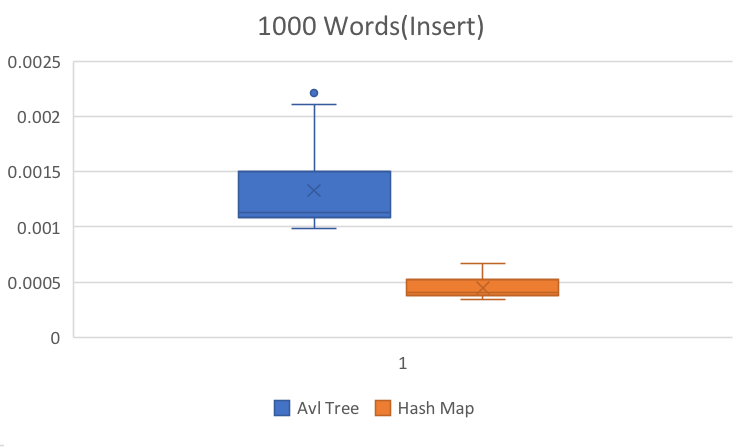


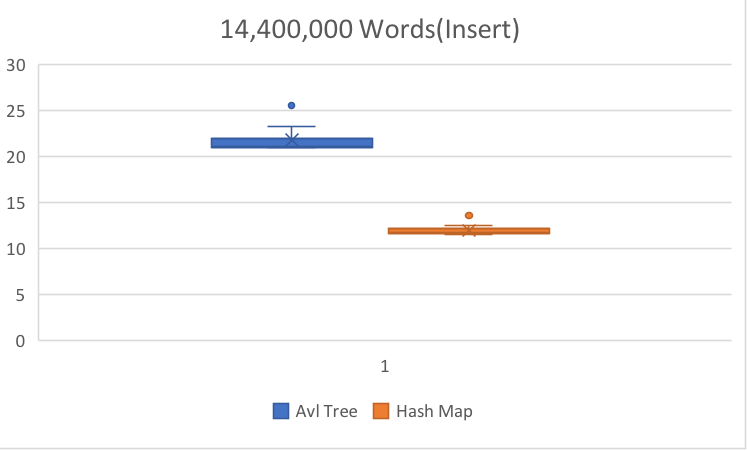


The AVL tree is consistently and significantly smaller than the hash map. In terms of memory, the AVL tree is by far the superior option. Also, resizing the hash map is a very resource heavy operation that copies the original hash map and makes another data container that is much larger than the original size. In the research I only accounted for the total structure size at the program exit, so the total amount of RAM the hash table could have occupied at once may even be around .2 times larger than what was shown in the graph. This can cause a very high probability for memory errors for extremely large data sets, which is why the AVL tree is superior if memory is the most important concern.

For large data sets, users should use the AVL tree. Although they are not as fast, computers may struggle with allocating enough RAM to perform efficiently with hash tables of a very large size. 3 Gigabytes of RAM vs. .5 GB of RAM is significant enough for large data sets, users should consider the AVL tree over the Hash Table.

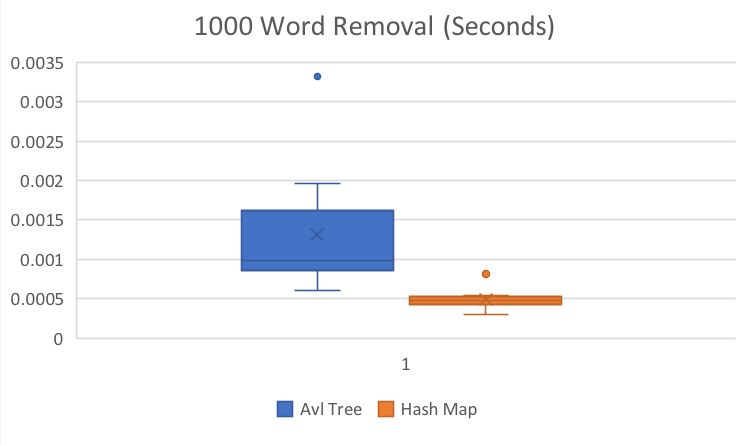
**INSERTION**

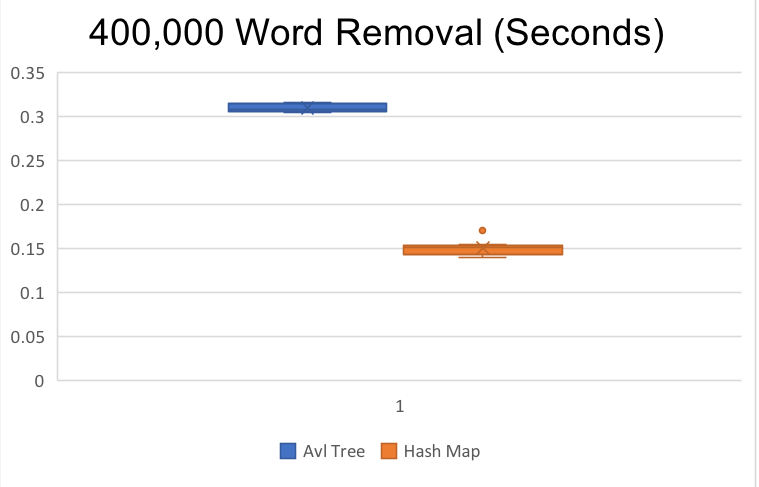
Throughout all the data sizes, the hash table nearly performed twice as fast as the AVL tree in every data set. Insertion is an extremely simple operation for a hash table since it only needs to hash the item and store it at the end of the chain rather than the AVL tree which needs to traverse until the correct available location is found, insert the node, and then possibly balance the tree as well. Therefore, since the tree is traversing in O(log(n)) operations, this implies a much slower time than the constant O(1) time that is the insertion method for a hash table. 

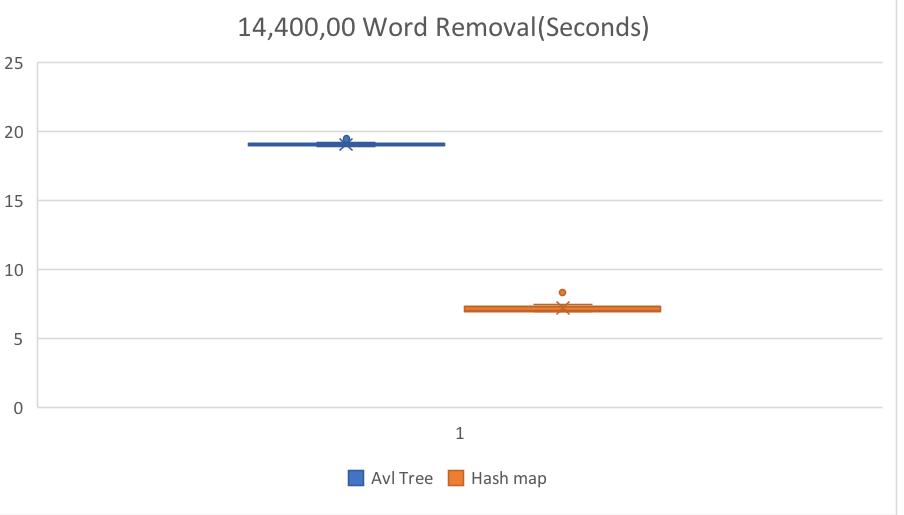


As shown in the data, the hash map consistently performs nearly twice the speed of the AVL tree. Throughout all three data tests, the slowest hash time never matches the fastest AVL time for insertion. I believe this is simply related to the O(log(n)) insertion time, and since the total hash table size does not impact performance of insertion, the hash map insertion function will always perform faster than the AVL insertion method no matter how large the data is.

**REMOVING**

The remove function has very different implications than insertion because not only does the container have to search for the key value, but once it is found it is removed, and the corresponding data structure is updated. Throughout testing, I have concluded that the hash map does indeed perform deleting faster than an AVL tree, and that is due to searching for the key value being O(1) for a hash table while it is still O(log(n)) for the AVL tree. 



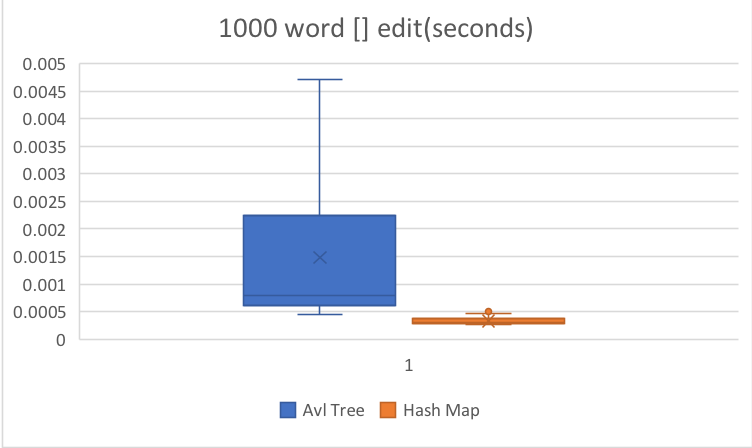


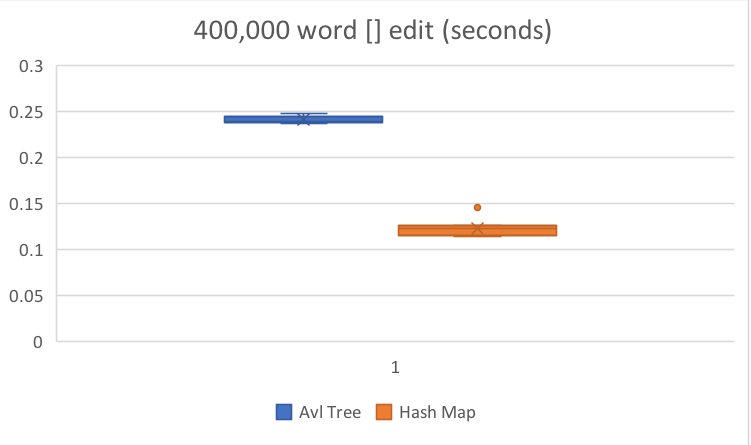
To test the remove functions of both items, I removed every single word that was inserted in the same order that they were originally read in. As more words are added, the total variance between each chart reduced significantly, so we have an accurate representation of how our item will perform in their respective data sets in real time, and a confident ratio between the two containers for real performance as well.

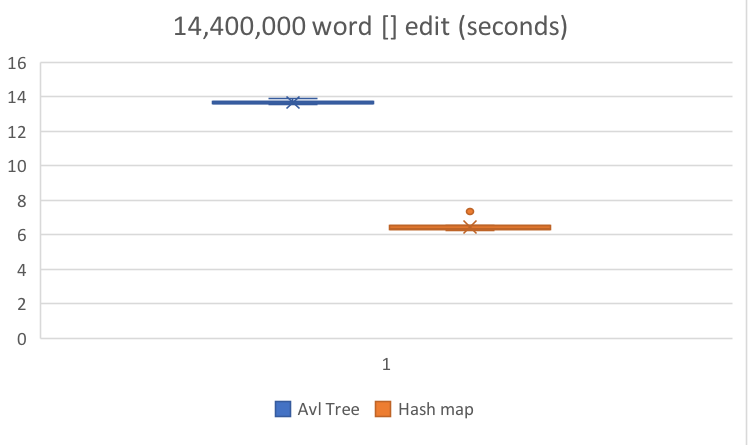
Once again, the hash map almost doubles the speed of the AVL tree in nearly every data size. The spread gets wider as the data set increases, and the hash map speed is much faster once again because of the quick search time. However, since chains in the hash map are handled using a list (which are utilized for their effectives with removing/inserting within the container) remove times within the each chain are also only O(1). If a programmer were to implement each chain in the HashMap with its own set of contiguous memory, the process would take much longer since removals in contiguous memory require the entire memory block to be replaced. Since the hash map is implemented using a list, this will always allow O(1) removal time for the hash map as a whole which will always perform better than an AVL tree at O(log(n)) regardless of the data size.

**SEARCH AND EDIT USING [ ]**

Now that we have tested the removal of nodes it is time to test the pure search for every key value. In this test, I inserted the respective data sets into the proper containers, and then I went back and edited every item using a recursive search algorithm for AVL trees and a hashing search for the hash map. Results show that, not surprisingly, the hash map outperforms the AVL tree in pure searching.





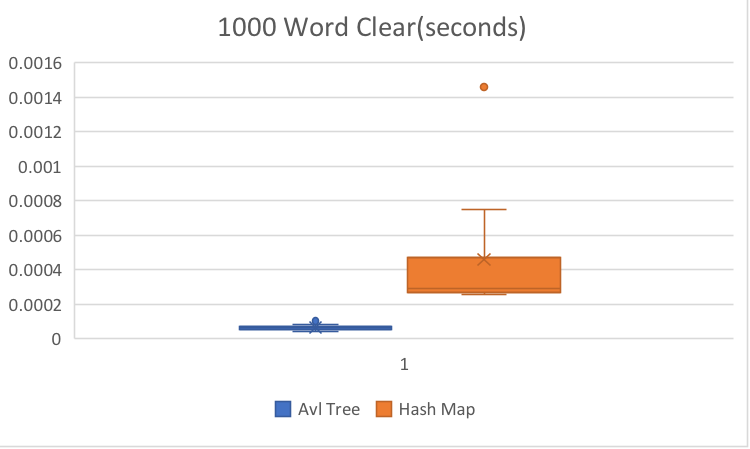


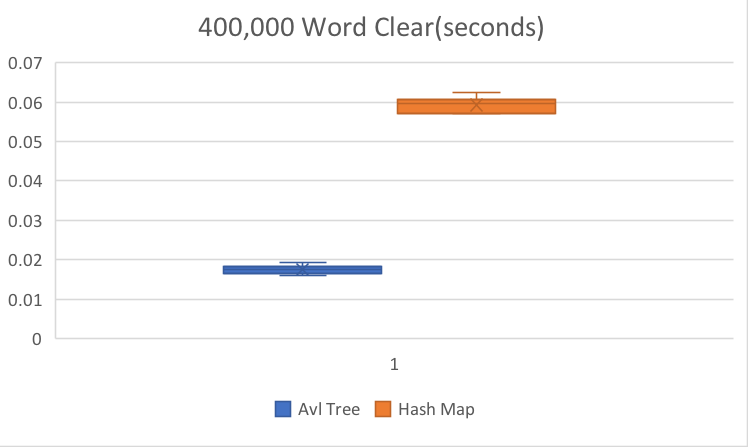
As the data sets increase, we once again see a correlation as the spread increases between search times of the two containers. This is purely because the number of recursive calls is directly related to the size of the tree which leads to slower runtime performance while a hash map at the proper size still only allows O(1) complexity, even with over 14 million keys. However, when the data sets are small, the median search times were similar, so for approximately one thousand keys, if a user were to only search, either container would be appropriate. Although, the Hash Map should always be used if the programmer plans to have over one hundred thousand unique keys.

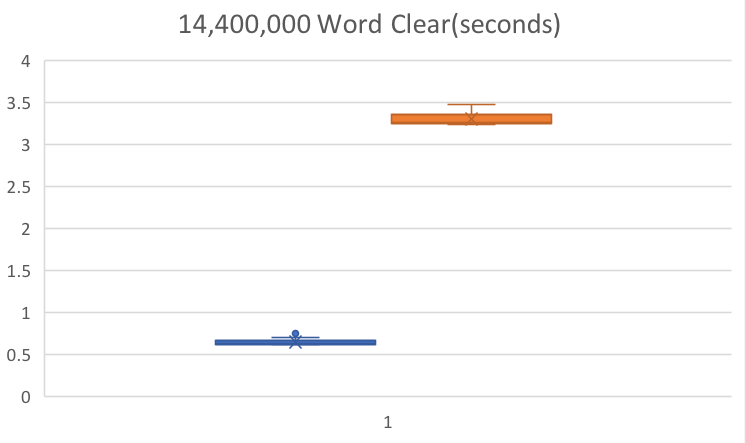
Since the search function is significantly faster in the hash map, I believe this is the reason that the remove method is much faster as well. If there was a faster non-recursive way to explore a tree, it would be the superior container in both memory and performance. However, since searching is significantly faster in hash tables, it causes a domino effect with its other methods, which cause the hash map to perform at a faster speed than the AVL tree in nearly every operation.

**CLEAR**

The clear method is the only method to favor the AVL tree over the hash map. This is because of the sheer size of the hash map vs. the AVL tree, so the tree has fewer locations to search than the hash map.







We see here that as the data set increases, there is a huge difference between the speed of the clear method between the AVL tree and the hash map. To clear a hash map, one must iterate over the entire container and clear every chain that exists throughout the container while an AVL tree simply is O(n) since it just iterates over the entire tree once and delete every node that is called from the root. If we were comparing the two data structures solely on their clear method, then the AVL tree will be faster for every data size.

Although the clear time for AVL tree is much faster than a hash map, it is not often that a clear method gets called for a hash map, and even then, the performance impact from a hash map in other operations still has a significant improvement over an AVL tree to the point where the clear method will be of limited impact. However, the only way to destroy a tree is to implement a similar destructor method that performs the same as clear while a hash map can just delete the allocated memory when the container goes out of scope which will be much faster than the AVL tree, but if we were just going to clear the containers and have them still exist, then the AVL tree will perform much faster.

**CONCLUSION**

Both the AVL tree and the hash table have their own benefits. To judge which one is better depends on what the user sees as more important. If memory is more important, the AVL tree is significantly more efficient than the hash table. As is shown with the hash map nearly occupying 2.7 GB of Ram at 14.4 million keys while the AVL tree only occupied .456 GB of RAM. I think this is partly because of my decision to make the table 5 times larger once we reach 75% of the table size, which explains why the AVL tree is around 5 times smaller than the hash map. However the table must always be large to minimize collisions to ensure O(1) time for many of the hash map functions. Therefore, the AVL tree will always be significantly better than the hash map in terms of memory.

In terms of performance, this is where the hash map shines. It is nearly twice as fast in every crucial container function than the AVL tree. If a user were to only analyze performance, then the hash map will always be the best option regardless of size. The constant O(1) time for many different operations is much faster than the tree traversal of O(log(n)) complexity. Once many different keys are placed into both data structures, it is obvious how the hash map will continue to outperform as the size increases.

If we were to weigh the importance of performance and memory equally, then the hash map would be the superior choice for everything except for the most extreme sizes. Once the amount of keys reach about 15 million, it may be better to consider to use an AVL tree instead because of the sheer amount of RAM a hash map occupies may lead to bad alloc runtime errors. However, in terms of which one is faster, the hash map is confidently the king.