CSCI 220 - Data Structures and Algorithms:
Project 4: Array Based Tries Analysis

Patrick Maes

November 30, 2022

Part A

First, analyze the complexity of inserting into a hash table while taking into account string length and elements stored. The worst and average cases will be most useful here, as the best case would likely just be $O(m)$, where m is the string length. If we had a good hash function, the worst case would be in the event of a collision. This complexity would also be dependant on the chain length, which will be l in my case. In the worst case, the time complexity would likely be m to hash, and $l * m$ for the comparison of the strings in the chain, $O(m(l+1))$. However, if we assume a good hash function that distributes the elements evenly, the average case would be O(m). We can reference the hash table analysis to see this, where the average chain size of a good hash is 1, meaning we would need no string comparisons in most cases. For search I would conclude the same, where the complexity is $O(m(l+1))$, since we need to hash the key and compare it to at least one element.

As for binary search trees, I think they would be the worst out of the three data structures. Insertion and search in binary search trees has a worst case complexity of $O(log_2(n))$. This is assuming that the tree is balanced, meaning we can guarantee the height is dependant on the number of elements. However, if we are comparing strings at each node, comparisons will take O(m) time, where m is still the string length. This means that the overall complexity of inserting and searching in a binary search tree would be $O(mlog_2(n))$. At first, this looks a little better than the hash table. However, it is necessary to recognize that the hash table will probably have max bin sizes of 4 or 5 in the worst case. On the other hand as we add more than 2^5 elements, $O(mlog_2(n))$ is greater than $O(m(l+1))$.

Last, for the trie, the worst case complexity for insertion and search is O(m), again where m is the string length. This is because we only have to iterate over each letter one time at most, and our comparisons are made as we traverse the trie. For insertion, we only check if the letter exists, and if not we insert it in the trie. Search is similar, except we stop if the letter is not contained. This makes the trie the best data structure, since we never have to do a string comparison.

From my hypothesises, I would assume that the trie would be the best data structure. Search always takes O(m) time, which is independent of the number of elements inserted into the trie. However, a hash table with a good hash function may be close to the same efficiency. One of the biggest advantages of the trie is finding elements that do not exist. In this case, tries can give sub O(m) performance since it returns false when a letter is not found. On the other hand, hash tables and binary trees would still need to compare entire strings or hash the string, which both take O(m) time. However since we are only testing contains for words that do exist, the hash table and trie may be closer in performance.

Below are the timings and tables for both the insertion and contains functions:

Insertion	Trie	BinaryTree	HashTable
Sorted	0.208	1.48	0.18
Unsorted	0.265	0.935	0.15

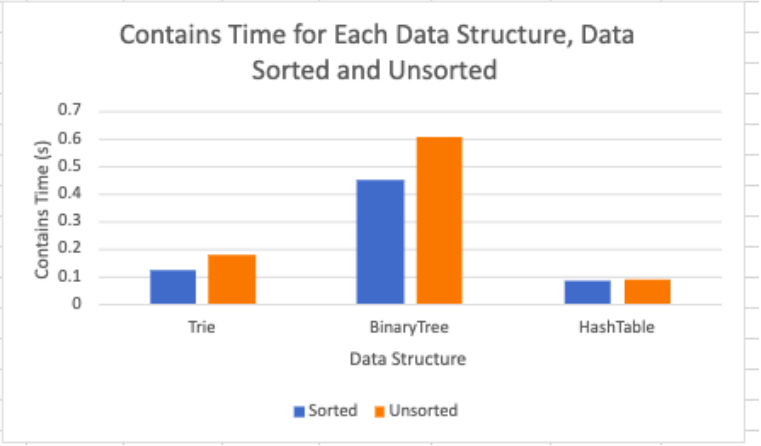

Figure 1: Insertion time of entire dictionary contents into each data structure in seconds.

Contains	Trie	BinaryTree	HashTable
Sorted	0.124	0.451	0.087
Unsorted	0.18	0.609	0.092

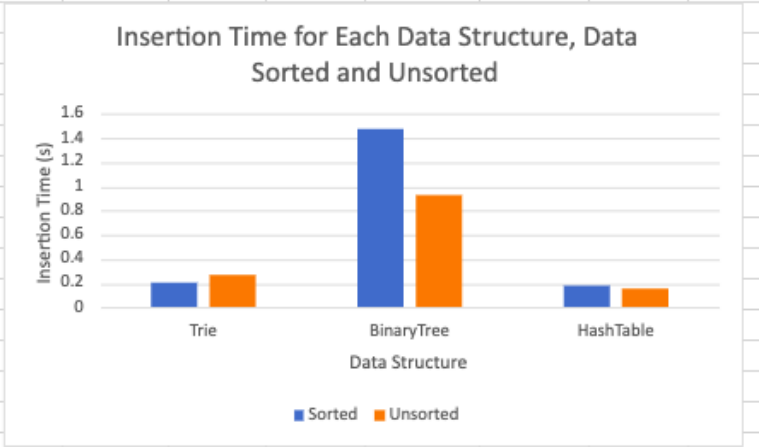

Figure 2: Find time of entire dictionary contents for each data structure in seconds.

The results of my timings slightly contradicted my hypothesis that the trie would be the best data structure for the string operations. In all cases the hash table seemed to be slightly faster than the trie, but almost always by less than one tenth of a second. I have a few guesses to why this is occurring, but I am still not completely sure.

On first thought, I am comparing my trie data structure with the c++ hash table. The hash table is almost definitely more refined in its implementation than my trie is. There are probably ways to increase string comparison best case complexity, for example by comparing the length of the strings before any characters. In the case of two strings with different lengths, its probably not neccesary to even compare characters.
My other hypothesis on why the trie performs slightly worse would be due to the way I tested the two data structures. Tries have very efficient search time on strings that do not exist in the data structure, but we are only searching strings that do exist. This means that the best case is always O(m), where m is the string length. In the hash table, we still need to hash every character, but the average search would also be O(m).

I would still expect the trie to be slightly better, but this at least explains why they are close in performance.

Last, I also think that cache hit rate could also play a factor since the hash table is stored as a vector or array which has constant access to the hashed values. On the other hand, searching the trie is much more difficult since it is dynamically allocated and the pointers lie in different places in memory. This could affect the speed of some of the operations making the slight advantage that the hashtable appears to have.

Part B

In order to test the two data structures memory wise, we just need to know how many keys are in the hashtable, and the number of nodes in the trie. For simplicity, the hashtable will be analyzed first since we need multiple different cases for the trie.

First I found the size of the outside array holding the linked list pointers. To make the hash table as small as possible, I computed the load factor as $.75(x) = 172,823$. Then I simply divided the number of elements (right hand side), by the load factor to find the minimum size of the table which holds 172,823 keys. Each space in the array holds an 8 byte pointer to a linked list:

$$.75(x) = 172823 \tag{1}$$

$$x = 230,431 \tag{2}$$

$$230,431 * 8 = 1,843,448 \tag{3}$$

The pointers to the linked lists themselves take up 1,843,448 bytes. We also need 172,823 linked list nodes, or in this case spaces in a dynamically allocated array. Each key/node will be 16 bytes.

$$172,823 * 16 = 2,765,168 \tag{4}$$

The number of bytes for chain nodes is 2,765,168. Now that we have these two sizes in bytes, we can add them together with the given size of all the stored strings.

$$2,765,168 + 1,843,448 + 1,743,363 = 6,351,979 \tag{5}$$

Convert to MB:

$$\frac{6,351,979}{2^{20}} = 6.0577 \tag{6}$$

The size of the hash table if it was as small as possible while maintaining a load factor of .75 is 6.0577 MB. Now compare this with different types of trie implementations to see the differences between the data structures. When I counted all the nodes in my trie, I found 387,889 nodes. If all of these nodes used 216 bytes each, then the trie would be 79.9 MB. This is considerably bad compared to the hash table, but it is probably due to the number of array elements that hold nullptr.
If I was to only allocate an array if the node is not a leaf, then each node would be 224 bytes, and leaf nodes are 16 bytes. I created a function similar to counting the nodes, but only increment if the node has no children. I found there to be 111,508 leaf nodes. This means that there are $387,889 - 111,508 = 276,381$ non-leaf nodes. Finally use these to find the total memory if we only allocated an array when needed. Total memory:

$$276,381(224) + 111,508(16) = 63,693,472$$
$$\frac{63,693,472}{2^{20}} = 60.74$$

Removing the arrays from the leaf nodes cuts the memory down to 60.74 MB. This is still not as good as the hash table memory efficiency.

Part C

In order to find the space complexity of the compressed trie, I first wrote a function to count the number of compressed trie nodes needed. I used the psudocode to count nodes, but only if they had more than 2 children, or no children. This is because any node with one child will be merged with a node in this case. I found that there were 173,785 compressed trie nodes, and 111,508 leaf nodes in the trie. This means that there are $173,785 - 111,508 = 62,277$ nodes that are not leaf nodes. For each leaf node, I chose not to allocate an array of pointers, but simply store the string. For the interior nodes, we need both the array of pointers and the space for the string.

Each character in a string takes up 1 byte, and we now have 8 extra bytes per node from the Boolean flag we removed. For simplicity, I will say each node has a fixed 8 byte character array. Therefore, each internal node will still take $26(8) + 8 = 216$ bytes. The leaf nodes will take 16 as well, since they hold an array pointer and the string itself (both 8 bytes). If we use these as the measurements for the compressed trie, the new total memory is:

$$\frac{(62,277(216) + 111,508(16))}{2^{20}} = 28.3 \tag{7}$$

The compressed trie under these assumptions takes 28.3 MB compared to more than 60 before. I would say that the added complexity of the compressed trie is worth it in our case. Surprisingly, the hash table is more space efficient. However, the trie has many advantages that the hashtable does not, such as sub O(m) search for keys not contained and features for prefix extention and search. The memory differences likely come from the large number of child array elements that are not needed. I'm sure that a map with only children that exist could increase the memory and time complexity of any of the trie implementations.