Autocomplete Lookup Report

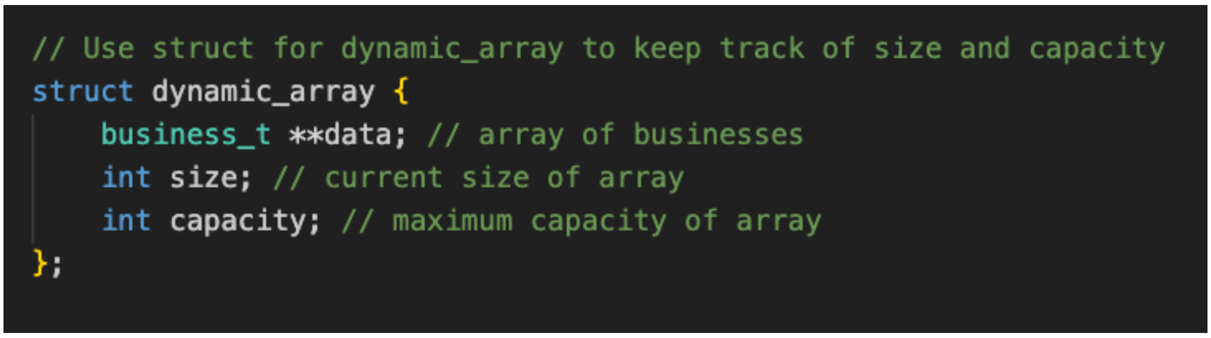
Introduction:

The two stages of the autocomplete lookup assignment involved reading data from a csv file and storing the information into one of two data structures: the sorted array and radix tree. This report hypothesises that whilst the search algorithms of both data structures run with similar time complexities, the insertion of data into the radix tree is more efficient than into the sorted array. From the outset, a comprehensive analysis of the two distinct data structures with a comparison of their performance will be done. Initially, the two stages will be examined individually, discussing their implementation methods and performance. Thereafter, both stages will be compared against each other discussing the pros and cons of both data structures.

Stage 2:

Method:

The sorted array implementation was generally straightforward, the data was read from the CSV file, and pointers to the business structs were kept order. Once the data was read from the file, a binary search was used to find the correct index for it to be placed in. As shown in figure 1, a struct was used to store keep track of the size and maximum capacity of the array, to know when to dynamically allocate more memory.



```c
// Use struct for dynamic_array to keep track of size and capacity
struct dynamic_array {
    business_t **data; // array of businesses
    int size; // current size of array
    int capacity; // maximum capacity of array
};
```

Figure 1: dynamic array struct

The find and traverse function, also used a binary search to find the first match to the query. Following this the algorithm would linearly search to the left and right for any additional matches.

Analysis and Comparison to Theory:

In the analysis section, rather than finding the time taken for each algorithm to run, which may differ depending on hardware, the number of comparisons will be weighed up against the expected number of comparisons for a given input size. To get the values for the average number of comparisons two new datasets were made of size 200 and 500 to further highlight the slowing growth of comparisons when size increases which occurs with O(log n) complexity.

As discussed above, using a sorted array required a binary search to find the correct index to place the new values. With this approach the expected behaviour (Big O) for insertion would be O(log n), as for each new value a binary search would have to be done which is expected to have O(log n) comparisons. Figure 2 shows the average number of comparisons for the binary search required for the insertion into the sorted array and compared it to the expected number of comparisons for each array size. As the searching would generally require less comparisons whilst the array is smaller, the number of comparisons should be less than the expected upper bound of O(log n). However, insertion through this method would require the moving of previous values of the array when a new value is added. This would result in O(n) operations to move all the values, which results is O(n log n) time to insert all values. If a different method was used such as sorting the array after all the insertions, it would likely have a similar time complexity of quasilinear O(n log n) time, if a sorting algorithm such as merge sort is used.

Size (n)	**Average Number of Comparisons**	**Expected Number of Comparisons O(log n)**
2	1.00	1.00
20	3.05	4.32

100	5.37	6.64
200	6.29	7.64
500	7.51	8.97
1000	8.36	9.97

Figure 2: Table of Average Number Comparisons for Insertion into sorted array

Figure 3 visualises the logarithmic growth in comparisons required when inserting values into a sorted array. The number of comparisons that occurred for each dataset remained under the theoretical upper bound of O(log n) for each input. This implies that to for an array of size n, it would require O(n log n) time for the insertion all the values. However, every time a new value is added to the array, the current values will have to be moved, resulting in extra run time. This is likely the operation that takes the most time when adding values to the array.

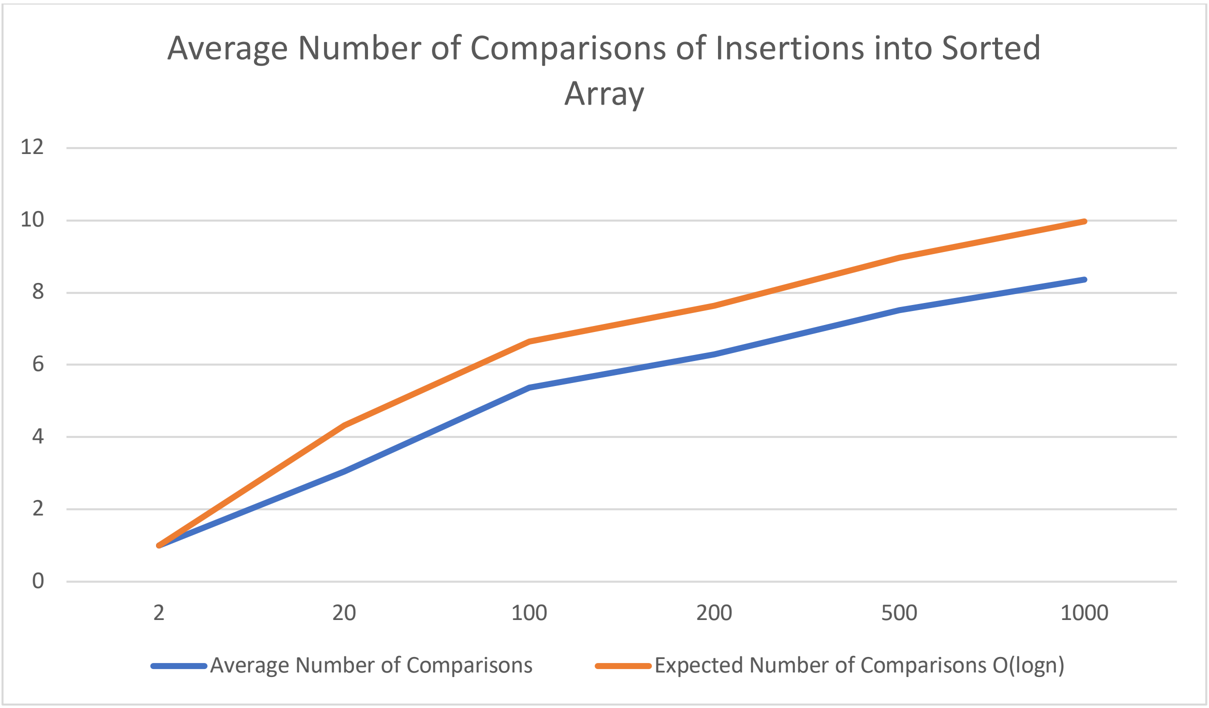


Figure 3: Graph of Average Number of Comparisons for insertions into Sorted Array

In a similar light, the find and traverse function also used a binary search for each input, but due to the linear traversal that occurs once a value is found, at least two more comparisons would occur for each value. In figure 4 the average number of comparisons is compared to the expected amount. With the smaller input sizes the average number of comparisons exceeded the expected number, due to the minimum of two equality checks to the left and right that occur once a value is found. This would imply that the complexity of the algorithm would be O(log n + k), with k being the number of equal values that is being searched for. Similarly, in figure 5, the graphical representation of the average number of comparisons, the growth of the average number of comparisons resembles the logarithmic growth of the expected number. Thus, implying the logarithmic time complexity of the find and traverse function.

As the size of the input grows, the effect of k on the number of comparisons decreases. With respect to theory, as extraneous information such as slower growing terms tend to be ignored in the final complexity, the extra comparisons from the traverse part of the function can be ignored. Therefore, it can be said that the complexity of the find and traverse function would be O(log n).

Size (n)	**Average Number of Comparisons**	**Expected Number of Comparisons O(logn)**
2	2.50	1.00
20	5.60	4.32
100	7.63	6.64
200	8.70	7.64
500	9.90	8.97
1000	11.01	9.97

Figure 4: Table of Average Number of Comparisons for Find and Traverse Function

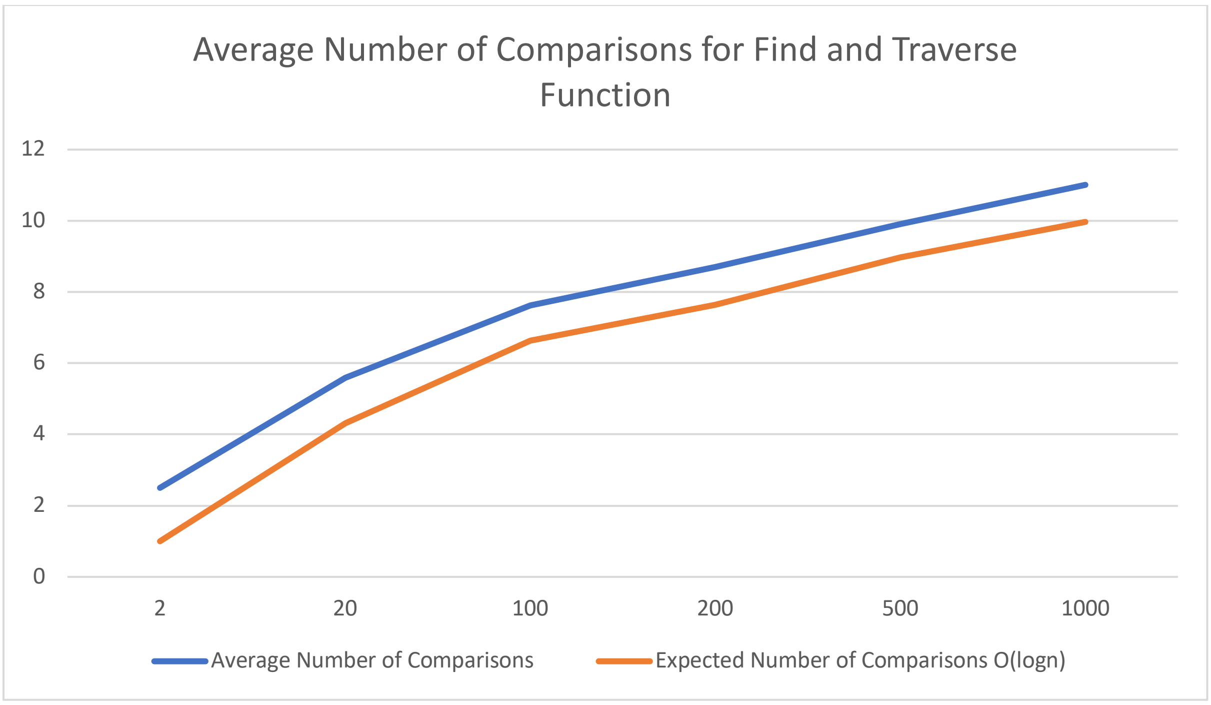


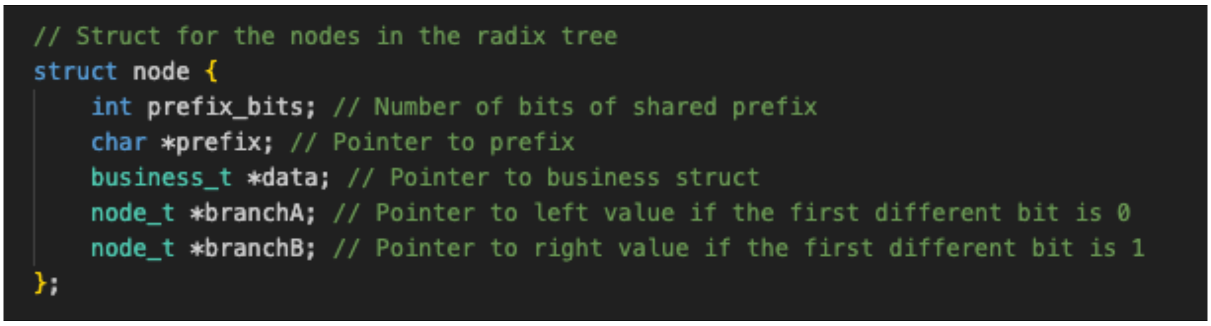
Figure 5: Graph of Comparisons for Find and Traverse Function

Overall, stage 2 involved the implementation of a simple yet generally, time efficient algorithm. The use of binary search for insertion resulted in the number of comparisons to have a theoretical upper bound of O(log n) with the linear traversal of the search function causing the number of comparisons to be slightly larger whilst still maintaining an O(log n) complexity as slower growing terms are generally ignored.

Stage 3:

Method:

The implementation of the radix tree data structure was less straightforward as compared to stage 2. My solution did not get the correct outputs for the larger data samples. The issue involved the reordering of the radix tree nodes when a value with a smaller number of prefix bits was found. However, the data was still successfully read to the radix tree when smaller data inputs were used.



```
// Struct for the nodes in the radix tree
struct node {
    int prefix_bits; // Number of bits of shared prefix
    char *prefix; // Pointer to prefix
    business_t *data; // Pointer to business struct
    node_t *branchA; // Pointer to left value if the first different bit is 0
    node_t *branchB; // Pointer to right value if the first different bit is 1
};
```

Figure 6: Node struct

When inserting the values into the radix tree, the binary representation of each character was compared to the existing nodes within the tree. When a bit differed a new branch would be made, pointing left or right depending on what the value of the different bit is. Searching for values was slightly more straightforward involving reading through the prefixes in the nodes and moving to the left or right branches depending on the value of the differing bits.

Analysis and Comparison to Theory:

In this section, the character comparisons and bit comparisons will be analysed. Since the solution did not produce correct outputs all the time, a script was made to read through the expected test outputs to find the number of char and bit comparisons for each input size. With working code, the number of comparisons during the insertion process would have been calculated as well as exploring the depth of the tree given certain inputs. Without a working solution, the comparisons in the searching process will be discussed instead, with hypothesis regarding the insertion being made.

Unbalanced binary search trees have a worst-case search time complexity of O(n), however the average case would generally be O(h), with h being the height of the search tree. As all the values could theoretically be on one branch, the worst case would therefore be O(n) as the height of the tree would be n. However, on average a search tree would have a time complexity of O(log n). To calculate the average number of comparisons in figure 7, the number of bit comparisons was divided by the number of total bits. When comparing the comparisons numbers of the radix tree to the sorted array, the radix tree has far fewer comparisons in all domains. The number of comparisons being far less than the expected even for the best case of O(log n).

	Average Sorted Array Comparisons			**Average Radix Tree Comparisons**		
Size (n)	Bit	Character	String	Bit	Character	String
1	1	1	1	0.93	0.93	1
2	1.08	1.08	2.5	0.94	1.00	1
20	1.48	1.48	5.6	0.94	1.17	1
100	1.73	1.73	7.63	0.94	1.32	1
1000	2.56	2.56	11.01	0.95	1.62	1

Figure 7: Table of Comparisons of Sorted Array and Radix Tree for Bit, Character, and String

The number of string comparisons remained 1 for all radix tree searches, as the only string comparison occurred when a node containing data was reached. Similarly, by comparing bits instead, the radix tree reduced the number of both bit comparison and character comparison, as a different bit would prove the inequality, rather than different strings. This therefore allows the comparisons of the radix tree to be less than the sorted array.

With this is mind, it may not seem like the radix tree is a more efficient solution when compared to the dynamic sorted array which took O(log n) time on average, however the insertion into the tree is hypothesised to be much more efficient. Without more information such as the actual time taken for execution and the depth of the tree, a proper time complexity analysis could be made. However, the time complexity of insertions into the radix tree is likely to perform better than the sorted array. This is due to the radix tree not requiring moving previous values when new values are added, which is done when inserting into the sorted array. Further research would seek to confirm this hypothesis.

Comparing Both Stages:

Whilst both stages were relatively efficient in terms of searching and insertion, both data structures have their own pros and cons. The implementation of the sorted array is relatively simple, with data being stored dynamically and binary search used for efficient retrieval and proper sorted insertion. On the other hand, the radix tree required comparing binary representations of characters and branching accordingly, adding implementation complexity. Insertion into the sorted array requires O(n log n) time due, with dynamically resizing the array introduces extra running costs. Insertion into the radix tree is expected to be more efficient than the sorted array, given the number of comparisons in the radix tree to be less than the sorted array, as demonstrated in figure 8. In addition, the insertion into a radix tree does not require moving previous when adding new values, resulting in less run time.

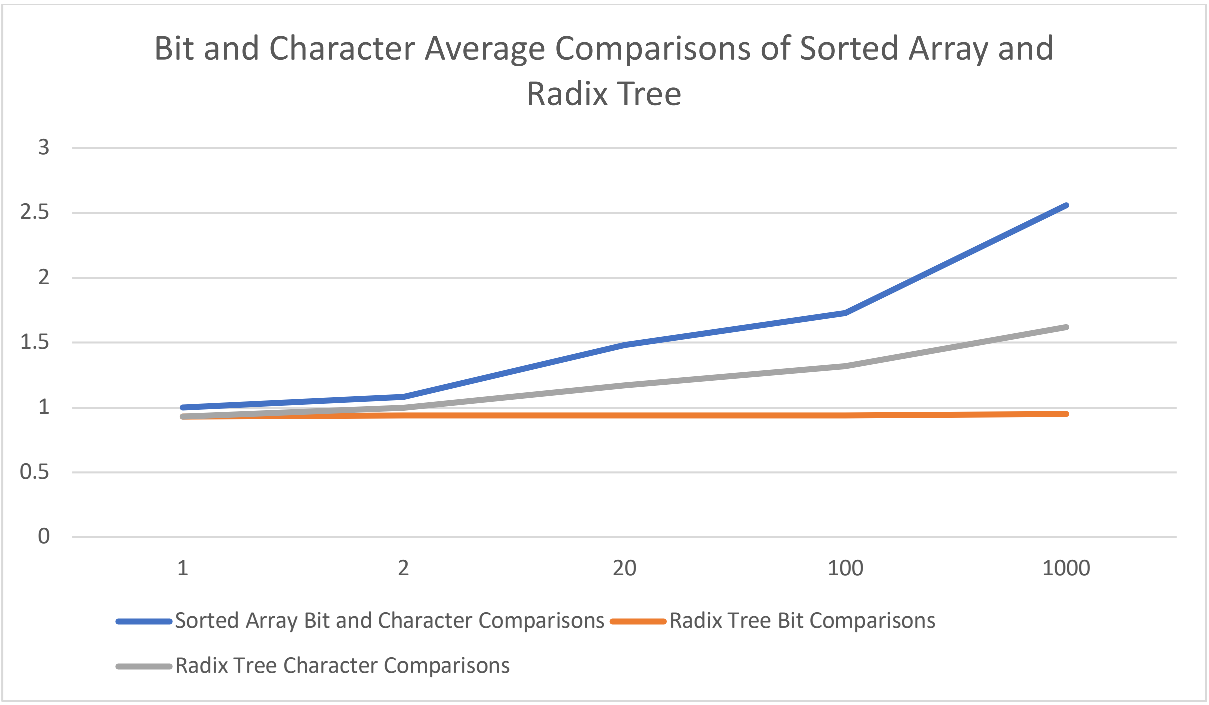


Figure 8: Graph of Bit and Character Comparisons in Sorted Array and Radix Tree

In a similar light, the sorted array search is also O(log n) due to the binary search, with the radix tree hypothesised to be slightly faster. However, without precise execution time data, it is challenging to determine the exact time complexity. Comparing the time takeqn for both algorithms to run would be more representative of their true time complexity, and with a working solution, the depth of the radix tree would be found to have a better understanding of its time complexity. However, as that is not feasible, the number of comparisons were used to infer the efficiency of both data structures. The space efficiency of both data structures was not able to be properly compared, but they are expected to use a similar amount of memory, especially as the data they hold are mostly pointers to dynamically allocated memory.

Conclusion:

In this assignment, two distinct data structures for an autocomplete lookup were used: the sorted array and the radix tree. Each stage was analysed individually, discussing their implementation and an analysis of their performance. Subsequently, both stages were compared highlighting the strengths and weaknesses of each data structure. The assignment demonstrated that both data structures have their merit. With the sorted array exhibiting relatively efficient performance with O(log n) complexity search, and quasilinear O(n log n) operations with insertion. Conversely, insertion into radix tree is likely far faster, with searching also having less comparisons than the sorted array, suggesting a performance of O(h). Further research and experimentation involving the analysis of execution time and tree depths would be necessary to confirm the exact time complexity of the radix tree. Overall, comparing both data structures showed that the choice between them depends on specific application requirements.