COMP20003 Assignment 2 Report

Acknowledgement of AI Use for Report

I utilized ChatGPT-3.5 to refine my choice of words and enhance sentence structure, with the aim of presenting the findings and analysis in a more coherent manner. I have retained the complete chat history within my ChatGPT account and am readily available to provide it as a point of reference if there are any concerns about the authenticity of this report.

I affirm, without reservation, that the ideas, findings, and analysis presented in this report are solely my own original work unless otherwise indicated by appropriate references. I acknowledge the gravity of plagiarism, where one wrongfully appropriates the ideas of others as their own, and I am fully prepared to accept the consequences for such misconduct.

Abstract

This assignment revolves around the implementation of autocomplete lookup using a dictionary data structure, divided into three distinct stages. The initial stage involves creating a program to read datasets from CSV files, which was successfully completed in assignment 1. The second stage focuses on implementing the dictionary utilizing a dynamic array, while the third stage explores the utilization of a radix tree with radix 2. This report will begin by commenting on the underlying logic of my implementation. Subsequently, it will delve into a comprehensive analysis of the performance exhibited by each data structure, encompassing their respective insertion and search algorithms. Ultimately, the report will draw conclusions regarding the efficiency of these structures concerning the provided test cases.

Implementation

Brief Summary

Attempts have been made to implement both but unfortunately I have not been able to finish stage 3 implementation in time. Therefore in this report I will not be able to analyze the actual performance of the radix tree, but I will give a theoretical prediction of its performance.

Stage 2 - Dynamic Array

Implementation Logic

The overall logic of my implementation can be broken down into the following steps:

1. Initiate an dynamic array of characters that will be used to store the trading names of the cafes read in. Whilst keeping the link list implementation from stage 1 for searching later.
2. I opted to employ the built-in "qsort()" function from the C standard library to sort the dynamic array. This decision stems from several considerations. While I possess a solid understanding of quicksort's underlying logic, my ability to select more sophisticated pivots than the "median of three," is currently limited. This limitation could risk a substantial degradation in performance, particularly when dealing with large and unfavorable datasets, where there is a higher likelihood of selecting extreme values as pivots. Furthermore, upon reviewing the implementation of the built-in "qsort()" function, I noted its adaptive nature. Specifically, it transitions to insertion sort when the data range has been reduced to a threshold where insertion sort's efficacy shines, particularly for small datasets. Given this optimization, it becomes apparent that utilizing the built-in "qsort()" function is a prudent choice for achieving superior sorting performance.

3. Implement the find_and_traverse() function by first using binary search to locate an initial match, and then use linear search around the first match to locate all the matching data. The process will also keep track of the number of character and string comparisons executed in the process of finding matches.

4. Search for the cafes with the trading names in the link list from stage one and print the result.

The number of comparisons will be analyzed in three units (strings, bytes and bits) to gain the most comprehensive insight into the performance possible. I have opted to compare the total number of comparisons rather than the average

Theoretical Prediction

Before making predictions based on theory, it is crucial to consider certain assumptions that can significantly impact the results. For stage 2, since the autocomplete's performance relies on the number of comparisons needed during a search, it is reasonable to exclude the influence of data preprocessing, such as sorting, when conducting the analysis. Another assumption is the RAM model of computation, which is employed when assessing search performance. In this model, basic operations, like arithmetic and comparisons, are assumed to take one time step.

Theoretical predictions suggest that the number of comparisons will increase with the input size following an O(n) order, where n represents the number of strings. This occurs because a linear search is required to find the first match, resulting in a time complexity of n + log(n). However, for sufficiently large datasets, the linear term (n) will dominate, resulting in an overall time complexity of O(n).

Nevertheless, it's worth noting that the maximum number of data tested is only 1000, which is not large enough to safely disregard the significance of the log(n) term. Therefore, for this assignment, predictions will consider both comparisons made in binary search and linear search. Meanwhile, the number of linear searches required will only be as high as the number of

items sharing the same prefix as the key. This number is highly unlikely to be as large as the dataset, so a reasonable number of such duplicates will be selected as the coefficient before n.

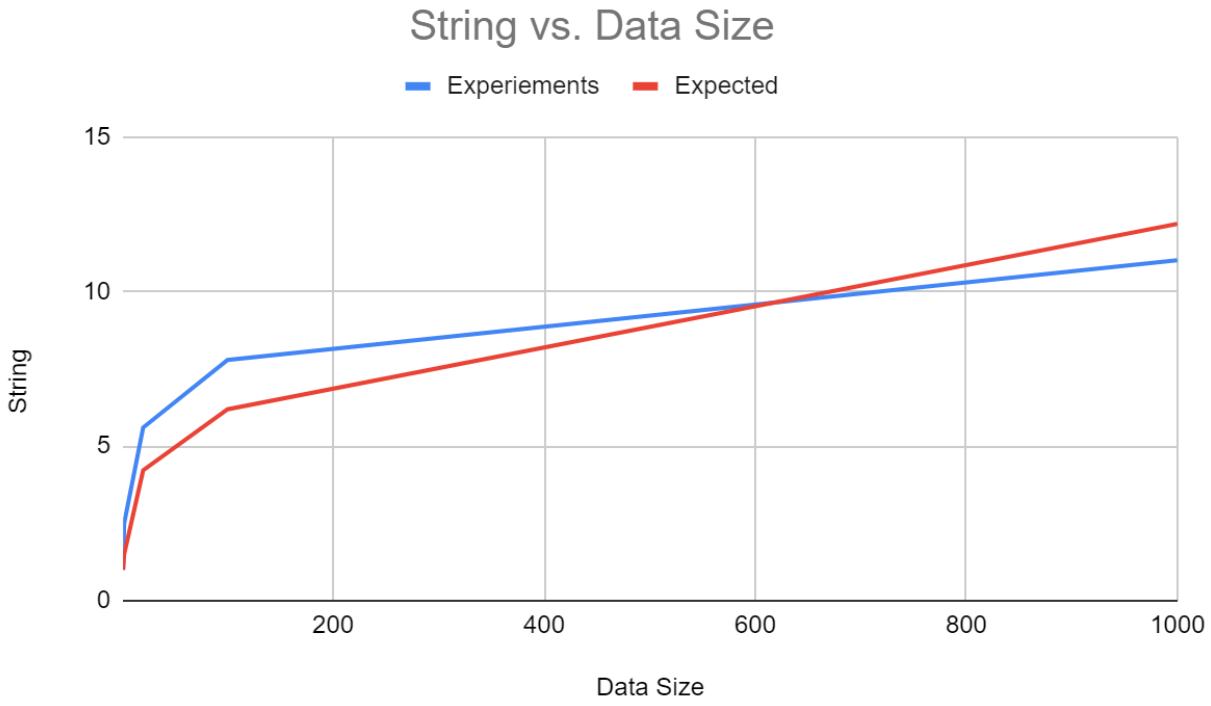
Comparison With Experimental Result

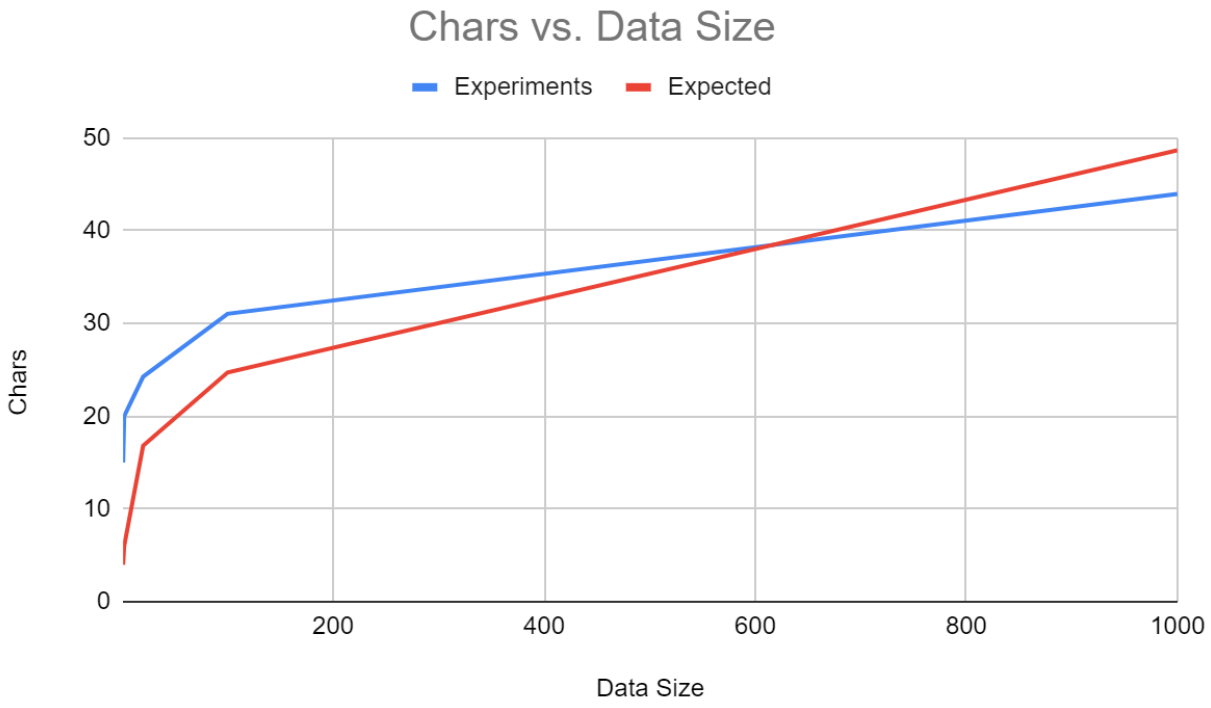
Using the provided reference output, I computed the average number of character comparisons needed before determining the result of comparisons between two strings. The result is just under 4 (3.99). Consequently, the predicted number of character comparisons will be four times the number of strings. For stage 2, where the smallest unit of comparison is on a byte-wise basis, the number of bits required for comparison will simply be 8 times the character count. The summarized results are presented in the following table and graphs (rounded to the nearest integer):

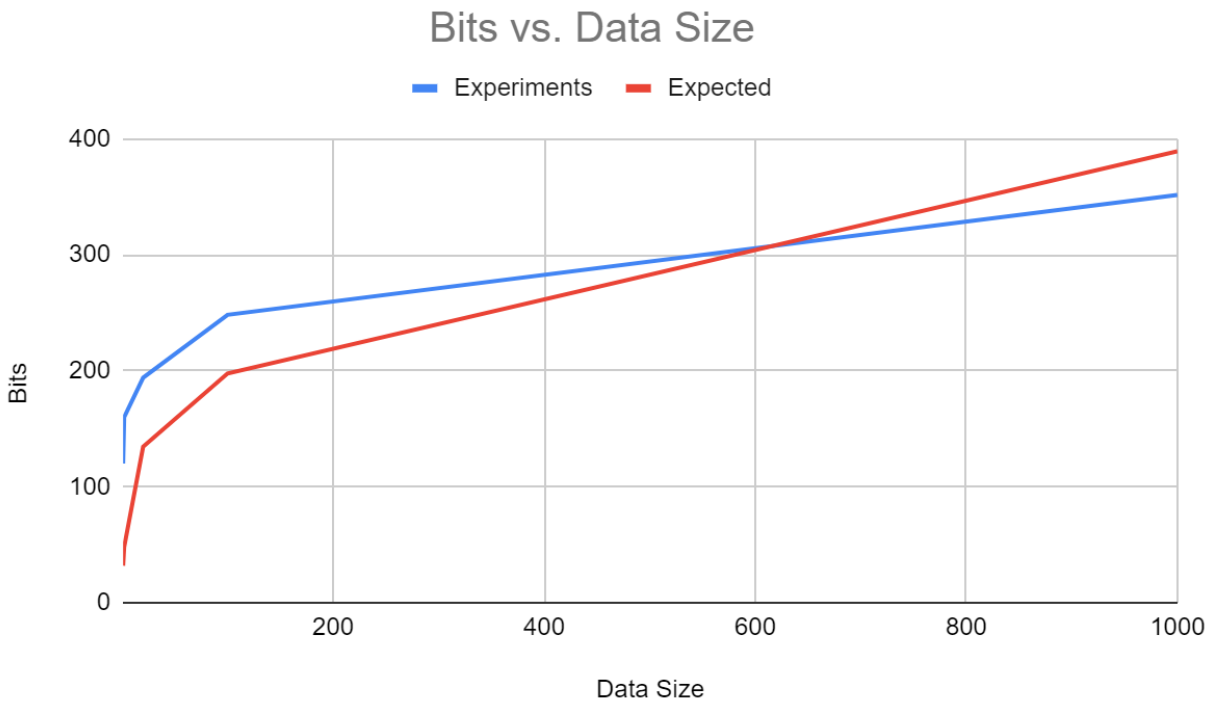
Expected			
Data Size/Total Comparison	Bits	Chars	String
1	120	15	1
2	160	20	2.5
20	194	24.25	5.6
100	248.3265306	31.04081633	7.785714286
1000	351.6004872	43.9500609	11.01218027

Expected			
Data Size	Bits	Chars	String
1	31.92832651	3.991040814	1
2	47.89248977	5.986561221	1.5
20	134.6129026	16.82661283	4.216096405
100	197.6441134	24.70551418	6.190243431
1000	389.2831376	48.6603922	12.19240656

P.S. The number of items that shares the same prefix as the key is assumed to be 10 for the 1000 dataset, 6 for the 100 dataset, 4 for the 20 dataset, none for both 2 and 1 dataset.


String vs. Data Size
Experiements
Expected
15
10
5
0
String
200
400
600
800
1000
Data Size


Chars vs. Data Size
Experiments
Expected
50
40
30
20
10
0
Chars
200
400
600
800
1000
Data Size



String Comparison

The results reveal that the actual number of comparisons needed for searching remains relatively consistently lower than the predicted number of comparisons for smaller datasets, specifically those containing fewer than 600 items. However, as the dataset size increases, the predictions notably surpass the experimental results. One primary factor contributing to this result is the selection of a "reasonable number of duplicates," which may not accurately represent the actual number of duplicates in the dataset. Consequently, this leads to a prediction of a less efficient linear search compared to what occurs in reality.

Another contributing factor could be the increasing significance of the number of comparisons in binary search predictions as the dataset expands beyond actual conditions. In predictions, the number of string comparisons in binary search consistently follows a logarithmic pattern (log(n)), where n represents the dataset size. In practice, it is probable that the number of comparisons required to locate the target is considerably lower than the worst-case scenario for most keys. This difference in actual versus predicted comparison counts contributes to the observed disparity in the displayed graphs above.

Character and Bits Comparison

Analyzing character and bit comparisons introduces complexity due to underlying assumptions, raising doubts about their validity. The initial assumption suggests that comparison functions stop at the first difference, complicating the process of averaging comparisons across the dataset, especially when string and key lengths vary. Consequently, relying on reference test results for the largest dataset provides a logical approach, closely approximating the actual average character and bit comparisons during string searches.

Both character and bit comparisons exhibit similar trends to string comparisons. Theoretical predictions and experimental results align for smaller datasets but diverge as the dataset grows larger. This isn't surprising since the predicted number of characters and bits compared linearly correlates with the number of strings compared. Thus, the analysis applied to string comparisons is also applicable to character and bit comparisons.

Brief Comments on Implementation

This implementation seems needlessly complex. It involves storing data names in an array of strings and, when presenting the findings, requires accessing these names to search for the associated data in the linked list from stage 1.

Upon reflection, a more streamlined approach would have involved constructing a dynamic array of cafes from the outset and adapting the insertion, sorting, and search functions accordingly. This approach would likely have resulted in fewer variables to manage and consequently cleaner code. Regrettably, time constraints prevented the implementation of this alternative approach.

Stage 3 - Radix Tree

Implementation Logic

I can only comment on my attempt to implement the radix tree since it could not be finished in time. However, towards the deadline of the assignments I understood the radix tree data structure and the associated insertion and searching algorithms.

I opted to implement the radix tree according to the suggestion provided in the implementation detail, where two functions of getBit and splitStem are used to read the specific bits of a string and obtain an array of characters that store a portion of the bits of the original strings.

Insertion[1]

The overall logic of the insertion involves the following steps (iteratively)::

1. Traverse through the tree by comparing the each binary bit of the key and the prefix of the current node.
2. If the length of the common prefix in the binary bit is the same as the length of the prefix of the current node, then the key will be split from the first bit of difference to the end and then update the insert location to one of the two branches depending on if the first different binary bit of the key is 0 (to A) or 1 (to B).
3. If the length of the common prefix is smaller than the length of the prefix in the current node, then the prefix of the current node needs to be updated. First, split the prefix of the current node from the first different binary bit to its final binary bit and insert it into branchA or B depending on the value of the first different binary bit, if the prefix has a non-null data list, assign it to the child branch, update the prefix length of the child branch and then set the branchA and branchB of the grandchildren branch to be NULL. Second, update the prefix length of the current node and split the prefix of the current node from beginning to the longest common prefix. Third, split the key from the first different binary bit to the final bit of the key update the insert location accordingly
4. Repeat step 3 and 4 until the insert location is a null pointer (meaning this is a new key) or the remaining bits of the key is 0 (which means the key is already in the tree).
5. Allocate memory for a new node. Insert the remaining binary bits of the key as the prefix, create a new link-list to store the data, update the prefix length of the insert location and set both branches to NULL.
6. Return the root.

Searching

1. Access the root, compare the value of the binary bit of the prefix and the key until all prefix has been compared, if any bits are found different before the length of the prefix, then the key is not stored in the tree and return nothing.
2. Check the first bit of the key after the prefixLength of the node, if getBit returns 1 then go to branchB and branchA for a return value of 0.
3. During comparison with the prefix, the bitCount will increment by one with each comparison of bits, and charCount will increase by 1 for every 8 bits. After the comparison of each node if the number of bit comparison mod 8 is not 0, it indicates there are extra bits stored in a character hence the charCount will need to increase by one.
4. Repeat until the data is not found or the remaining bits of the key is 0. If not found, return nothing. If found but not at leaf, keep traversing through both branches of the final node without comparisons until leaf and print the data to the outFile. If found at a leaf, just print the datalist associated.

[1] COMP20003 Week5 Workshop Solution of the implementation of the bst insertion inspired the iterative approach, with Grady suggesting that iterative insertion provides easier debugging experience.

Free Memories Recursively

1. If the tree is a null pointer, then return nothing.
2. If branchA is not NULL, call the free function on branchA
3. If branchB is not NULL, call the free function on branchB
4. Free the prefix
5. If the list is not NULL, free the list using the listFree function in list.h module
6. Free the tree node.

Theoretical Prediction

In my attempted implementation, if the key were to match any data, the maximum number of bit comparisons would equal the number of bits required to store the key. This is because every binary bit of the key must be compared against the tree's branches to determine whether it has been found, hence it will have time complexity due to bitwise comparison of O(n) where n is the number of bits required for the computer to store the key.

For every 8 bits of comparison undertaken, it is effectively a comparison of a one byte character. And remainder bits will also have to be stored in a one byte character, which means during searching, an extra character will need to be accessed to compare the remaining bits, hence the number of character comparisons will also have to increase by 1 in this case, and from experience this is a highly probable scenario when searching, hence will have a time complexity due to character comparison of O(n), where n is the number of character of the key.

This indicates that regardless of the size of the datasize, assuming the input search key remains relatively small and constant (meaning the length of the input strings in terms of characters and hence bits are not significantly different) then it is possible to get constant lookup time regardless of dataset size.

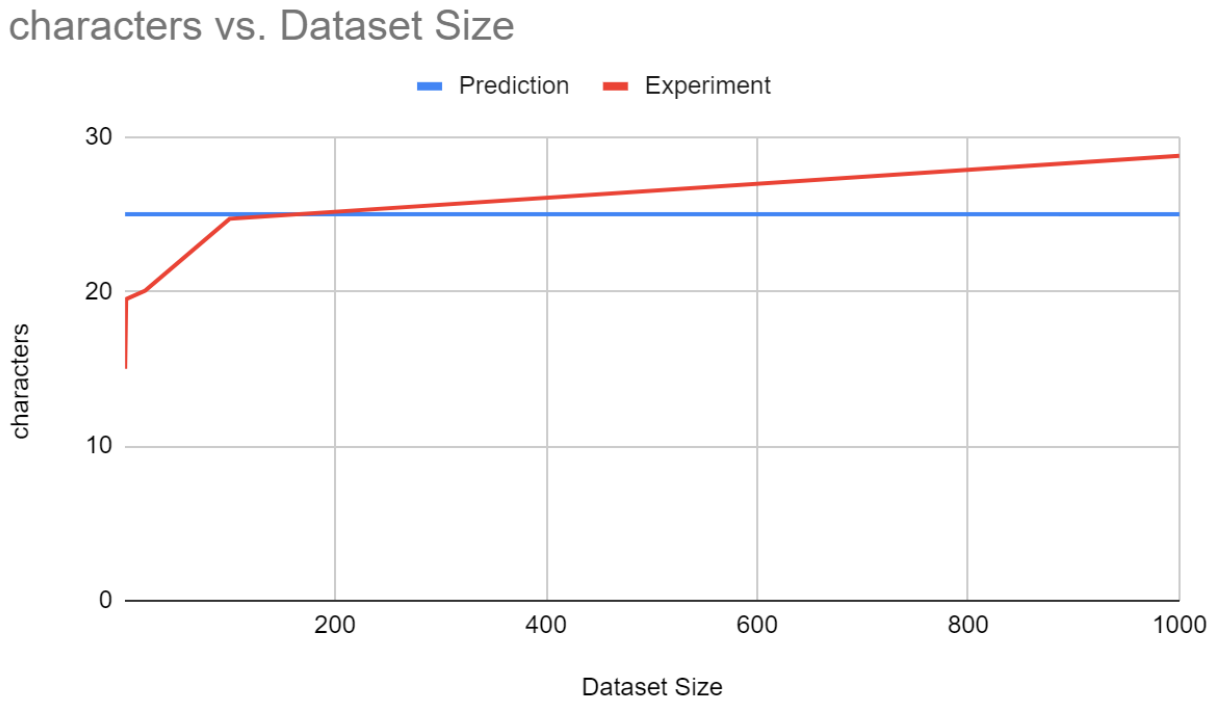
Due to the way the radix tree data structure stores the data and the search algorithm only searches through one branch of the entire tree for all input, the effective number comparison of strings will always be 1.

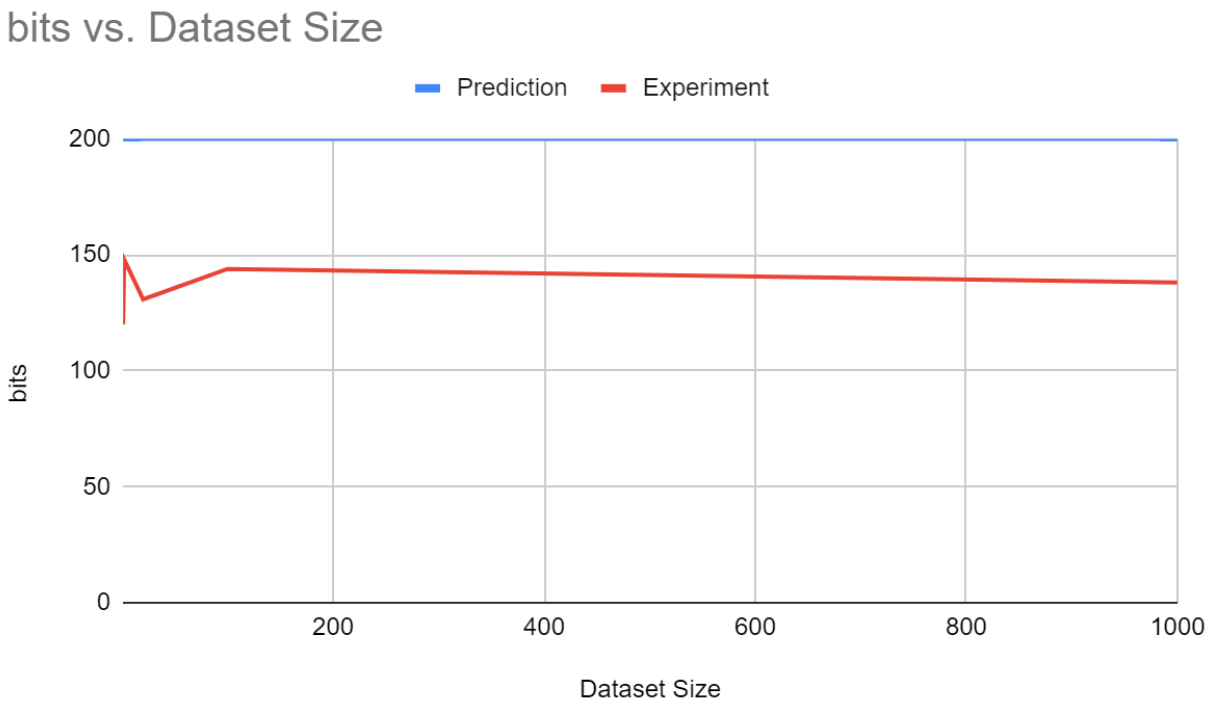
The search algorithm has a time complexity of theta(log(n)) where n is the number of nodes required to store all the trading names of the cafes in the dataset. This is because a radix tree with radix of 2 that stores binary data rather than characters will always be complete. It is possible to end up with an extremely left or right heavy tree that extremely closely resembles a stick with extremely unfavorable datasets that will make the time complexity approach O(n) where n is the number of nodes required to store the data. (it is impossible for a stick to be formed if the dataset contains different elements as it always split during insertion in such cases), however it is so improbable that this report will neglect to account for it.

Comparison With Reference Experiment Result

Expected			
Dataset Size	strings	characters	bits
1	1	25	200
2	1	25	200
20	1	25	200
100	1	25	200
1000	1	25	200

Experiment Result			
Data Size	strings	characters	bits
1	1	15	120
2	1	19.5	148
20	1	20.05	130.8
100	1	24.70408163	143.9183673
1000	1	28.77466504	138.1059683





The experimental data reveals that the average number of comparisons per search remains relatively consistent regardless of the dataset size. This observation validates the prediction that the number of required comparisons is indeed unaffected by the dataset size.

The discrepancy between the predicted bit comparisons and the experimental results can be attributed to the assumption that, on average, each key requires approximately 25 characters. In reality, the average key length in terms of characters is smaller. As the dataset size increases, this discrepancy grows progressively larger, causing the prediction to increasingly deviate from the actual average.

Performance Comparison between Data Structures

Theoretical Prediction

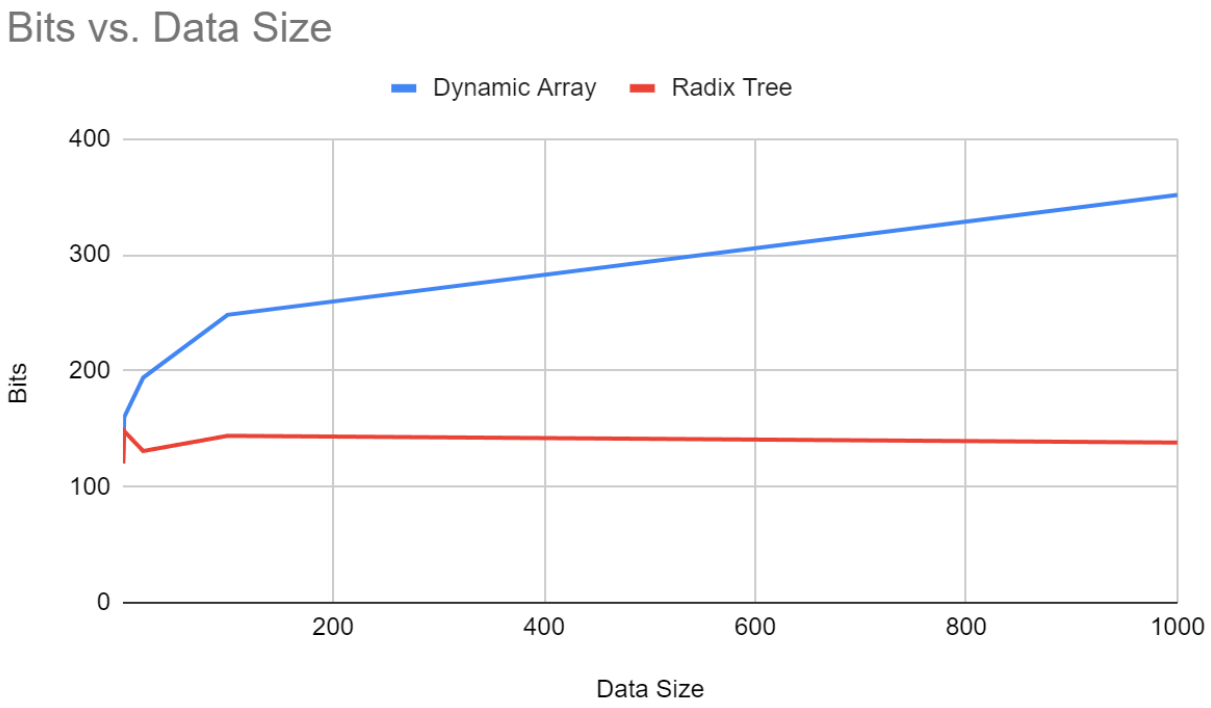
As previously mentioned, the configuration of the radix tree and its search algorithm guarantees that the number of required string comparisons will consistently be 1. This stands in contrast to dynamic arrays, where the worst-case scenario is O(log(n)), with 'n' representing the dataset

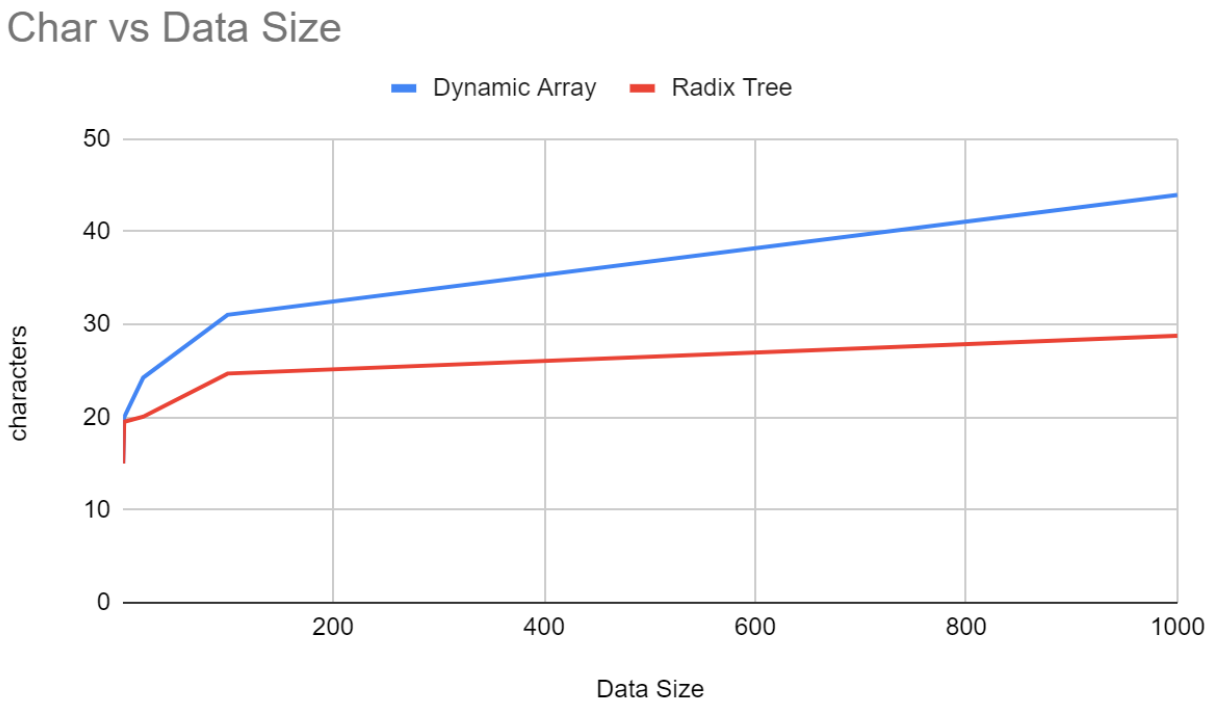
size. Consequently, it's evident that for most large datasets, a radix tree will outperform dynamic arrays in terms of the required number of string comparisons. Thus, a more comprehensive analysis of this aspect will be omitted, as it lacks novelty.

When considering both character and bit comparisons, the radix tree should also exhibit superior performance compared to dynamic arrays, particularly for large datasets. This stems from the fact that, for autocomplete lookups, the number of character comparisons is limited to the length of the longest string in the dataset. Similarly, the number of bit comparisons is constrained by the number of bits needed to store a string, a value that typically remains unaffected by the dataset size. This results in more efficient comparisons compared to multiple string comparisons of varying lengths.

However, in terms of memory usage, constructing a tree data structure may incur a higher memory overhead than a dynamic array. This is primarily due to the necessity of tracking the memory location of each node's branches. Moreover, the more complex memory management required during insertion poses an inherent risk of memory leaks.

Analysis of Experimental Result Against Prediction





The experimental results affirm the prediction that the number of required character and bit comparisons is consistently higher for dynamic arrays in comparison to radix trees. This underscores the fact that the radix tree data structure indeed offers superior performance for autocomplete lookups in the context of large datasets. Importantly, this superiority will only become more pronounced as the dataset size continues to expand.

Conclusion

The experimental results unequivocally demonstrate that the radix tree data structure significantly outperforms the dynamic array when it comes to search performance in the implementation of an autocomplete lookup dictionary. This strong empirical evidence aligns seamlessly with the theoretical predictions.