Final Project

Group 10 - Dorothy Sahijwani, Fatimah Kamal, Izzat Ikram, Sameer Khan

INFR 2828 - Algorithms and Data Structure

Professor Sara Motlagh

Jul 31, 2025

a. Space Efficient and how. Support your answer by explaining how you achieved the notation for example if it's log(n), how you achieved log(n).

For the Standard Trie, every node stores a dictionary of its children. For n words with an average length m, the space is about O(n * m). Since we use a dictionary, we don't pre-allocate storage like an array does, so we don't include the alphabet size in the complexity. However, each node still has some overhead because dictionaries store extra information. This overhead becomes noticeable when nodes have many children or when the alphabet is large.

The Ternary Trie is different because each node only has three pointers (left, middle, right) and a single character. It still creates one node per character, so it's also O(n * m), but it avoids the extra dictionary or array overhead. This makes it more space-efficient, especially when the alphabet is big.

Conclusion: Even though both have the same O(n * m) complexity, the Ternary Trie uses memory better because it only stores three pointers per node. The Standard Trie's dictionaries add extra overhead, so in real usage, it takes up more space.

b. Time Efficient and how. Support your answer by explaining how you achieved the notation for example if it's log(n), how you achieved log(n).

For the Standard Trie, inserting or searching goes character by character through the word. Looking up a character in a dictionary is O(1), so these operations take O(m), where m is the length of the word. For autocomplete, we first find the prefix in O(m) and then collect all words that match it, which takes p time, where p is the number of results. So the total for autocomplete is O(m + p).

For the Ternary Trie, the time depends on how balanced it is. If it's balanced, each step to find a character takes about O(log a), where a is the alphabet size. This makes inserting and searching O(m log a) on average. In the worst case, if the tree becomes unbalanced, it could take O(a) comparisons for each character, so O(m * a). For autocomplete, we first get to the end of the prefix in O(m log a) (average case), then we still have to collect the p matches. This gives O(m log a + p) overall.

Conclusion: Standard Trie is usually faster because it does direct lookups, while the Ternary Trie has extra comparisons that can make it slower.

c) Let’s suppose that we bring in ranking algorithm in our project, ranking algorithm work on queries, for example I search the word Sample twice and rest of the word with the same prefix (Same, Sampling, Samplers) only once, now if I type Sam, it should bring Sample on top and rest below it (based on alphabetical order or any other order). Is it going to impact space and time complexity overall project (Project Part-1 a & b part)? If yes how? If no, why? Explain your answer.

How adding Ranking would affect space complexity:

Adding a ranking system would require a frequency counter for each word that tracks how often it has been searched. For example, if the user searches for “Sample” multiple times, the frequency would increase, bringing the word “Sample” higher in the rank. So this adds only a small constant memory overhead per word. So the space complexity isn't significantly changed.. This is because of how minor the space the frequency counter would take up.

Standard Trie: O(n*m)

Ternary Trie: O(n*m)

How Ranking would affect search time:

With ranking, after a prefix is inputted, the results would be placed sorted, based on frequency. This means that once all words matching the prefix are found (let’s say *p* words), the system needs to compare their frequencies and sort them accordingly. This would incur a cost of p log p, because of the constant comparison and sorting involved.

Revised Time Complexity after implementing Ranking:

Standart Trie: O(m + p log p)

Ternary Trie: O(m*log a +p log p)

Here, *m* represents the length of the input prefix, and *a* is the size of the alphabet.

In the Standard Trie, the time complexity for retrieving autocomplete suggestions with ranking is O(m + p log p). The O(m) comes from going through one character at a time till it reaches short nodes that match the user input prefix. So if the user writes, "Sam," the Trie checks 'S', then 'A', then 'M'. after it reaches that point, the Trie collects all the words that start with the prefix. If there are p number words, they are ranked based on how often each word is used. because of this O(p log p) comes into play. a swords the words by frequency using sorting algorithms like merge sort. For example, if “Sample” is used more than “Same” or “Samplers,” sorting makes sure it shows up first in the suggestions. So overall the time it takes to reach the prefix and sort all the matching words gives the total time complexity of O(m + p log p).
In a Ternary Search Trie, the time complexity for retrieving autocomplete suggestions with ranking is O(m × log a + p log p). This is because it takes longer to reach the prefix than in a

standard trie. Instead of following one direct path for each letter, the Ternary Trie uses left, middle, and right pointers, which work like binary comparisons. So For each character in the prefix (m characters), this search can take O(log a) time, where else a is the total number of letters in the alphabet. So the total time for prefix traversal becomes O(m × log a). After reaching the right node, we still need to collect all matching words, and if there are p of them, we sort them by frequency to rank them. So this sorting step takes O(p log p) time. In the end, the time complexity for autocomplete with ranking in a Ternary Search Trie is O(m × log a + p log p).

d) If we deploy the same project using Compressed Trie (including point c of analysis part), what would be the time and space complexity. Do explain/show how you will achieve that
time and space complexity.

Adding Compressed Trie, where multiple nodes are put together into a single node, would impact the space and time complexity greatly.

Theoretical explanation:

Instead of having three nodes s - a - m, compressed trie would have one node representing “sam”, lowering the height of the tree, especially when there are a large amount of words sharing the same prefix (our example)

Space Complexity impact:
Compressed tries are more space efficient compared to our standard trie and Ternary trie. This is because we no longer have to have a node for every single character in the alphabet. A compressed trie’s would have an O(n) complexity where n is the number of words in the dictionary in the best case, based on the amount of endings we can get from a prefix. This takes up a lot less space compared to having more nodes for each letter of the prefix. In average or worst case, the complexity will be closer to O(n × m), where m is the average length of words

Time complexity:

The process of inserting and searching requires comparing whole strings within the nodes. The process would still begin with an O(m) time, with m representing the length of the prefix, because each string must be compared with the input prefix to determine which path to follow. Then, we would have to compare the prefix with the amount of endings that match the prefix, which would be O(m + p) where p is the number of words that match the prefix. Finally, since it is a Rank, we would have to sort out “p”, based on the frequency. Therefore the final time complexity of a compressed trie would be O(m +p log p)

e) Can we deploy the same project using an Array with Standard Trie? If yes, what would be the space and time complexity? If not, why?

Yes, we can create the same project using a standard Trie. Instead of using a dictionary to hold child nodes for each letter, we use an array of 26 slots 26 slots for each letter of the alphabet. So, because the letter 'c' is the third letter of the alphabet, we move to slot number 2 in the array to find its child node. This makes inserting and finding words extremely quick, because looking at a given slot in an array takes constant time, O(1), regardless of how large the Trie is. That's why inserting and searching takes O(m) time, where m is the length of the word.

But the downside is the memory. Every node in the Trie has a 26-slot array, even if only one or two are used. This wastes space, especially when words have few common letters. So, while the Trie is faster, it requires a lot more memory than a dictionary that simply stores the letters it needs. They have the worst time complexity of O(26 × a × L), where a is the number of words and L is the average word length. This means that even if a node only has one child, it will reserve memory for all 26 slots, wasting space.

So that's why it is highly time-efficient, but not space-efficient.

f) Can we deploy the same project using a LinkedList with Standard Trie? If yes, what would be the space and time complexity? If not, why?

Yes, we can build the same project using a Standard Trie and a LinkedList. Instead of using a dictionary or an array to store the child nodes for each letter, we use a linked list. This means that each node keeps a list of its children, which we walk through one by one to find the letter we're looking for. For example, if a node has three children, 'a', 'c', and 'd', and we want to find 'd', we must first check 'a', then 'c', and finally 'd', which takes longer as the number of children increases. The time complexity becomes O(L × k), where L is the length of the word and k is the number of children per node. In the worst-case scenario, with 26 children at each node, inserting or finding a word can take significantly longer.

But the good part is the memory. Unlike an array that always reserves 26 slots, a linked list only uses memory for the letters it requires. Because of that, if a node only has two children, it will only take up two spaces rather than 26. This saves a lot of space, especially when the Trie is small and the majority of the nodes have few children. Therefore, the space complexity is O(n × L), where n is the number of words and L is the average word length.

So that's why using a LinkedList with a Trie is space-efficient but not time-efficient.