**Implement Trie - Solution**

Problem: Implement a trie with insert, search, and startsWith methods.

*Solution*

The key to this problem is to design the Trie node structure. In order to know if the string represented by the node is a word or not, we need an extra boolean flag.

When we insert a new word, we will set the flag in the end node of the word to be true. When we implement the startsWith method, we return true if we successfully find the path. However, when we implement the search method, we return true only if we successfully find the path and the flag of the end node is true.

We provide both C++ and Java codes for your reference. We implement Trie using the map.

|  |
| --- |
| **Implement Trie - Solution**  Problem: Implement a trie with insert, search, and startsWith methods.    *Solution*  The key to this problem is to design the Trie node structure. In order to know if the string represented by the node is a word or not, we need an extra boolean flag.  When we insert a new word, we will set the flag in the end node of the word to be true. When we implement the startsWith method, we return true if we successfully find the path. However, when we implement the search method, we return true only if we successfully find the path and the flag of the end node is true.  We provide both C++ and Java codes for your reference. We implement Trie using the map.    *Complexity Analysis*  Let's discuss the complexity of this algorithm.  If the longest length of the word is N, the height of Trie will be N + 1. Therefore, the time complexity of all insert, search and startsWith methods will be O(N).  How about space complexity?  If we have M words to insert in total and the length of words is at most N, there will be at most M\*N nodes in the worst case (any two words don't have a common prefix).  Let's assume that there are maximum K different characters (K is equal to 26 in this problem, but might differs in different cases). So each node will maintain a map whose size is at most K.  Therefore, the space complexity will be O(M\*N\*K).  It seems that Trie is really space consuming, however, the real space complexity of Trie is much smaller than our estimation, especially when the distribution of words is dense.  You can also implement it by the array which will achieve a slightly better time performance but a slightly lower space performance.    *Comparison with Hash Table*  You might wonder why not use a hash table to store strings. Let's do a brief comparison between these two data structures. We assume there are N keys and the maximum length of a key is M.   1. Time Complexity   The time complexity to search in hash table is typically O(1), but will be O(logN) in the worst time if there are too many collisions and we solve collisions using height-balanced BST.  The time complexity to search in Trie is O(M).  The hash table wins in most cases.   1. Space Complexity   The space complexity of hash table is O(M \* N). If you want hash table to have the same function with Trie, you might need to store several copies of the key. For instance, you might want to store "a", "ap", "app", "appl" and also "apple" for a keyword "apple" in order to search by prefix. The space complexity can be even much larger in that case.  The space complexity of Trie is O(M \* N) as we estimated above. But actually far smaller than the estimation since there will be a lot of words have the similar prefix in real cases.  Trie wins in most cases. |

*Complexity Analysis*

Let's discuss the complexity of this algorithm.

If the longest length of the word is N, the height of Trie will be N + 1. Therefore, the time complexity of all insert, search and startsWith methods will be O(N).

How about space complexity?

If we have M words to insert in total and the length of words is at most N, there will be at most M\*N nodes in the worst case (any two words don't have a common prefix).

Let's assume that there are maximum K different characters (K is equal to 26 in this problem, but might differs in different cases). So each node will maintain a map whose size is at most K.

Therefore, the space complexity will be O(M\*N\*K).

It seems that Trie is really space consuming, however, the real space complexity of Trie is much smaller than our estimation, especially when the distribution of words is dense.

You can also implement it by the array which will achieve a slightly better time performance but a slightly lower space performance.

*Comparison with Hash Table*

You might wonder why not use a hash table to store strings. Let's do a brief comparison between these two data structures. We assume there are N keys and the maximum length of a key is M.

1. Time Complexity

The time complexity to search in hash table is typically O(1), but will be O(logN) in the worst time if there are too many collisions and we solve collisions using height-balanced BST.

The time complexity to search in Trie is O(M).

The hash table wins in most cases.

1. Space Complexity

The space complexity of hash table is O(M \* N). If you want hash table to have the same function with Trie, you might need to store several copies of the key. For instance, you might want to store "a", "ap", "app", "appl" and also "apple" for a keyword "apple" in order to search by prefix. The space complexity can be even much larger in that case.

The space complexity of Trie is O(M \* N) as we estimated above. But actually far smaller than the estimation since there will be a lot of words have the similar prefix in real cases.

Trie wins in most cases.