Solution: Design Add and Search Words Data Structure

Let's solve the Design Add and Search Words Data Structure problem using the Trie pattern.

We'll cover the following

Statement

Design a data structure called **WordDictionary** that supports the following functionalities:

- **Constructor:** This function will initialize the object.
- **Add Word(word):** This function will store the provided word in the data structure.
- **Search Word(word):** This function will return TRUE if any string in the **WordDictionary** object matches the query word. Otherwise, it will return FALSE. If the query word contains dots, `.`, each dot is free to match any letter of the alphabet.

  For example, the dot in the string ".ad" can have 26 possible search results like "aad", "bad", "cad", and so on.
- **Get Words():** This function will return all the words in the **WordDictionary** class.

Constraints:

- $1 \leq$ `word.length` ≤ 25
- Words passed to **Add Word()** consist of lowercase English letters.
- Words passed to **Search Word()** consist of `.` or lowercase English letters.
- There will be, at most, three dots in a word passed to **Search Word()**.
- At most, 10^3 calls will be made to **Add Word()** and **Search Word()**.

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The problem statement asks us to devise an algorithm to support the three mentioned functionalities. One possible approach is to utilize a hash map. We can start by creating an empty hash map such that the key is the length of the word and the value is the word itself. If we get multiple words of the same length, we create a list of words, instead of a single value, and associate this list with the key. When searching for a word in the hash map, we use the length of the word as the key and then go through the associated list of words looking for our word. In this search, we would have to compare each word, one character at a time, with our word.

Function	Time complexity	Space complexity
Add Word()	*O(1)*	*O(1)*
Search Word()	*O(w)*, where *w* is the number of words that have the same length as the word being search	*O(1)*
Get Words()	*O(t)*, where *t* is the total number of words	*O(1)*

However, this solution would be inefficient when searching for all strings with a common prefix.

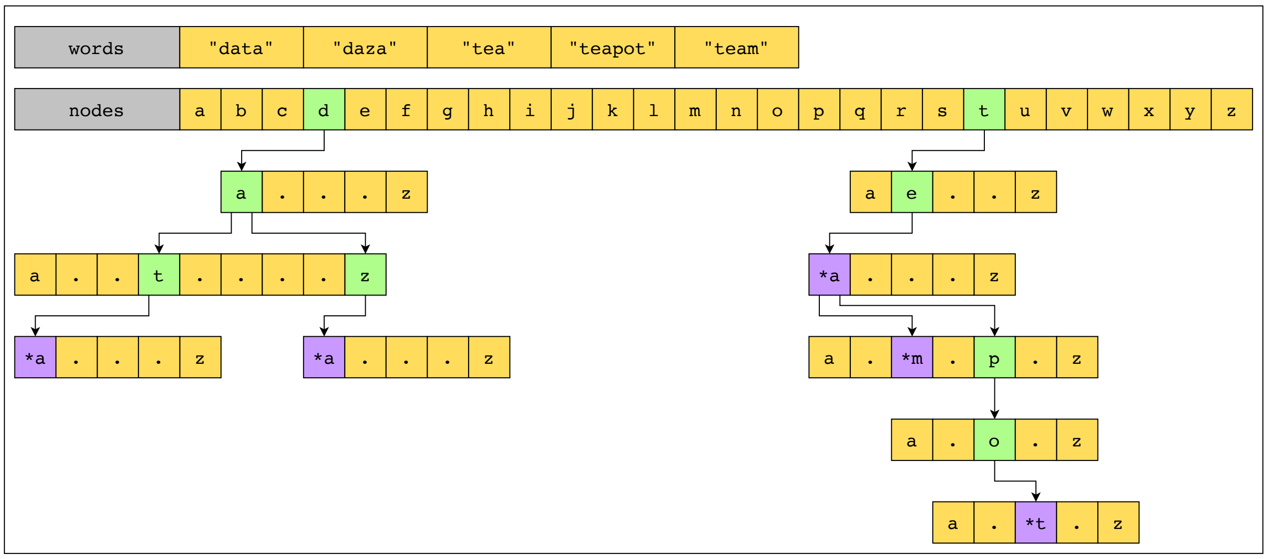
Furthermore, once the size of the hash map increases, there will be a large number of words against each key, which would result in poor performance when searching for a word.

Optimized approach using trie

A more efficient approach is to use a trie data structure. Using a trie, we can search for words that exactly match a query string, as well as retrieve words that partially match query strings with wildcard characters. Each node in a trie is a specialized data structure, a trie node, featuring the following members:

- **Children:** This is an array of length 26. Each index of this array corresponds to a letter of the English alphabet.
- **Complete:** This variable takes a boolean value. The value will be TRUE if the node represents the last character of the word. Otherwise, it will be FALSE.

For better understanding, let's look at the diagram below:



While adding a word, we'll verify at each step if the child node that needs to be added is already present. If it's already present, we'll go down one step. Otherwise, we'll add the child node to the trie and go down one step.

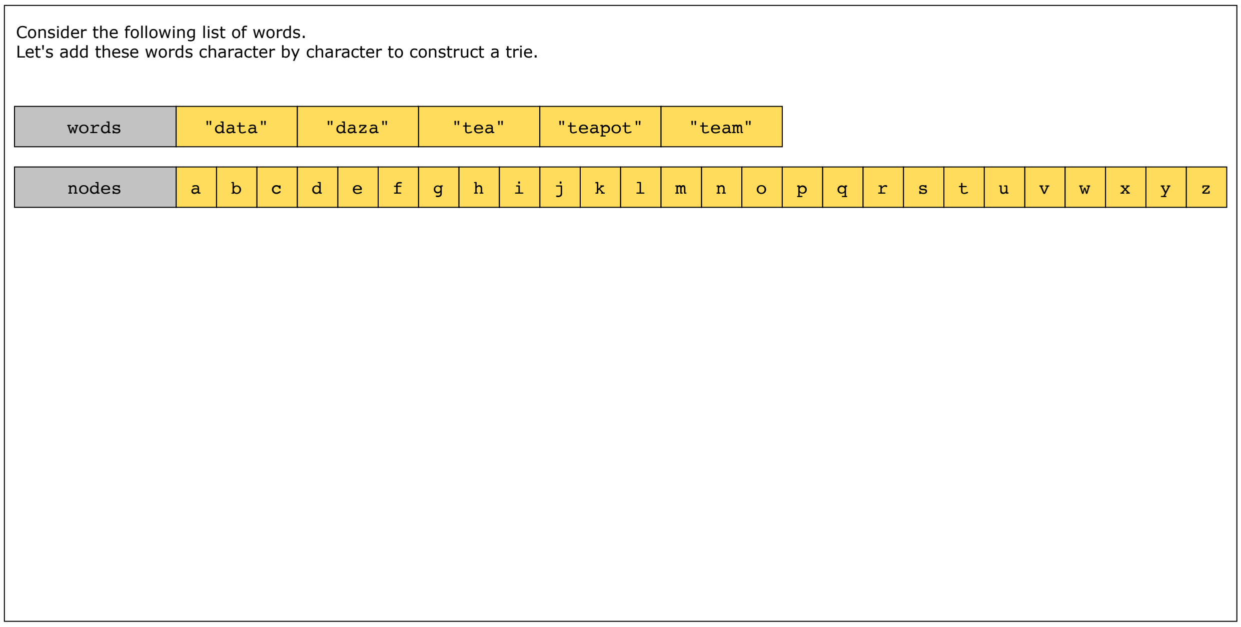
For searches, there are these two possibilities:

1. If the current character is not ".", the search will be as simple as the add feature: we start from the root and go down the trie, looking for the current character of the query string among the children of the current trie node. If the character is not found, the search returns FALSE. If the character is found, we move to the matching child of the current node and search among its children for the next character of

the query string. This process continues until a character is not found, or until all the characters in the query string are found in the trie.

2. When the current character is “.”, all the children of the current trie node are considered valid matches for this position in the query string. So, for each child, we need to check *its* children to see if one of them matches the character following “.”.

The slides below illustrate how the algorithm runs:



Consider the following list of words.
Let's add these words character by character to construct a trie.

words	"data"	"daza"	"tea"	"teapot"	"team"

nodes	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

1 of 24

Let’s look at the code for this solution below:

Java

main.java

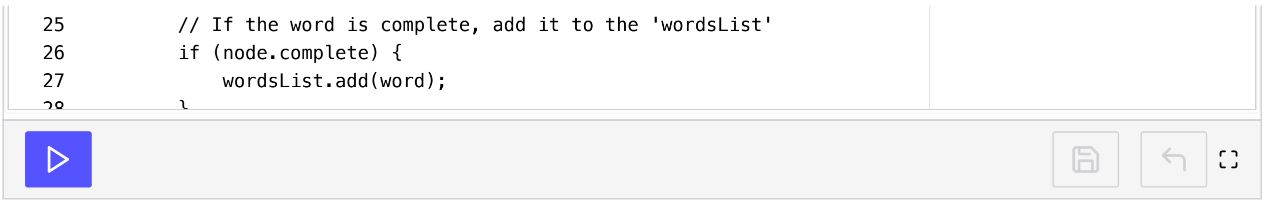
TrieNode.java

```java
class WordDictionary {
    TrieNode root;
    boolean canFind;

    // Initialize the root with TrieNode and set
    // the 'canFind' boolean to FALSE
    public WordDictionary() {
        root = new TrieNode();
        canFind = false;
    }

    // Function to get all words in the dictionary
    public List<String> getWords() {
        List<String> wordsList = new ArrayList < String > ();
        // Return an empty list if the root is NULL
        if (root == null)
            return new ArrayList < String > ();
        // Perform depth first search on the trie
        return DFS(root, "", wordsList);
    }

    private List<String> DFS(TrieNode node, String word, List<String> wordsLi
        // If the node is NULL, return the 'wordsList'
        if (node == null) return wordsList;
```



```
        // If the word is complete, add it to the 'wordsList'
        if (node.complete) {
            wordsList.add(word);
```

Design Add and Search Words Data Structure

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Add words to a trie data structure by creating a new node for each character.
2. For a simple search, start from the root, go down the trie, matching each query string character with a child of the current node.
3. When the query string contains a “.”, consider that all the children of the current trie node are valid matches and check all of their children against the next character in the query string.

Time complexity

- **Add Word(word):** At each step, we’ll either examine or create a node in the trie. In the worst-case scenario, the examination or creation of a node in the trie will take $O(m)$ operations, where m will be the length of the word. As $m \leq 25$, this means that the worst-case time complexity is $O(25)$, that is, $O(1)$.
- **Search Word(word):** Depending upon the search query, the time complexity for searching a word in a trie would vary:
  - **Scenario 1:** For a word without any “.”, the time complexity would be $O(m)$, where m is the length of the word. As $m \leq 25$, this means that the worst-case time complexity is $O(1)$.
  - **Scenario 2:** For a word that is a combination of letters and “.”, the function would have to explore 26 branches whenever a “.” is encountered. Let’s say the word to search is “...oo.”. Due to the first “.”, we would have to check all the children of the root node in the trie. Due to the second dot, we would have to check all the grandchildren of the root node. So, the cost of searching for this word depends on the number of wildcard characters, d, and the total length of the word m. So, the cost in this scenario will be $O((m - d) \times 26^d)$.

    Let’s consider the case where the $m - d$ term is maximized, that is, $25 - 1$. In this case, the time complexity is $O((24) \times 26^1)$, that is, $O(24 \times 26)$, that is, $O(1)$. Alternatively, we may want to

    complexity is $O((25 - 3) \times 26^3)$, that is, $O(22 \times 26^3)$, that is, $O(1)$.
  - **Scenario 3:** If the query word only contains wildcard characters, then searching for such a word will take $O(26^d)$, where d is the number of wildcard characters. As $d \leq 3$, this means that the worst-case time complexity is $O(26^3)$, that is, $O(1)$.
- **Get Words():** If the length of the longest word is l, and given that there are 26 letters in the English alphabet, the time complexity of retrieving all the words will be $O(26^l)$. This case occurs where all the children of every node in the trie are in use. As $l \leq 25$, this means that the worst-case time complexity is $O(1)$.

Space complexity

- **Add Word(word):** In the worst-case scenario, the newly inserted word doesn’t share a prefix with the words already inserted in the trie. For a word with m characters, we would need $O(m)$ additional space in memory. As $m \leq 25$, this means that the worst-case space complexity is $O(25)$, that is, $O(1)$.

- **Search Word(word):** In the worst case, for a word of length m, there would be $O(m)$ recursive calls to the search function, so we would take up $O(m)$ space on the stack. Again, as $m \leq 25$, this means that the worst-case space complexity is $O(1)$.
- **Get Words():** In the worst case, we would need to make $O(l)$ recursive calls to retrieve the longest word of length l, taking up $O(l)$ space on the stack. Again, as $l \leq 25$, the worst-case space complexity is $O(1)$.

← Back
Design Add and Searc...

Next →
Word Search II

✓ Mark as Completed