# Question

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

* WordDictionary() Initializes the object.
* void addWord(word) Adds word to the data structure, it can be matched later.
* bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

**Example:**

**Input**

["WordDictionary","addWord","addWord","addWord","search","search","search","search"]

[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]

**Output**

[null,null,null,null,false,true,true,true]

**Explanation**

WordDictionary wordDictionary = new WordDictionary();

wordDictionary.addWord("bad");

wordDictionary.addWord("dad");

wordDictionary.addWord("mad");

wordDictionary.search("pad"); // return False

wordDictionary.search("bad"); // return True

wordDictionary.search(".ad"); // return True

wordDictionary.search("b.."); // return True

**Constraints:**

* 1 <= word.length <= 500
* word in addWord consists lower-case English letters.
* word in search consist of  '.' or lower-case English letters.
* At most 50000 calls will be made to addWord and search.

# Solution

#### **Data Structure Trie**

This article introduces the data structure [trie](https://en.wikipedia.org/wiki/Trie). It could be pronounced in two different ways: as "tree" or "try". Trie which is also called a digital tree or a prefix tree is a kind of search ordered tree data structure mostly used for the efficient dynamic add/search operations with the strings.

Trie is widely used in real life: autocomplete search, spell checker, T9 predictive text, [IP routing (longest prefix matching)](https://www.researchgate.net/figure/An-example-routing-table-and-the-corresponding-binary-trie-built-from-it_fig3_4236637), [some GCC containers](https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/trie_based_containers.html).

Here is how it looks like

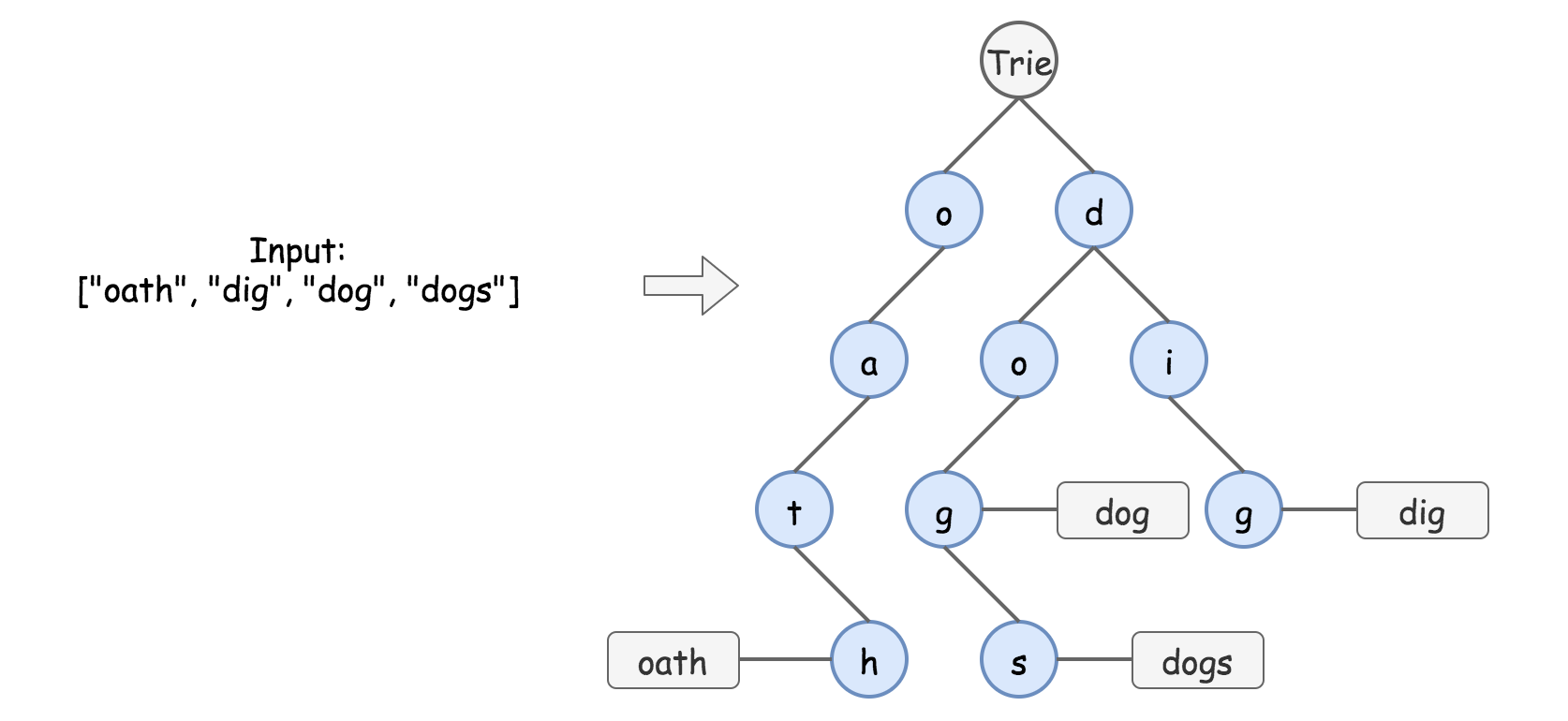


Figure 1. Data structure trie.

There are two main types of trie interview questions:

* [Standard Trie](https://en.wikipedia.org/wiki/Trie). Design a structure to dynamically add and search strings, for example
  + [Add and Search Word](https://leetcode.com/problems/design-add-and-search-words-data-structure/solution/).
  + [Word Search II](https://leetcode.com/articles/word-search-ii).
  + [Design Search Autocomplete System](https://leetcode.com/articles/design-search-autocomplete-system/).
* [Bitwise Trie](https://en.wikipedia.org/wiki/Trie#Bitwise_tries). Design a structure to dynamically add binary strings and compute maximum/minimum XOR/AND/etc, for example
  + [Maximum XOR of Two Number in an Array](https://leetcode.com/articles/maximum-xor-of-two-numbers-in-an-array/).

#### **Why Trie and not HashMap**

It's quite easy to write the solution using such data structures as hashmap or balanced tree.

|  |
| --- |
| class WordDictionary {  Map<Integer, Set<String>> d;  /\*\* Initialize your data structure here. \*/  public WordDictionary() {  d = new HashMap();  }  /\*\* Adds a word into the data structure. \*/  public void addWord(String word) {  int m = word.length();  if (!d.containsKey(m)) {  d.put(m, new HashSet());  }  d.get(m).add(word);  }  /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/  public boolean search(String word) {  int m = word.length();  if (d.containsKey(m)) {  for (String w : d.get(m)) {  int i = 0;  while ((i < m) && (w.charAt(i) == word.charAt(i) || word.charAt(i) == '.')) {  i++;  }  if (i == m) return true;  }  }  return false;  }  } |

This solution passes all leetcode test cases, and formally has \mathcal{O}(M \cdot N)O(*M*⋅*N*) time complexity for the search, where M*M* is a length of the word to find, and N*N* is the number of words. Although this solution is not efficient for the most important practical use cases:

* Finding all keys with a common prefix.
* Enumerating a dataset of strings in lexicographical order.
* Scaling for the large datasets. Once the hash table increases in size, there are a lot of hash collisions and the search time complexity could degrade toO(*N*^2⋅*M*) , where *N* is the number of the inserted keys.

Trie could use less space compared to hashmap when storing many keys with the same prefix. In this case, using trie has only \mathcal{O}(M \cdot N)O(*M*⋅*N*) time complexity, where M*M* is the key length, and N*N* is the number of keys.

#### **Approach 1: Trie**

**How to Implement Trie: addWord function**

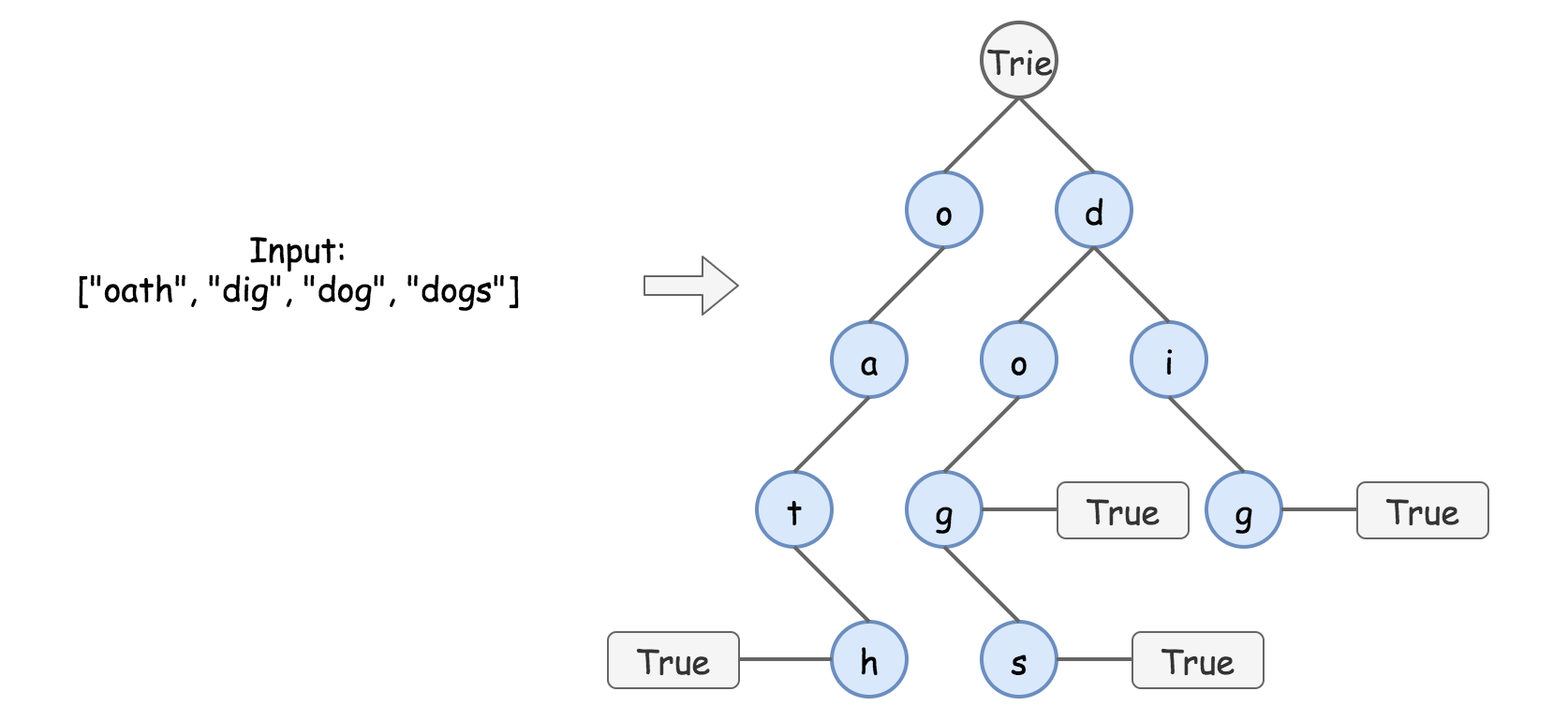


Figure 2. Trie implementation.

In trie, each path from the root to the "word" node represents one of the input words, for example, o -> a -> t -> h is "oath".

Trie implementation is pretty straightforward, it's basically nested hashmaps. At each step, one has to verify, if the child node to add is already present. If yes, just go one step down. If not, add it into the trie and then go one step down.

1 / 6

|  |
| --- |
| **class TrieNode {**  **Map<Character, TrieNode> children = new HashMap();**  **boolean word = false;**  **public TrieNode() {}**  **}**  **class WordDictionary {**  **TrieNode trie;**  **/\*\* Initialize your data structure here. \*/**  **public WordDictionary() {**  **trie = new TrieNode();**  **}**  **/\*\* Adds a word into the data structure. \*/**  **public void addWord(String word) {**  **TrieNode node = trie;**  **for (char ch : word.toCharArray()) {**  **if (!node.children.containsKey(ch)) {**  **node.children.put(ch, new TrieNode());**  **}**  **node = node.children.get(ch);**  **}**  **node.word = true;**  **}**  **}** |

**Complexity Analysis**

* Time complexity: \mathcal{O}(M)O(*M*), where M*M* is the key length. At each step, we either examine or create a node in the trie. That takes only M*M* operations.
* Space complexity: \mathcal{O}(M)O(*M*). In the worst-case newly inserted key doesn't share a prefix with the keys already inserted in the trie. We have to add M*M* new nodes, which takes \mathcal{O}(M)O(*M*) space.

**Search in Trie**

In the absence of '.' characters, the search would be as simple as addWord. Each key is represented in the trie as a path from the root to the internal node or leaf. We start from the root and go down in trie, checking character by character.

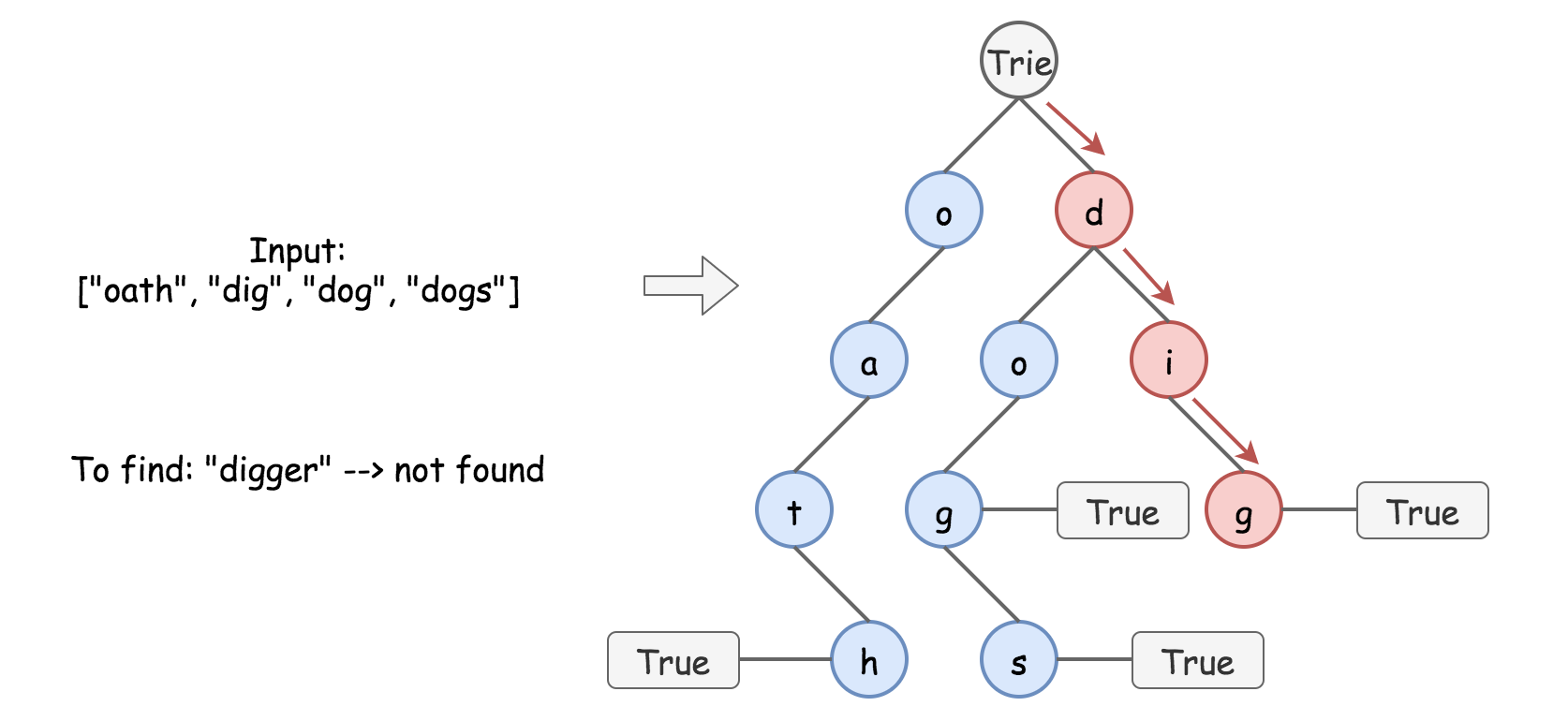


Figure 3. Search in trie.

The presence of '.' characters forces us to explore all possible paths at each . level.

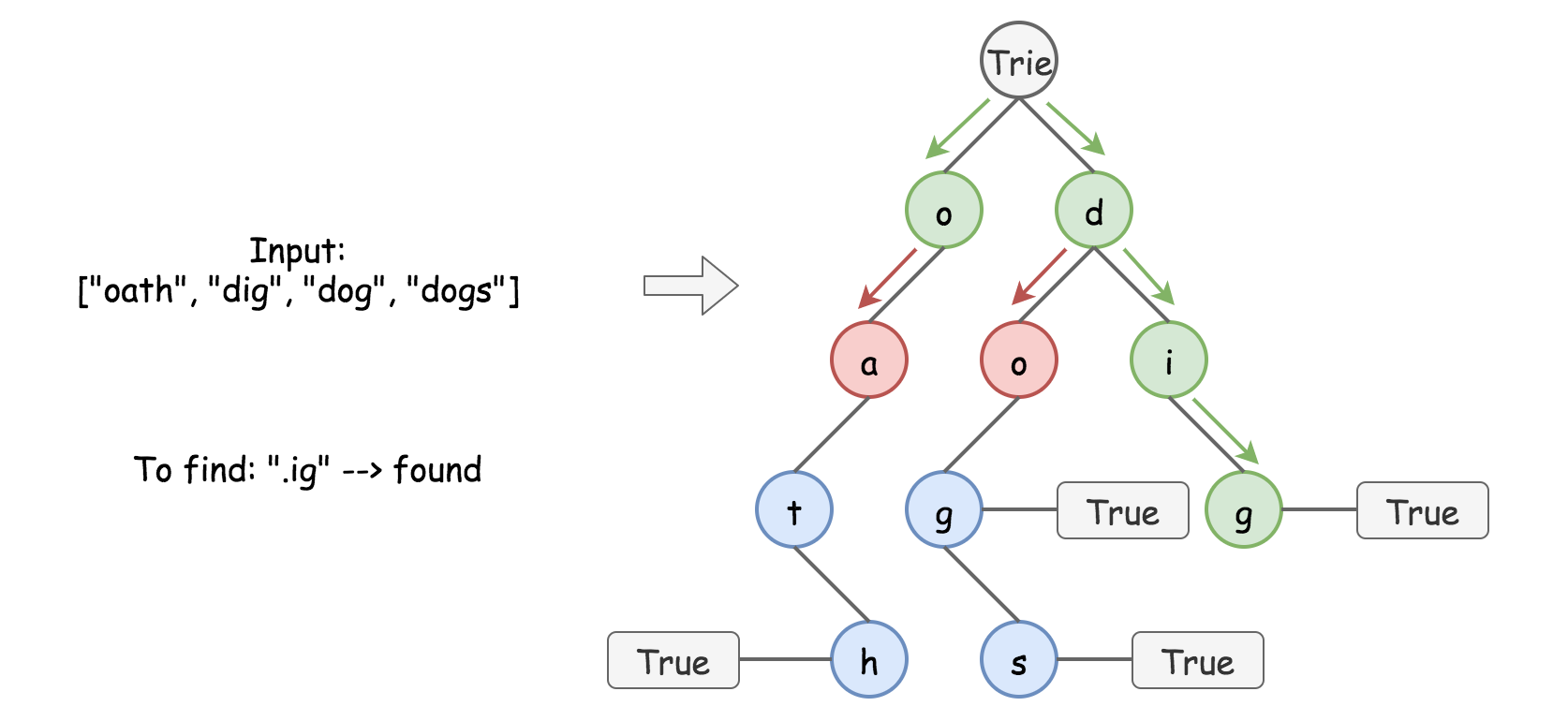
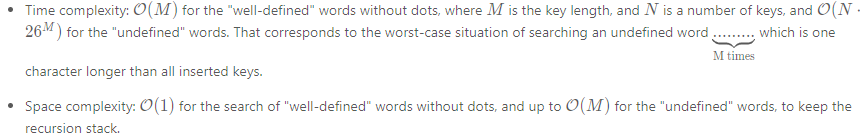


Figure 4. Search in trie.

|  |
| --- |
| **/\*\* Returns if the word is in the node. \*/**  **public boolean searchInNode(String word, TrieNode node) {**  **for (int i = 0; i < word.length(); ++i) {**  **char ch = word.charAt(i);**  **if (!node.children.containsKey(ch)) {**  **// if the current character is '.'**  **// check all possible nodes at this level**  **if (ch == '.') {**  **for (char x : node.children.keySet()) {**  **TrieNode child = node.children.get(x);**  **if (searchInNode(word.substring(i + 1), child)) {**  **return true;**  **}**  **}**  **}**  **// if no nodes lead to answer**  **// or the current character != '.'**  **return false;**  **} else {**  **// if the character is found**  **// go down to the next level in trie**  **node = node.children.get(ch);**  **}**  **}**  **return node.word;**  **}**  **/\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/**  **public boolean search(String word) {**  **return searchInNode(word, trie);**  **}** |

**Complexity Analysis**



**Implementation**

|  |
| --- |
| class TrieNode {  Map<Character, TrieNode> children = new HashMap();  boolean word = false;  public TrieNode() {}  }  class WordDictionary {  TrieNode trie;  /\*\* Initialize your data structure here. \*/  public WordDictionary() {  trie = new TrieNode();  }  /\*\* Adds a word into the data structure. \*/  public void addWord(String word) {  TrieNode node = trie;  for (char ch : word.toCharArray()) {  if (!node.children.containsKey(ch)) {  node.children.put(ch, new TrieNode());  }  node = node.children.get(ch);  }  node.word = true;  }  /\*\* Returns if the word is in the node. \*/  public boolean searchInNode(String word, TrieNode node) {  for (int i = 0; i < word.length(); ++i) {  char ch = word.charAt(i);  if (!node.children.containsKey(ch)) {  // if the current character is '.'  // check all possible nodes at this level  if (ch == '.') {  for (char x : node.children.keySet()) {  TrieNode child = node.children.get(x);  if (searchInNode(word.substring(i + 1), child)) {  return true;  }  }  }  // if no nodes lead to answer  // or the current character != '.'  return false;  } else {  // if the character is found  // go down to the next level in trie  node = node.children.get(ch);  }  }  return node.word;  }  /\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/  public boolean search(String word) {  return searchInNode(word, trie);  }  } |