<https://leetcode.com/problems/design-add-and-search-words-data-structure/>

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 <= 25
* word in addWord consists of lowercase English letters.
* word in search consist of '.' or lowercase English letters.
* There will be at most 2 dots in word for search queries.
* At most 104 calls will be made to addWord and search.

**Attempt 1: 2023-06-17**

**Wrong Solution:**

**Because of when search recursion happen, the passed in node cannot keep as 'root', needs to correspondingly passed in as its descendant child node**

Wrong solution: because of when search recursion happen, the passed in node cannot keep as 'root', needs to correspondingly passed in as its descendant child node

e.g

level 1 -> pass in node = root

level 2 -> pass in node = root.children[index1]

level 3 -> pass in node = root.children[index2].children[index2]

....

etc.

=========================================================================

e.g

Current trie has 3 words: "bad", "dad", "mad", search with ".ad"

Each recursion level two parameters relation represent as the string and its corresponding node pass to the same level

Level 1: search(".ad", root)

'.' of ".ad" mapping to root

.ad => root -> children[1,3,12], .(dot) mapping to b,d,m

b,d,m

Level 2: search("ad", root.children[1])

'a' of "ad" mapping to one of Level 1's node parameter(root)'s children(children[1],[3],[12]) as children[1]'s only child(children[0])

ad => child(children[1]) -> children[0], a mapping to a

b a

-------------------------------------------------------------------------

Current trie has 3 words: "bad", "dad", "mad", search with "..d"

Each recursion level two parameters relation represent as the string and its corresponding node pass to the same level

Level 1: search("..d", root)

'.' of "..d" mapping to root

..d => root -> children[1,3,12], .(dot) mapping to b,d,m

b,d,m

Level 2: search(".d", root.children[1])

'.' of ".d" mapping to one of Level 1's node parameter(root)'s children(children[1],[3],[12]) as children[1]'s only child(children[0])

.d => child(children[1]) -> children[0], . mapping to a

b a

Level 3: search("d", root.children[1].children[0])

'd' of "d" mapping to one of Level 2's node parameter(root.children[1].children[0]) as children[1].children[0]'s only child(children[3])

d => child(children[3]), d mapping to d

d

=========================================================================

class WordDictionary {

TrieNode root;

public WordDictionary() {

root = new TrieNode();

}

public void addWord(String word) {

TrieNode cur = root;

for(int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if(cur.children[index] == null) {

cur.children[index] = new TrieNode();

}

cur = cur.children[index];

}

cur.isEnd = true;

}

public boolean search(String word) {

TrieNode cur = root;

for(int i = 0; i < word.length(); i++) {

char c = word.charAt(i);

if(c == '.') {

// If character as dot, we need to check all children of the current node

for(TrieNode child : cur.children) {

// Skip dot and continue check remain substring

if(child != null && search(word.substring(i + 1))) {

return true;

}

}

return false;

} else {

// If character as normal lower English letters, we process as normal trie

int index = c - 'a';

if(cur.children[index] == null) {

return false;

}

cur = cur.children[index];

}

}

return cur != null && cur.isEnd;

}

}

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode() {

this.children = new TrieNode[26];

this.isEnd = false;

}

}

/\*\*

\* Your WordDictionary object will be instantiated and called as such:

\* WordDictionary obj = new WordDictionary();

\* obj.addWord(word);

\* boolean param\_2 = obj.search(word);

\*/

**Solution 1: Trie + DFS (60 min)**

**Style 1: For loop DFS**

class WordDictionary {

TrieNode root;

public WordDictionary() {

root = new TrieNode();

}

public void addWord(String word) {

TrieNode cur = root;

for(int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if(cur.children[index] == null) {

cur.children[index] = new TrieNode();

}

cur = cur.children[index];

}

cur.isEnd = true;

}

public boolean search(String word) {

return helper(word, root);

}

private boolean helper(String word, TrieNode root) {

TrieNode cur = root;

for(int i = 0; i < word.length(); i++) {

char c = word.charAt(i);

if(c == '.') {

// If character as dot, we need to check all children of the current node

for(TrieNode child : cur.children) {

// Skip dot and continue check remain substring

if(child != null && helper(word.substring(i + 1), child)) {

return true;

}

}

return false;

} else {

// If character as normal lower English letters, we process as normal trie

int index = c - 'a';

if(cur.children[index] == null) {

return false;

}

cur = cur.children[index];

}

}

return cur != null && cur.isEnd;

}

}

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode() {

this.children = new TrieNode[26];

this.isEnd = false;

}

}

/\*\*

\* Your WordDictionary object will be instantiated and called as such:

\* WordDictionary obj = new WordDictionary();

\* obj.addWord(word);

\* boolean param\_2 = obj.search(word);

\*/

**Refer to**

<https://leetcode.com/problems/design-add-and-search-words-data-structure/solutions/1725327/java-c-python-a-very-well-detailed-explanation/>

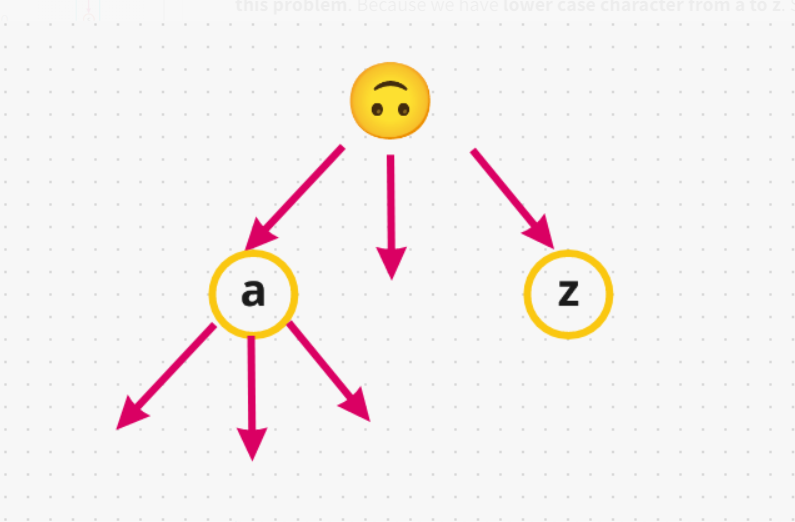
**So, to design a DS we only need to support 3 function's for this data structure.**

* One is the **constructor** that's gonna initialize the object.
* One is **Adding** the **word's**, but all of them are lower case from **a to z**
* One is **Searching** a **word**, and the word can contains any character from **a - z**. But there is **1 additional character** it contains, **"."** character [And what they told "." character is a wild card, can match any character easily in the string]

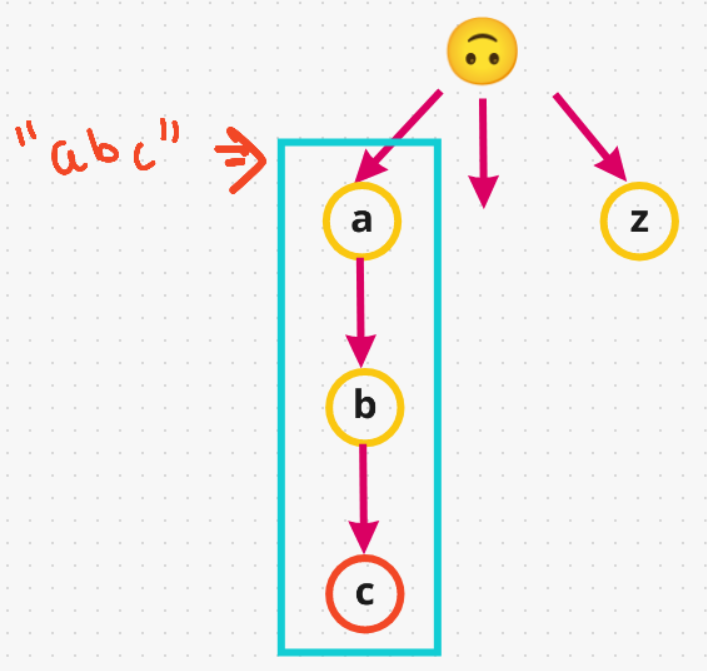
**The brute force way to solve this problem** is pretty simple, "*having a list of words & then just for every search we would just see, if this search match any of the word in input list*." **[But it's not an efficient way]**

Let's talk about, **efficient way to solve this problem**. And for that we require **Trie data structure** a.k.a **Prefix tree**

Let's understand **Trie first**, **Trie** is basically a **tree** that has some kind of **root node** & each node can have up to **26 children in this problem**. Because we have **lower case character from a to z**. So, basically each **node represents a single character**. And each node could have up to 26 additional children:



And basically word in this example means, let's take **"a"** having child **"b" -> with "c"**, so that's will be a **single word**. And if we insert the word **"abc"** in our **trie**, so basically how's it looks like:



One additional thing we have to say is, for a particular node such as **"c"** this is the **end of the word**. Because if we **added another word** example:- **"ab"**. So, we don't add them back again as if you notice they are already available to us, we just **gonna re-use** these characters. So, we have 2 word's along this **path "abc" & "ab"**

Basically, all words start with **"a"** are gonna be here, that's what it make it **efficient**. That's why it's called **prefix tree**.

Now, you have understood how Trie aka Prefix Tree work.

Now let's take an example & build our tree,

**Input**

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

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

**Output**

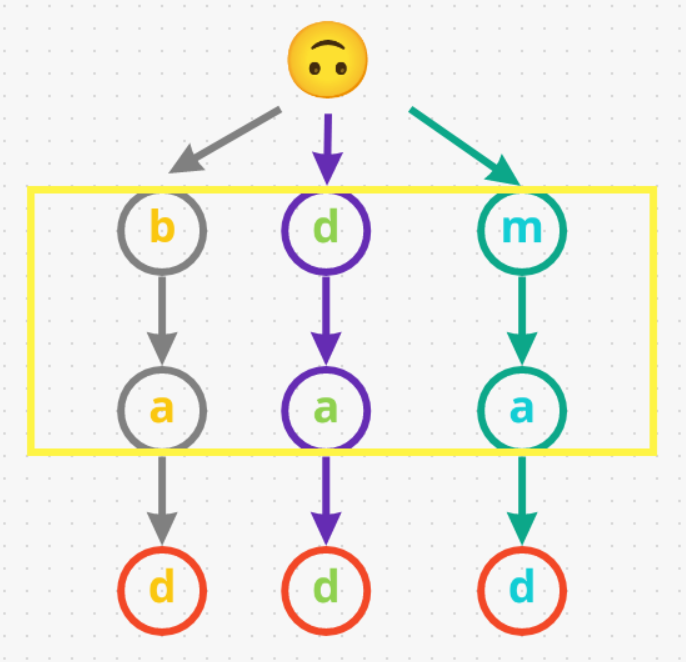
[null,null,null,null,false,true,true,true]

* The first word we add is **"bad". "b" -> "a" -> "d"**
* Next we adding another word **"dad"**. So, we have to start with **different path**. As, these two word's have a **different prefix**. One start with **"b"**and one start with **"d"**. So, let's add it: **"d" -> "a" -> "d"**
* We have one last word before we start searching, this one gonna be **"mad"**. So, we don't have **"m"**, then let's add it: **"m" -> "a" -> "d"**

**It’s a GIF file**



So far we have **3 word's** and all of them end with different **"d"**. But they all **3 of them have different prefix** that's why they are along different path's.



Now let's go for searching path:-

* So, first word we gonna search for **"pad"**. We gonna **start** at **beginning**. First we gonna try are there any **"p"** in this. That means immediately we **return false** as **"pad"** doesn't exist in our input.
* Now, we search another word **"bad"**. So, we **start at the root** and see there are any **"b"**. Yes, we have **"b"**. now we check does this **"b"** have a child **"a"**, yes it does. Now last character **"d"** does **"a"** has a child **"d"**, yes it does have. Lastly we have to say the last character **"d"** is in our **trie**, which designated as the end of the word. Since it's **marked red**. Therefore, we **return true** for this input **"bad"**
* Now, in this search we have **".ad"**, the dot **"."** character matches any character. So, we start at the root and go to any of the path and to do that we have to use **DFS or backtracking approach**. So, let's say first path we decided **"b"**. now we check does this **"b"** have a child **"a"**, yes it does. Now last character **"d"** does **"a"** has a child **"d"**, yes it does have. Lastly we have to say the last character **"d"** in our trie, is designated as the end of the word. Therefore, we **return true** for this input **".ad"**
* One last search, In this we have **"b.."**. So, we start at the root and see there are any **"b"**. Yes, we have **"b"**. Now we check do we have any character to go below for our current dot **"."**, yes we have **"a"**. Now we are looking for any character for our last **dot "."** Yes we have **"d"** & it is end of the word. Therefore, we **return true** for this input **"b.."**

Now understand this ***VISUALLY, it's not super hard.***

**It’s a GIF file**



**Now, let's code it up:**

**Java**

class WordDictionary {

private WordDictionary[] children;

boolean isEndOfWord;

// Initialize your data structure here.

public WordDictionary() {

children = new WordDictionary[26];

isEndOfWord = false;

}

// Adds a word into the data structure.

public void addWord(String word) {

WordDictionary curr = this;

for(char c: word.toCharArray()){

if(curr.children[c - 'a'] == null)

curr.children[c - 'a'] = new WordDictionary();

curr = curr.children[c - 'a'];

}

curr.isEndOfWord = true;

}

// 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) {

WordDictionary curr = this;

for(int i = 0; i < word.length(); ++i){

char c = word.charAt(i);

if(c == '.'){

for(WordDictionary ch: curr.children)

if(ch != null && ch.search(word.substring(i+1))) return true;

return false;

}

if(curr.children[c - 'a'] == null) return false;

curr = curr.children[c - 'a'];

}

return curr != null && curr.isEndOfWord;

}

}

ANALYSIS :-

* **Time Complexity :-** BigO(M) for well defined words, But in worse case BigO(M.26^N)
* **Space Complexity :-** BigO(1) for well defined words, But for worst case BigO(M)

**Complexity**: Easy part is space complexity, it is O(M), where M is sum of lengths of all words in our Trie. This is upper bound: in practice it will be less than M and it depends, how much words are intersected. The worst time complexity is also O(M), potentially we can visit all our Trie, if we have pattern like ...... For words without ., time complexity will be O(h), where h is height of Trie. For words with several letters and several ., we have something in the middle.

**Style 2: Standard DFS**

class WordDictionary {

TrieNode root;

public WordDictionary() {

root = new TrieNode();

}

public void addWord(String word) {

TrieNode cur = root;

for(int i = 0; i < word.length(); i++) {

int index = word.charAt(i) - 'a';

if(cur.children[index] == null) {

cur.children[index] = new TrieNode();

}

cur = cur.children[index];

}

cur.isEnd = true;

}

public boolean search(String word) {

return helper(word, 0, root);

}

private boolean helper(String word, int i, TrieNode node) {

if(i == word.length()) {

return node.isEnd;

}

char c = word.charAt(i);

if(c == '.') {

for(TrieNode child : node.children) {

if(child != null && helper(word, i + 1, child)) {

return true;

}

}

} else {

int index = c - 'a';

if(node.children[index] != null) {

if(helper(word, i + 1, node.children[index])) {

return true;

}

}

}

return false;

}

}

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode() {

this.children = new TrieNode[26];

this.isEnd = false;

}

}

/\*\*

\* Your WordDictionary object will be instantiated and called as such:

\* WordDictionary obj = new WordDictionary();

\* obj.addWord(word);

\* boolean param\_2 = obj.search(word);

\*/

**Refer to**

<https://leetcode.com/problems/design-add-and-search-words-data-structure/solutions/59554/my-simple-and-clean-java-code/>

public class WordDictionary {

public class TrieNode {

public TrieNode[] children = new TrieNode[26];

public String item = "";

}

private TrieNode root = new TrieNode();

public void addWord(String word) {

TrieNode node = root;

for (char c : word.toCharArray()) {

if (node.children[c - 'a'] == null) {

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.item = word;

}

public boolean search(String word) {

return match(word.toCharArray(), 0, root);

}

private boolean match(char[] chs, int k, TrieNode node) {

if (k == chs.length) return !node.item.equals("");

if (chs[k] != '.') {

return node.children[chs[k] - 'a'] != null && match(chs, k + 1, node.children[chs[k] - 'a']);

} else {

for (int i = 0; i < node.children.length; i++) {

if (node.children[i] != null) {

if (match(chs, k + 1, node.children[i])) {

return true;

}

}

}

}

return false;

}

}

<https://leetcode.com/problems/design-add-and-search-words-data-structure/solutions/59718/easy-to-understand-java-solution-using-trie-and-recursion-with-explanation/>

public class WordDictionary {

private class TrieNode {

private boolean isWord;

private HashMap<Character, TrieNode> childList;

public TrieNode() {

isWord = false;

childList = new HashMap<Character, TrieNode>();

}

}

private TrieNode root = new TrieNode();

// Adds a word into the data structure.

public void addWord(String word) {

TrieNode curr = root;

for (int i = 0; i < word.length(); i++) {

if (!curr.childList.containsKey(word.charAt(i))) {

curr.childList.put(word.charAt(i), new TrieNode());

}

curr = curr.childList.get(word.charAt(i));

}

curr.isWord = true;

}

// 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 searchHelper(root, 0, word);

}

private boolean searchHelper(TrieNode node, int pos, String word) {

//if the word has all been scanned, return

if (pos == word.length()) {

return node.isWord;

}

//reach the leaf before finishing scanning the word

if (node.childList.size() == 0) {

return false;

}

//if the character at current position is '.',

//recursive check whether the remaing word is in the trie

if (word.charAt(pos) == '.') {

for (Character c : node.childList.keySet()) {

if (searchHelper(node.childList.get(c), pos + 1, word)) {

return true;

}

}

}

//if character at position 'pos' is neither equal to the node nor '.', return false

if (!node.childList.containsKey(word.charAt(pos))) {

return false;

}

//if character at current position matches the node,

//recursively search the remaining word

return searchHelper(node.childList.get(word.charAt(pos)), pos + 1, word);

}

}

<https://leetcode.com/problems/design-add-and-search-words-data-structure/solutions/3313638/java-c-simple-solution-easy-to-understand/>

class TrieNode {

Map<Character, TrieNode> children;

boolean isWord;

public TrieNode() {

children = new HashMap<>();

isWord = false;

}

}

class WordDictionary {

private TrieNode root;

public WordDictionary() {

root = new TrieNode();

}

public void addWord(String word) {

TrieNode node = root;

// Traverse the trie for each character in the word

for (int i = 0; i < word.length(); i++) {

char c = word.charAt(i);

// If the current node does not have a child with character c,

// create a new node and add it as a child of the current node

if (!node.children.containsKey(c)) {

node.children.put(c, new TrieNode());

}

// Move to the child node corresponding to character c

node = node.children.get(c);

}

// Mark the last node as a word node

node.isWord = true;

}

public boolean search(String word) {

return searchHelper(root, word, 0);

}

private boolean searchHelper(TrieNode node, String word, int index) {

// If we have reached the end of the word,

// check if the current node is a word node

if (index == word.length()) {

return node.isWord;

}

char c = word.charAt(index);

if (c == '.') {

// If the current character is a dot, we need to check all children of the current node

// recursively by skipping over the dot character and moving to the next character of the word

for (TrieNode child : node.children.values()) {

if (searchHelper(child, word, index + 1)) {

return true;

}

}

// If no child node matches the remaining characters of the word,

// return false

return false;

} else {

// If the current character is not a dot, move to the child node

// corresponding to that character and continue recursively

TrieNode child = node.children.get(c);

if (child == null) {

// If there is no child node corresponding to the current character,

// return false

return false;

}

return searchHelper(child, word, index + 1);

}

}

}