**Problem Statement: Word Dictionary**

You are developing a word dictionary application that supports adding words and searching for words with wildcard support. The wildcard character . can represent any one letter.

**Input Format:**

1. The first line contains an integer n denoting the number of operations.
2. The next n lines contain operations in the format:
   1. addWord word: To add the word to the dictionary.
   2. search word: To search for the word in the dictionary. The word can contain the wildcard character . which can match any single letter.

**Output Format:** For each search operation, print "Found" if the word (including any wildcards) matches any word in the dictionary. Otherwise, print "Not Found".

**Constraints:**

* The number of operations n is in the range [1, 10000].
* Each word consists of lowercase English letters and the wildcard character ..
* All words and search queries have a length in the range [1, 100].

**Sample Input:**

5

addWord bad

addWord dad

addWord mad

search pad

search b.d

**Sample Output:**

Not Found

Found

**Explanation:**

* The first operation adds the word bad to the dictionary.
* The second operation adds the word dad to the dictionary.
* The third operation adds the word mad to the dictionary.
* The fourth operation searches for pad, which is not in the dictionary.
* The fifth operation searches for b.d, which matches the word bad in the dictionary.

**Hints:**

1. Use a trie (prefix tree) data structure to efficiently support the operations.
2. For the search operation with the wildcard character .:
   * Use a depth-first search (DFS) approach to explore all possible paths in the trie.
   * For each . in the word, try all possible children nodes.

**Solution Template:**

python

class TrieNode:

def \_\_init\_\_(self):

self.children = {}

self.is\_end\_of\_word = False

class WordDictionary:

def \_\_init\_\_(self):

self.root = TrieNode()

def addWord(self, word: str) -> None:

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

def search(self, word: str) -> bool:

return self.\_search\_in\_node(word, self.root)

def \_search\_in\_node(self, word: str, node: TrieNode) -> bool:

for i, char in enumerate(word):

if char == '.':

for child in node.children.values():

if self.\_search\_in\_node(word[i+1:], child):

return True

return False

elif char in node.children:

node = node.children[char]

else:

return False

return node.is\_end\_of\_word

def main():

import sys

input = sys.stdin.read

data = input().strip().split('\n')

n = int(data[0])

word\_dictionary = WordDictionary()

results = []

for i in range(1, n + 1):

command = data[i].split()

if command[0] == "addWord":

word\_dictionary.addWord(command[1])

elif command[0] == "search":

found = word\_dictionary.search(command[1])

results.append("Found" if found else "Not Found")

for result in results:

print(result)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Extra Test Cases:**

**Test Case 1:**

6

addWord apple

addWord application

addWord apply

search a..le

search apl.

search .pplication

**Output:**

Found

Not Found

Found

**Test Case 2:**

5

addWord book

addWord notebook

addWord look

search b.ok

search n..ebook

**Output:**

Found

Found

**Test Case 3:**

4

addWord cat

addWord bat

search .at

search c.t

**Output:**

Found

Found

**Test Case 4:**

5

addWord home

addWord homework

addWord house

search ho.e

search h.se

**Output:**

Found

Not Found

**Test Case 5:**

6

addWord garden

addWord gardener

addWord gardening

search g.rd.n

search g.rden

search gard..ning

**Output:**

Found

Found

Not Found