**Problem Statement: Contact List**

You are building a contact list application that needs to support adding contacts and searching for contacts by prefix. Each contact consists of a single word, representing a name. Implement a trie data structure that supports the following operations:

1. **Add Contact**: Add a contact to the list.
2. **Find Contacts by Prefix**: Find all contacts that start with a given prefix.

**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:
   * add contact\_name: To add the contact name to the contact list.
   * find prefix: To find and print all contacts that start with the given prefix. If no contacts match, print "No contacts found".

**Output Format:** For each find operation, print all matching contacts in lexicographical order. If no contacts match, print "No contacts found".

**Constraints:**

* The number of operations n is in the range [1, 10000].
* Each contact name and prefix consist of lowercase English letters only.
* All contact names and prefixes have a length in the range [1, 100].

**Sample Input:**

5

add alice

add bob

add alicea

find ali

find b

**Sample Output:**

alice

alicea

bob

**Explanation:**

* The first operation adds alice to the contact list.
* The second operation adds bob to the contact list.
* The third operation adds alicea to the contact list.
* The fourth operation finds all contacts starting with ali, which are alice and alicea.
* The fifth operation finds all contacts starting with b, which is bob.

**Hints:**

1. Use a trie (prefix tree) data structure to efficiently support the operations.
2. For each node in the trie, keep a map of child nodes corresponding to the next character in the contact name or prefix.
3. Perform a depth-first search (DFS) from the node corresponding to the last character of the prefix to collect all matching contacts.

**Solution Template:**

python

class TrieNode:

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

self.children = {}

self.is\_end\_of\_word = False

self.contacts = []

class Trie:

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

self.root = TrieNode()

def add\_contact(self, name: str) -> None:

node = self.root

for char in name:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.contacts.append(name)

node.is\_end\_of\_word = True

def find\_contacts(self, prefix: str):

node = self.root

for char in prefix:

if char not in node.children:

return []

node = node.children[char]

return sorted(node.contacts)

def main():

import sys

input = sys.stdin.read

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

n = int(data[0])

trie = Trie()

results = []

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

command = data[i].split()

if command[0] == "add":

trie.add\_contact(command[1])

elif command[0] == "find":

contacts = trie.find\_contacts(command[1])

if contacts:

results.extend(contacts)

else:

results.append("No contacts found")

for result in results:

print(result)

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

main()

**Extra Test Cases:**

**Test Case 1:**

arduino

6

add alice

add alicia

add bob

find ali

find al

find b

**Output:**

alice

alicia

alice

alicia

bob

**Test Case 2:**

arduino

4

add john

add jack

find j

find jo

**Output:**

jack

john

john

**Test Case 3:**

5

add sam

add samantha

add samuel

find sam

find samantha

**Output:**

sam

samantha

samuel

samantha

**Test Case 4:**

7

add adam

add addison

add brad

add bradley

find ad

find b

find bra

**Output:**

adam

addison

brad

bradley

brad

bradley

**Test Case 5:**

5

add mike

add mick

add michelle

find mic

find mi

**Output:**

mick

michelle

mick

michelle

mike