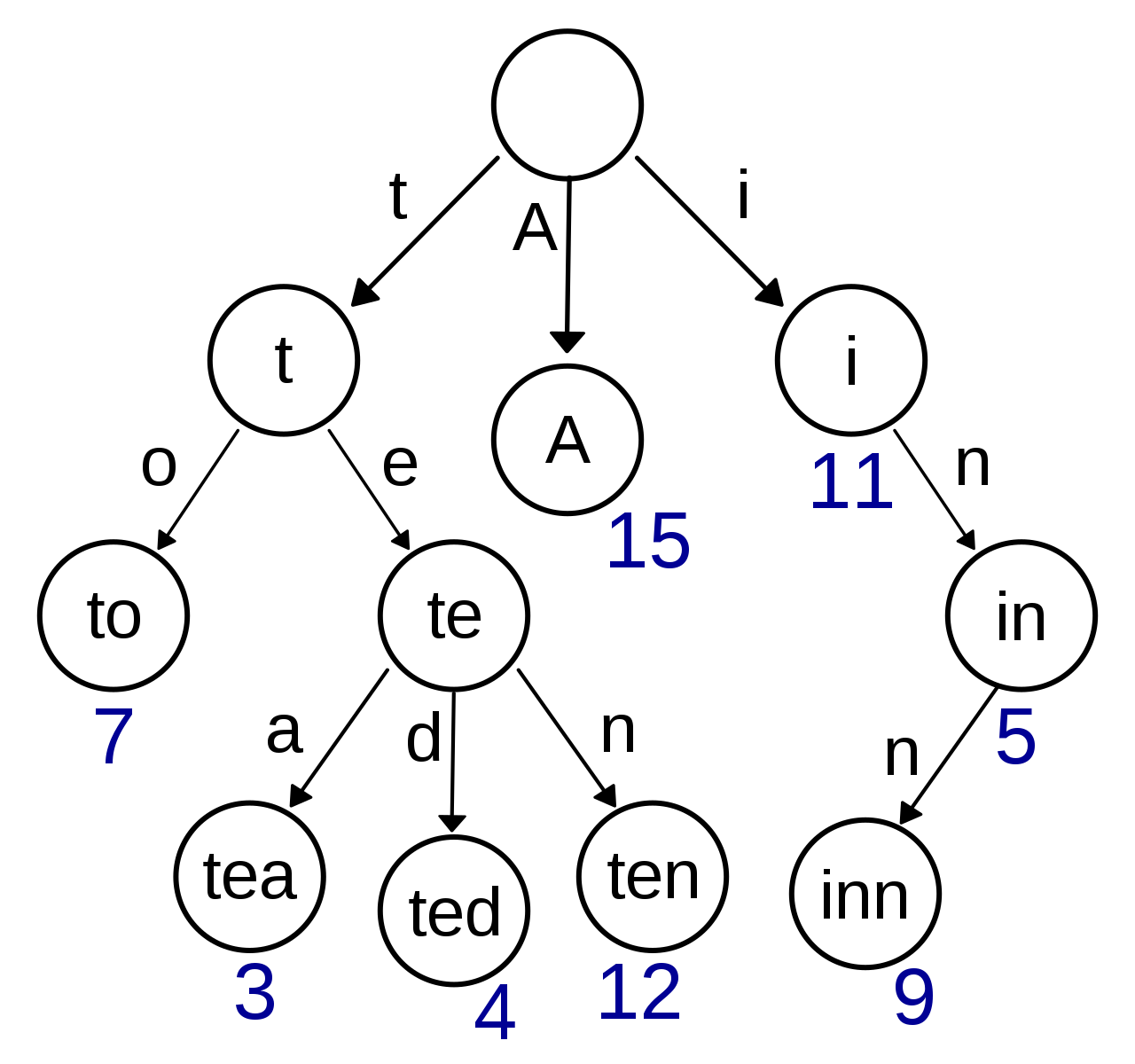
LeetCode Training Day 21 Trie

Trie is a data structure used to store a dictionary in a compressed mode. A dictionary contains a lot of words, and many words have the same prefix. The Trie is to store the common prefix in the same storage, in this way it does not only save the storage, but also speed up the search if you want to count all words with similar prefix.

A Trie data structure looks like below:



The Trie data structure is a linked list in tree structure, from a dummy root, then under each node, we will have a list of nodes with leading next character in the word.

Rember from root to any node it can be a complete word, or it can be a prefix of some words. If it is an complete word we will mark it here by either an is\_end flag or a word count to say how many words (duplicates) ending here.

Ideally the children of the Trie Node should be a hash table with next character as key. But in LeetCode context, we normally talk about lower case English words, so it can be reduced as a array of TrieNode pointer with capacity of 26.

Trie normally has two operations, one is to insert a new word, another is to search any word. An empty Trie start with dummy root, with all 26 children pointing to null, which means we do not have any words leading with such character, when a new word is inserted, we will start to create the Trie node in that position. (starting ‘a’ will be put in slot 0 and ‘z’ will be put in slot 25), we repeat doing so until we reach to the end of word. The insert and search can be two native function coming from Trie class implementation with recursive call.

In the LeetCode Trie problem, many of them can be solved in another way, which is called prefix hashmap, we add the word to all its prefix hashtable, so we can search the words by prefix in O(1) operation. The disadvantage is that a lot of memory space is wasted.

## 208. Implement Trie (Prefix Tree)

Medium

A [**trie**](https://en.wikipedia.org/wiki/Trie) (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

* Trie() Initializes the trie object.
* void insert(String word) Inserts the string word into the trie.
* boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
* boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

**Example 1:**

**Input**

["Trie", "insert", "search", "search", "startsWith", "insert", "search"]

[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]

**Output**

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

**Explanation**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // return True

trie.search("app"); // return False

trie.startsWith("app"); // return True

trie.insert("app");

trie.search("app"); // return True

**Constraints:**

* 1 <= word.length, prefix.length <= 2000
* word and prefix consist only of lowercase English letters.
* At most 3 \* 104 calls **in total** will be made to insert, search, and startsWith.

### Analysis:

This is a typical Trie implementation.

/// <summary>

/// Leet code #208 Implement Trie (Prefix Tree)

///

/// Medium

///

/// Implement a trie with insert, search, and startsWith methods.

///

/// Example:

///

/// Trie trie = new Trie();

///

/// trie.insert("apple");

/// trie.search("apple"); // returns true

/// trie.search("app"); // returns false

/// trie.startsWith("app"); // returns true

/// trie.insert("app");

/// trie.search("app"); // returns true

/// Note:

///

/// 1. You may assume that all inputs are consist of lowercase letters a-z.

/// 2. All inputs are guaranteed to be non-empty strings.

/// </summary>

class Trie

{

private:

bool is\_end;

vector<Trie \*> children;

public:

Trie()

{

children = vector<Trie \*>(26, nullptr);

is\_end = false;

};

~Trie()

{

for (size\_t i = 0; i < children.size(); i++)

{

if (children[i] != nullptr) delete children[i];

}

}

void insert(string word)

{

if (word.empty())

{

is\_end = true;

}

else

{

int i = word[0] - 'a';

if (children[i] == nullptr)

{

children[i] = new Trie();

}

children[i]->insert(word.substr(1));

}

};

bool search(string word)

{

if (word.empty())

{

return is\_end;

}

else

{

int i = word[0] - 'a';

if (children[i] == nullptr)

{

return false;

}

return children[i]->search(word.substr(1));

}

}

bool startsWith(string prefix)

{

if (prefix.empty())

{

return true;

}

else

{

int i = prefix[0] - 'a';

if (children[i] == nullptr)

{

return false;

}

return children[i]->startsWith(prefix.substr(1));

}

};

};

## 211. Design Add and Search Words Data Structure

Medium

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.

### Analysis:

It is typical example of Trie implementation. Please watch in this problem if you do destructor it will TLE ( do not understand it)

/// <summary>

/// Leet code #211. Add and Search Word - Data structure design

/// Design a data structure that supports the following two operations:

/// void addWord(word)

/// bool search(word)

///

/// search(word) can search a literal word or a regular expression string containing only

/// letters a-z or .. A . means it can represent any one letter.

///

/// For example:

/// addWord("bad")

/// addWord("dad")

/// addWord("mad")

/// search("pad") -> false

/// search("bad") -> true

/// search(".ad") -> true

/// search("b..") -> true

///

/// Note:

/// You may assume that all words are consist of lowercase letters a-z.

///

/// click to show hint.

/// You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

/// Your WordDictionary object will be instantiated and called as such:

/// WordDictionary wordDictionary;

/// wordDictionary.addWord("word");

/// wordDictionary.search("pattern");

/// </summary>

class WordDictionary

{

private:

bool m\_end;

vector<WordDictionary\*> m\_children;

public:

/// <summary>

/// Constructor an empty WordDictionary

/// </summary>

/// <returns></returns>

WordDictionary()

{

m\_end = false;

m\_children = vector<WordDictionary\*>(26, nullptr);

};

/// <summary>

/// Destructor of an WordDictionary

/// </summary>

/// <returns></returns>

~WordDictionary()

{

// Leet code does not like my destructor so it cause TLE.

//for (int i = 0; i < 26; i++)

//{

// if (m\_children[i] == nullptr) delete m\_children[i];

//}

}

/// <summary>

/// Adds a word into the data structure.

/// </summary>

/// <param name="word>The word</param>

/// <returns></returns>

void addWord(string word, int p = 0)

{

if (p == word.size())

{

m\_end = true;

return;

}

if (m\_children[word[p] - 'a'] == nullptr)

{

m\_children[word[p] - 'a'] = new WordDictionary();

}

WordDictionary \* next = m\_children[word[p] - 'a'];

next->addWord(word, p + 1);

}

/// <summary>

// Returns if the word is in the data structure. A word could

// contain the dot character '.' to represent any one letter.

/// </summary>

/// <param name="word">The word</param>

/// <returns>true, if found</returns>

bool search(string word, int p = 0)

{

if (p == word.size())

{

return m\_end;

}

if (word[p] != '.')

{

if (m\_children[word[p] - 'a'] == nullptr)

{

return false;

}

else

{

return m\_children[word[p] - 'a']->search(word, p + 1);

}

}

else

{

bool ret = false;

for (int i = 0; i < 26; i++)

{

if (m\_children[i] != nullptr)

{

ret = m\_children[i]->search(word, p + 1);

}

if (ret == true) break;

}

return ret;

}

}

};

## 1804. Implement Trie II (Prefix Tree)

Medium

A [**trie**](https://en.wikipedia.org/wiki/Trie) (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

* Trie() Initializes the trie object.
* void insert(String word) Inserts the string word into the trie.
* int countWordsEqualTo(String word) Returns the number of instances of the string word in the trie.
* int countWordsStartingWith(String prefix) Returns the number of strings in the trie that have the string prefix as a prefix.
* void erase(String word) Erases the string word from the trie.

**Example 1:**

**Input**

["Trie", "insert", "insert", "countWordsEqualTo", "countWordsStartingWith", "erase", "countWordsEqualTo", "countWordsStartingWith", "erase", "countWordsStartingWith"]

[[], ["apple"], ["apple"], ["apple"], ["app"], ["apple"], ["apple"], ["app"], ["apple"], ["app"]]

**Output**

[null, null, null, 2, 2, null, 1, 1, null, 0]

**Explanation**

Trie trie = new Trie();

trie.insert("apple"); // Inserts "apple".

trie.insert("apple"); // Inserts another "apple".

trie.countWordsEqualTo("apple"); // There are two instances of "apple" so return 2.

trie.countWordsStartingWith("app"); // "app" is a prefix of "apple" so return 2.

trie.erase("apple"); // Erases one "apple".

trie.countWordsEqualTo("apple"); // Now there is only one instance of "apple" so return 1.

trie.countWordsStartingWith("app"); // return 1

trie.erase("apple"); // Erases "apple". Now the trie is empty.

trie.countWordsStartingWith("app"); // return 0

**Constraints:**

* 1 <= word.length, prefix.length <= 2000
* word and prefix consist only of lowercase English letters.
* At most 3 \* 104 calls **in total** will be made to insert, countWordsEqualTo, countWordsStartingWith, and erase.
* It is guaranteed that for any function call to erase, the string word will exist in the trie.

### Analysis:

It is typical example of Trie implementation. Each Trie node will have two count, prefix count and word count.

/// <summary>

/// Leet Code 1804. Implement Trie II (Prefix Tree)

///

/// Medium

///

///

/// A trie (pronounced as "try") or prefix tree is a tree data structure

/// used to efficiently store and retrieve keys in a dataset of strings.

/// There are various applications of this data structure, such as

/// autocomplete and spellchecker.

///

/// Implement the Trie class:

///

/// Trie() Initializes the trie object.

/// void insert(String word) Inserts the string word into the trie.

/// int countWordsEqualTo(String word) Returns the number of instances

/// of the string word in the trie.

/// int countWordsStartingWith(String prefix)

/// Returns the number of strings in the trie that have the

/// string prefix as a prefix.

/// void erase(String word) Erases the string word from the trie.

///

/// Example 1:

/// Input

/// ["Trie", "insert", "insert", "countWordsEqualTo",

/// "countWordsStartingWith", "erase", "countWordsEqualTo",

/// "countWordsStartingWith", "erase", "countWordsStartingWith"]

/// [[], ["apple"], ["apple"], ["apple"], ["app"], ["apple"], ["apple"],

/// ["app"], ["apple"], ["app"]]

/// Output

/// [null, null, null, 2, 2, null, 1, 1, null, 0]

///

/// Explanation

/// Trie trie = new Trie();

/// trie.insert("apple"); // Inserts "apple".

/// trie.insert("apple"); // Inserts another "apple".

/// trie.countWordsEqualTo("apple"); // There are two instances of

/// // "apple" so return 2.

/// trie.countWordsStartingWith("app"); // "app" is a prefix of "apple"

/// // so return 2.

/// trie.erase("apple"); // Erases one "apple".

/// trie.countWordsEqualTo("apple"); // Now there is only one instance

/// // of "apple" so return 1.

/// trie.countWordsStartingWith("app"); // return 1

/// trie.erase("apple"); // Erases "apple". Now the trie is

/// // empty.

/// trie.countWordsStartingWith("app"); // return 0

///

/// Constraints:

/// 1. 1 <= word.length, prefix.length <= 2000

/// 2. word and prefix consist only of lowercase English letters.

/// 3. At most 3 \* 104 calls in total will be made to insert,

/// countWordsEqualTo, countWordsStartingWith, and erase.

/// 4. It is guaranteed that for any function call to erase, the string

/// word will exist in the trie.

/// </summary>

class TrieII

{

private:

int m\_words;

int m\_prefix;

vector<TrieII\*> m\_children;

public:

TrieII()

{

m\_words = 0;

m\_prefix = 0;

m\_children = vector<TrieII\*>(26);

}

void insert(string word, int index = 0)

{

m\_prefix++;

if (index == word.size())

{

m\_words++;

return;

}

else

{

if (m\_children[word[index] - 'a'] == nullptr)

{

m\_children[word[index] - 'a'] = new TrieII();

}

m\_children[word[index] - 'a']->insert(word, index + 1);

}

}

int countWordsEqualTo(string word, int index = 0)

{

if (index == word.size())

{

return m\_words;

}

else

{

if (m\_children[word[index] - 'a'] == nullptr)

{

return 0;

}

else

{

return m\_children[word[index] - 'a']->countWordsEqualTo(word, index + 1);

}

}

}

int countWordsStartingWith(string prefix, int index = 0)

{

if (index == prefix.size())

{

return m\_prefix;

}

else

{

if (m\_children[prefix[index] - 'a'] == nullptr)

{

return 0;

}

else

{

return m\_children[prefix[index] - 'a']->countWordsStartingWith(prefix, index + 1);

}

}

}

void erase(string word, int index = 0)

{

m\_prefix--;

if (index == word.size())

{

m\_words--;

return;

}

else

{

if (m\_children[word[index] - 'a'] == nullptr)

{

return;

}

m\_children[word[index] - 'a']->erase(word, index + 1);

}

}

};

## 648. Replace Words

Medium

In English, we have a concept called **root**, which can be followed by some other word to form another longer word - let's call this word **successor**. For example, when the **root** "an" is followed by the **successor** word "other", we can form a new word "another".

Given a dictionary consisting of many **roots** and a sentence consisting of words separated by spaces, replace all the **successors** in the sentence with the **root** forming it. If a **successor** can be replaced by more than one **root**, replace it with the **root** that has **the shortest length**.

Return *the sentence* after the replacement.

**Example 1:**

**Input:** dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by the battery"

**Output:** "the cat was rat by the bat"

**Example 2:**

**Input:** dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"

**Output:** "a a b c"

**Constraints:**

* 1 <= dictionary.length <= 1000
* 1 <= dictionary[i].length <= 100
* dictionary[i] consists of only lower-case letters.
* 1 <= sentence.length <= 106
* sentence consists of only lower-case letters and spaces.
* The number of words in sentence is in the range [1, 1000]
* The length of each word in sentence is in the range [1, 1000]
* Every two consecutive words in sentence will be separated by exactly one space.
* sentence does not have leading or trailing spaces.

### Analysis:

This problem can use Trie, but to make it easier it should use prefix hash table, and check each word to see if prefix is in hash set.

/// <summary>

/// Leet code #648. Replace Words

/// </summary>

string LeetCodeString::replaceWord(unordered\_map<int, unordered\_set<string>>&dict\_map, const string&word)

{

string result = word;

for (size\_t i = 0; i < word.size(); i++)

{

string sub\_word = word.substr(0, i + 1);

if (dict\_map[sub\_word.size()].count(sub\_word) > 0)

{

result = sub\_word;

break;

}

}

return result;

}

/// <summary>

/// Leet code #648. Replace Words

///

/// In English, we have a concept called root, which can be followed by

/// some other words to form another longer word - let's call this word

/// successor. For example, the root an, followed by other, which can

/// form another word another.

/// Now, given a dictionary consisting of many roots and a sentence. You

/// need to replace all the successor in the sentence with the root forming

/// it. If a successor has many roots can form it, replace it with the root

/// with the shortest length.

/// You need to output the sentence after the replacement.

/// Example 1:

/// Input: dict = ["cat", "bat", "rat"]

/// sentence = "the cattle was rattled by the battery"

/// Output: "the cat was rat by the bat"

///

/// Note:

/// 1. The input will only have lower-case letters.

/// 2. 1 <= dict words number <= 1000

/// 3. 1 <= sentence words number <= 1000

/// 4. 1 <= root length <= 100

/// 5. 1 <= sentence words length <= 1000

/// </summary>

string LeetCodeString::replaceWords(vector<string>& dict, string sentence)

{

string result;

unordered\_map<int, unordered\_set<string>> dict\_map;

for (size\_t i = 0; i < dict.size(); i++)

{

dict\_map[dict[i].size()].insert(dict[i]);

}

int first = 0, last = 0;

while (last <= (int)sentence.size())

{

if (last == sentence.size() || isspace(sentence[last]))

{

if (first < last)

{

if (!result.empty()) result.push\_back(' ');

string word = replaceWord(dict\_map, sentence.substr(first, last - first));

result.append(word);

}

first = last + 1;

}

last++;

}

return result;

}

## 720. Longest Word in Dictionary

Medium

Given an array of strings words representing an English Dictionary, return *the longest word in* words *that can be built one character at a time by other words in* words.

If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.

**Example 1:**

**Input:** words = ["w","wo","wor","worl","world"]

**Output:** "world"

**Explanation:** The word "world" can be built one character at a time by "w", "wo", "wor", and "worl".

**Example 2:**

**Input:** words = ["a","banana","app","appl","ap","apply","apple"]

**Output:** "apple"

**Explanation:** Both "apply" and "apple" can be built from other words in the dictionary. However, "apple" is lexicographically smaller than "apply".

**Constraints:**

* 1 <= words.length <= 1000
* 1 <= words[i].length <= 30
* words[i] consists of lowercase English letters.

### Analysis:

Sort the dictionary by word length and lexicon order, process the shortest words first. For any word check if first n-1 substring in dictionary, if not skip the word.

/// <summary>

/// Leet code #720. Longest Word in Dictionary

///

/// Given a list of strings words representing an English Dictionary, find

/// the longest word in words that can be built one character at a time by

/// other words in words. If there is more than one possible answer,

/// return the longest word with the smallest lexicographical order.

///

/// If there is no answer, return the empty string.

/// Example 1:

/// Input:

/// words = ["w","wo","wor","worl", "world"]

/// Output: "world"

/// Explanation:

/// The word "world" can be built one character at a time by "w", "wo",

/// "wor", and "worl".

///

/// Example 2:

/// Input:

/// words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

/// Output: "apple"

/// Explanation:

/// Both "apply" and "apple" can be built from other words in the

/// dictionary. However, "apple" is lexicographically smaller than

/// "apply".

///

/// Note:

///

/// All the strings in the input will only contain lowercase letters.

/// The length of words will be in the range [1, 1000].

/// The length of words[i] will be in the range [1, 30].

/// </summary>

string LeetCodeString::longestWord(vector<string>& words)

{

set<pair<int, string>> pq;

unordered\_set<string> word\_map;

word\_map.insert("");

for (size\_t i = 0; i < words.size(); i++)

{

pq.insert(make\_pair((int)words[i].size(), words[i]));

}

string result;

while (!pq.empty())

{

pair<int, string> pair = \*pq.begin();

pq.erase(pq.begin());

if (pair.first > (int)result.size() + 1) break;

if (word\_map.count(pair.second.substr(0, pair.first - 1)) == 0)

{

continue;

}

word\_map.insert(pair.second);

// longer word found

if (pair.first > (int)result.size())

{

result = pair.second;

}

}

return result;

}

## 1858. Longest Word With All Prefixes

Medium

Given an array of strings words, find the **longest** string in words such that **every prefix** of it is also in words.

* For example, let words = ["a", "app", "ap"]. The string "app" has prefixes "ap" and "a", all of which are in words.

Return *the string described above. If there is more than one string with the same length, return the****lexicographically smallest****one, and if no string exists, return*"".

**Example 1:**

**Input:** words = ["k","ki","kir","kira", "kiran"]

**Output:** "kiran"

**Explanation:** "kiran" has prefixes "kira", "kir", "ki", and "k", and all of them appear in words.

**Example 2:**

**Input:** words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

**Output:** "apple"

**Explanation:** Both "apple" and "apply" have all their prefixes in words.

However, "apple" is lexicographically smaller, so we return that.

**Example 3:**

**Input:** words = ["abc", "bc", "ab", "qwe"]

**Output:** ""

**Constraints:**

* 1 <= words.length <= 105
* 1 <= words[i].length <= 105
* 1 <= sum(words[i].length) <= 105

### Analysis:

Sort the dictionary by word length and lexicon order, process the shortest words first. For any word check if first n-1 substring in dictionary, if not skip the word. Same as above

/// <summary>

/// Leet Code 1858. Longest Word With All Prefixes

///

/// Medium

///

/// Given an array of strings words, find the longest string in words such

/// that every prefix of it is also in words.

///

/// For example, let words = ["a", "app", "ap"]. The string "app" has

/// prefixes "ap" and "a", all of which are in words.

/// Return the string described above. If there is more than one string

/// with the same length, return the lexicographically smallest one, and

/// if no string exists, return "".

///

/// Example 1:

/// Input: words = ["k","ki","kir","kira", "kiran"]

/// Output: "kiran"

/// Explanation: "kiran" has prefixes "kira", "kir", "ki", and "k", and

/// all of them appear in words.

///

/// Example 2:

/// Input: words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

/// Output: "apple"

/// Explanation: Both "apple" and "apply" have all their prefixes in words.

/// However, "apple" is lexicographically smaller, so we return that.

///

/// Example 3:

/// Input: words = ["abc", "bc", "ab", "qwe"]

/// Output: ""

///

/// Constraints:

/// 1. 1 <= words.length <= 10^5

/// 2. 1 <= words[i].length <= 10^5

/// 3. 1 <= sum(words[i].length) <= 10^5

/// </summary>

string LeetCodeString::longestWordII(vector<string>& words)

{

map<int, vector<int>> word\_map;

unordered\_set<string> hash\_set;

for (size\_t i = 0; i < words.size(); i++)

{

word\_map[words[i].size()].push\_back(i);

}

hash\_set.insert("");

int length = 0;

string result;

while (true)

{

length++;

bool found = false;

for (auto i : word\_map[length])

{

if (hash\_set.count(words[i].substr(0, words[i].size() - 1)) > 0)

{

found = true;

hash\_set.insert(words[i]);

if (length > (int)result.size() || words[i] < result)

{

result = words[i];

}

}

}

if (found == false) break;

}

return result;

}

## 677. Map Sum Pairs

Medium

Design a map that allows you to do the following:

* Maps a string key to a given value.
* Returns the sum of the values that have a key with a prefix equal to a given string.

Implement the MapSum class:

* MapSum() Initializes the MapSum object.
* void insert(String key, int val) Inserts the key-val pair into the map. If the key already existed, the original key-value pair will be overridden to the new one.
* int sum(string prefix) Returns the sum of all the pairs' value whose key starts with the prefix.

**Example 1:**

**Input**

["MapSum", "insert", "sum", "insert", "sum"]

[[], ["apple", 3], ["ap"], ["app", 2], ["ap"]]

**Output**

[null, null, 3, null, 5]

**Explanation**

MapSum mapSum = new MapSum();

mapSum.insert("apple", 3);

mapSum.sum("ap"); // return 3 (apple = 3)

mapSum.insert("app", 2);

mapSum.sum("ap"); // return 5 (apple + app = 3 + 2 = 5)

**Constraints:**

* 1 <= key.length, prefix.length <= 50
* key and prefix consist of only lowercase English letters.
* 1 <= val <= 1000
* At most 50 calls will be made to insert and sum.

### Analysis:

It is nothing wrong to build the word count in a Trie, but that will make the code a little bit complicated. You can do it simpler. Simply build the word count in a Tree map, the use binary search to find the first word no less than the prefix, for example from lexicon, "ap" < "app” < "apple" < “banana". Then keep on iterate the words until you see the prefix from the word does not match, which means you already go too far (for example you reach "banana").

/// <summary>

/// Leet code #677. Map Sum Pairs

///

/// Implement a MapSum class with insert, and sum methods.

///

/// For the method insert, you'll be given a pair of (string, integer).

/// The string represents the key and the integer represents the value.

/// If the key already existed, then the original key-value pair will be

/// overridden to the new one.

///

/// For the method sum, you'll be given a string representing the prefix,

/// and you need to return the sum of all the pairs' value whose key

/// starts with the prefix.

///

/// Example 1:

/// Input: insert("apple", 3), Output: Null

/// Input: sum("ap"), Output: 3

/// Input: insert("app", 2), Output: Null

/// Input: sum("ap"), Output: 5

/// </summary>

class MapSum

{

private:

map<string, int> m\_MapValue;

public:

// Initialize your data structure here.

MapSum()

{

}

// insert a value

void insert(string key, int val)

{

m\_MapValue[key] = val;

}

// get the sum

int sum(string prefix)

{

int sum = 0;

map<string, int>::iterator itr = m\_MapValue.lower\_bound(prefix);

while (itr != m\_MapValue.end())

{

if (itr->first.substr(0, prefix.size()) == prefix)

{

sum += itr->second;

}

else

{

break;

}

++itr;

}

return sum;

}

};

## 1268. Search Suggestions System

Medium

You are given an array of strings products and a string searchWord.

Design a system that suggests at most three product names from products after each character of searchWord is typed. Suggested products should have common prefix with searchWord. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return *a list of lists of the suggested products after each character of*searchWord*is typed*.

**Example 1:**

**Input:** products = ["mobile","mouse","moneypot","monitor","mousepad"], searchWord = "mouse"

**Output:** [

["mobile","moneypot","monitor"],

["mobile","moneypot","monitor"],

["mouse","mousepad"],

["mouse","mousepad"],

["mouse","mousepad"]

]

**Explanation:** products sorted lexicographically = ["mobile","moneypot","monitor","mouse","mousepad"]

After typing m and mo all products match and we show user ["mobile","moneypot","monitor"]

After typing mou, mous and mouse the system suggests ["mouse","mousepad"]

**Example 2:**

**Input:** products = ["havana"], searchWord = "havana"

**Output:** [["havana"],["havana"],["havana"],["havana"],["havana"],["havana"]]

**Example 3:**

**Input:** products = ["bags","baggage","banner","box","cloths"], searchWord = "bags"

**Output:** [["baggage","bags","banner"],["baggage","bags","banner"],["baggage","bags"],["bags"]]

**Constraints:**

* 1 <= products.length <= 1000
* 1 <= products[i].length <= 3000
* 1 <= sum(products[i].length) <= 2 \* 104
* All the strings of products are **unique**.
* products[i] consists of lowercase English letters.
* 1 <= searchWord.length <= 1000
* searchWord consists of lowercase English letters.

### Analysis:

You can build the Trie, if you do so during the search you should go to all words level and collect the words. To make the code simple and short, you can use prefix hash table, which is write the words to all its prefix hash table.

/// <summary>

/// Leet code #1268. Search Suggestions System

///

/// Given an array of strings products and a string searchWord. We want to

/// design a system that suggests at most three product names from

/// products after each character of searchWord is typed. Suggested

/// products should have common prefix with the searchWord. If there are

/// more than three products with a common prefix return the three

/// lexicographically minimums products.

///

/// Return list of lists of the suggested products after each character of

/// searchWord is typed.

///

/// Example 1:

///

/// Input: products = ["mobile","mouse","moneypot","monitor","mousepad"],

/// searchWord = "mouse"

/// Output: [

/// ["mobile","moneypot","monitor"],

/// ["mobile","moneypot","monitor"],

/// ["mouse","mousepad"],

/// ["mouse","mousepad"],

/// ["mouse","mousepad"]

/// ]

/// Explanation: products sorted lexicographically = ["mobile","moneypot",

/// "monitor","mouse","mousepad"]

/// After typing m and mo all products match and we show user

/// ["mobile","moneypot","monitor"]

/// After typing mou, mous and mouse the system suggests

/// ["mouse","mousepad"]

///

/// Example 2:

///

/// Input: products = ["havana"], searchWord = "havana"

/// Output: [["havana"],["havana"],["havana"],["havana"],["havana"],

/// ["havana"]]

///

/// Example 3:

///

/// Input: products = ["bags","baggage","banner","box","cloths"],

/// searchWord = "bags"

/// Output: [["baggage","bags","banner"],["baggage","bags","banner"],

/// ["baggage","bags"],["bags"]]

///

/// Example 4:

///

/// Input: products = ["havana"], searchWord = "tatiana"

/// Output: [[],[],[],[],[],[],[]]

///

/// Constraints:

/// 1. 1 <= products.length <= 1000

/// 2. 1 <= SUM products[i].length <= 2 \* 10^4

/// 3. All characters of products[i] are lower-case English letters.

/// 4. 1 <= searchWord.length <= 1000

/// 5. All characters of searchWord are lower-case English letters.

/// </summary>

vector<vector<string>> LeetCodeString::suggestedProducts(vector<string>& products, string searchWord)

{

unordered\_map<string, multiset<string>> product\_map;

for (size\_t i = 0; i < products.size(); i++)

{

for (size\_t j = 0; j < products[i].size(); j++)

{

string prefix = products[i].substr(0, j + 1);

product\_map[prefix].insert(products[i]);

}

}

vector<vector<string>> result(searchWord.size());

for (size\_t i = 0; i < searchWord.size(); i++)

{

auto itr = product\_map[searchWord.substr(0, i + 1)].begin();

for (size\_t j = 0; j < 3; j++)

{

if (itr == product\_map[searchWord.substr(0, i + 1)].end()) break;

result[i].push\_back(\*itr);

itr++;

}

}

return result;

}

# Advanced Problems

## 1032. Stream of Characters

Hard

Design an algorithm that accepts a stream of characters and checks if a suffix of these characters is a string of a given array of strings words.

For example, if words = ["abc", "xyz"] and the stream added the four characters (one by one) 'a', 'x', 'y', and 'z', your algorithm should detect that the suffix "xyz" of the characters "axyz" matches "xyz" from words.

Implement the StreamChecker class:

* StreamChecker(String[] words) Initializes the object with the strings array words.
* boolean query(char letter) Accepts a new character from the stream and returns true if any non-empty suffix from the stream forms a word that is in words.

**Example 1:**

**Input**

["StreamChecker", "query", "query", "query", "query", "query", "query", "query", "query", "query", "query", "query", "query"]

[[["cd", "f", "kl"]], ["a"], ["b"], ["c"], ["d"], ["e"], ["f"], ["g"], ["h"], ["i"], ["j"], ["k"], ["l"]]

**Output**

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

**Explanation**

StreamChecker streamChecker = new StreamChecker(["cd", "f", "kl"]);

streamChecker.query("a"); // return False

streamChecker.query("b"); // return False

streamChecker.query("c"); // return False

streamChecker.query("d"); // return True, because 'cd' is in the wordlist

streamChecker.query("e"); // return False

streamChecker.query("f"); // return True, because 'f' is in the wordlist

streamChecker.query("g"); // return False

streamChecker.query("h"); // return False

streamChecker.query("i"); // return False

streamChecker.query("j"); // return False

streamChecker.query("k"); // return False

streamChecker.query("l"); // return True, because 'kl' is in the wordlist

**Constraints:**

* 1 <= words.length <= 2000
* 1 <= words[i].length <= 2000
* words[i] consists of lowercase English letters.
* letter is a lowercase English letter.
* At most 4 \* 104 calls will be made to query.

### Analysis:

First build the reverse words in Trie dictionary, and keep the last K characters in buffer, K is the maximum size of word, on every new character arrival, search the reverse substring in buffer to see if it makes any word.

/// <summary>

/// Leet code #1032. Stream of Characters

///

/// Implement the StreamChecker class as follows:

///

/// StreamChecker(words): Constructor, init the data structure with the given

/// words.

/// query(letter): returns true if and only if for some k >= 1, the last k

/// characters queried (in order from oldest to newest, including this letter

/// just queried) spell one of the words in the given list.

///

/// Example:

///

/// // init the dictionary.

/// StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]);

///

/// streamChecker.query('a'); // return false

/// streamChecker.query('b'); // return false

/// streamChecker.query('c'); // return false

/// streamChecker.query('d'); // return true, because 'cd' is in the wordlist

/// streamChecker.query('e'); // return false

/// streamChecker.query('f'); // return true, because 'f' is in the wordlist

/// streamChecker.query('g'); // return false

/// streamChecker.query('h'); // return false

/// streamChecker.query('i'); // return false

/// streamChecker.query('j'); // return false

/// streamChecker.query('k'); // return false

/// streamChecker.query('l'); // return true, because 'kl' is in the wordlist

///

/// Note:

///

/// 1. 1 <= words.length <= 2000

/// 2. 1 <= words[i].length <= 2000

/// 3. Words will only consist of lowercase English letters.

/// 4. Queries will only consist of lowercase English letters.

/// 5. The number of queries is at most 40000.

/// </summary>

class StreamChecker {

private:

struct Trie

{

bool is\_end;

vector<Trie \*> children;

Trie()

{

is\_end = false;

children = vector<Trie \*>(26, nullptr);

};

~Trie()

{

for (size\_t i = 0; i < children.size(); i++)

{

if (children[i] != nullptr) delete children[i];

}

}

void insert(string word)

{

if (word.empty())

{

is\_end = true;

}

else

{

int i = word[0] - 'a';

if (children[i] == nullptr)

{

children[i] = new Trie();

}

children[i]->insert(word.substr(1));

}

};

Trie \* next(char ch)

{

return children[ch - 'a'];

}

};

Trie m\_root;

size\_t m\_max\_len = 0;

deque<char> m\_buffer;

public:

StreamChecker(vector<string>& words)

{

for (size\_t i = 0; i < words.size(); i++)

{

string word = words[i];

std::reverse(word.begin(), word.end());

m\_root.insert(word);

m\_max\_len = max(m\_max\_len, word.size());

}

}

bool query(char letter)

{

m\_buffer.push\_front(letter);

if (m\_buffer.size() > m\_max\_len) m\_buffer.pop\_back();

Trie \* trie = nullptr;

for (size\_t i = 0; i < m\_buffer.size(); i++)

{

if (i == 0) trie = m\_root.next(m\_buffer[i]);

else

{

trie = trie->next(m\_buffer[i]);

}

if (trie == nullptr) return false;

if (trie->is\_end) return true;

}

return false;

}

};

## 642. Design Search Autocomplete System

Hard

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#').

You are given a string array sentences and an integer array times both of length n where sentences[i] is a previously typed sentence and times[i] is the corresponding number of times the sentence was typed. For each input character except '#', return the top 3 historical hot sentences that have the same prefix as the part of the sentence already typed.

Here are the specific rules:

* The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
* The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same hot degree, use ASCII-code order (smaller one appears first).
* If less than 3 hot sentences exist, return as many as you can.
* When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Implement the AutocompleteSystem class:

* AutocompleteSystem(String[] sentences, int[] times) Initializes the object with the sentences and times arrays.
* List<String> input(char c) This indicates that the user typed the character c.
  + Returns an empty array [] if c == '#' and stores the inputted sentence in the system.
  + Returns the top 3 historical hot sentences that have the same prefix as the part of the sentence already typed. If there are fewer than 3 matches, return them all.

**Example 1:**

**Input**

["AutocompleteSystem", "input", "input", "input", "input"]

[[["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2]], ["i"], [" "], ["a"], ["#"]]

**Output**

[null, ["i love you", "island", "i love leetcode"], ["i love you", "i love leetcode"], [], []]

**Explanation**

AutocompleteSystem obj = new AutocompleteSystem(["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2]);

obj.input("i"); // return ["i love you", "island", "i love leetcode"]. There are four sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot degree. Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be ignored.

obj.input(" "); // return ["i love you", "i love leetcode"]. There are only two sentences that have prefix "i ".

obj.input("a"); // return []. There are no sentences that have prefix "i a".

obj.input("#"); // return []. The user finished the input, the sentence "i a" should be saved as a historical sentence in system. And the following input will be counted as a new search.

**Constraints:**

* n == sentences.length
* n == times.length
* 1 <= n <= 100
* 1 <= sentences[i].length <= 100
* 1 <= times[i] <= 50
* c is a lowercase English letter, a hash '#', or space ' '.
* Each tested sentence will be a sequence of characters c that end with the character '#'.
* Each tested sentence will have a length in the range [1, 200].
* The words in each input sentence are separated by single spaces.
* At most 5000 calls will be made to input.

### Analysis:

You can implement it with Trie.

/// <summary>

/// Leet code #642. Design Search Autocomplete System

///

/// Design a search autocomplete system for a search engine. Users may

/// input a sentence (at least one word and end with a special character

/// '#'). For each character they type except '#', you need to return the

/// top 3 historical hot sentences that have prefix the same as the part

/// of sentence already typed. Here are the specific rules:

/// The hot degree for a sentence is defined as the number of times a user

/// typed the exactly same sentence before.

/// The returned top 3 hot sentences should be sorted by hot degree (The

/// first is the hottest one). If several sentences have the same degree

/// of hot, you need to use ASCII-code order (smaller one appears first).

/// If less than 3 hot sentences exist, then just return as many as you

/// can.

/// When the input is a special character, it means the sentence ends, and

/// in this case, you need to return an empty list.

///

/// Your job is to implement the following functions:

/// The constructor function:

/// AutocompleteSystem(String[] sentences, int[] times): This is the

/// constructor. The input is historical data. Sentences is a string array

/// consists of previously typed sentences. Times is the corresponding

/// times a sentence has been typed. Your system should record these

/// historical data.

/// Now, the user wants to input a new sentence. The following function

/// will provide the next character the user types:

/// List<String> input(char c): The input c is the next character typed by

/// the user. The character will only be lower-case letters ('a' to 'z'),

/// blank space (' ') or a special character ('#'). Also, the previously

/// typed sentence should be recorded in your system. The output will be

/// the top 3 historical hot sentences that have prefix the same as the

/// part of sentence already typed.

///

/// Example:

/// Operation: AutocompleteSystem(["i love you", "island","ironman",

/// "i love leetcode"], [5,3,2,2])

/// The system have already tracked down the following sentences and

/// their corresponding times:

/// "i love you" : 5 times

/// "island" : 3 times

/// "ironman" : 2 times

/// "i love leetcode" : 2 times

/// Now, the user begins another search:

///

/// Operation: input('i')

/// Output: ["i love you", "island","i love leetcode"]

/// Explanation:

/// There are four sentences that have prefix "i". Among them, "ironman"

/// and "i love leetcode" have same hot degree. Since ' ' has ASCII code 32

/// and 'r' has ASCII code 114, "i love leetcode" should be in front of

/// "ironman". Also we only need to output top 3 hot sentences, so

/// "ironman" will be ignored.

///

/// Operation: input(' ')

/// Output: ["i love you","i love leetcode"]

/// Explanation:

/// There are only two sentences that have prefix "i ".

///

/// Operation: input('a')

/// Output: []

/// Explanation:

/// There are no sentences that have prefix "i a".

///

/// Operation: input('#')

/// Output: []

/// Explanation:

/// The user finished the input, the sentence "i a" should be saved as a

/// historical sentence in system. And the following input will be counted

/// as a new search.

///

/// Note:

/// 1. The input sentence will always start with a letter and end with '#',

/// and only one blank space will exist between two words.

/// 2. The number of complete sentences that to be searched won't exceed

/// 100.

/// 3. The length of each sentence including those in the historical data

/// won't exceed 100.

/// 4. Please use double-quote instead of single-quote when you write test

/// cases even for a character input.

/// 5. Please remember to RESET your class variables declared in class

/// AutocompleteSystem, as static/class variables are persisted across

/// multiple test cases. Please see here for more details.

/// </summary>

class AutocompleteSystem

{

private:

struct Trie

{

set<pair<int, string>> m\_hot;

vector<Trie \*> children;

Trie()

{

children = vector<Trie \*>(27, nullptr);

};

~Trie()

{

for (size\_t i = 0; i < children.size(); i++)

{

if (children[i] != nullptr) delete children[i];

}

}

void add\_sentence(string sentence, int times)

{

pair<int, string> prev\_str = make\_pair(-times + 1, sentence);

if (m\_hot.count(prev\_str) > 0)

{

m\_hot.erase(prev\_str);

}

m\_hot.insert(make\_pair(-times, sentence));

if (m\_hot.size() > 3) m\_hot.erase(prev(m\_hot.end()));

}

vector<string> get\_hot\_sentences()

{

vector<string> result;

for (auto itr : m\_hot)

{

result.push\_back(itr.second);

}

return result;

}

Trie \* next(char ch)

{

if (ch == '#') return nullptr;

int i = ch - 'a';

if (ch == ' ') i = 26;

if (children[i] == nullptr)

{

children[i] = new Trie();

}

return children[i];

}

};

Trie m\_root;

unordered\_map<string, int> m\_sentence\_freq;

vector<Trie\*> m\_arr;

string m\_sentence;

public:

AutocompleteSystem(vector<string> sentences, vector<int> times)

{

for (size\_t i = 0; i < sentences.size(); i++)

{

m\_sentence\_freq[sentences[i]] = times[i];

for (size\_t j = 0; j < sentences[i].size(); j++)

{

if (j == 0) m\_arr.push\_back(m\_root.next(sentences[i][j]));

else

{

m\_arr.push\_back(m\_arr.back()->next(sentences[i][j]));

}

}

for (size\_t j = 0; j < m\_arr.size(); j++)

{

m\_arr[j]->add\_sentence(sentences[i], times[i]);

}

m\_arr.clear();

}

}

vector<string> input(char c)

{

vector<string> result;

if (c == '#')

{

m\_sentence\_freq[m\_sentence]++;

for (size\_t i = 0; i < m\_arr.size(); i++)

{

m\_arr[i]->add\_sentence(m\_sentence, m\_sentence\_freq[m\_sentence]);

}

m\_arr.clear();

m\_sentence.clear();

}

else

{

if (m\_arr.empty())

{

m\_arr.push\_back(m\_root.next(c));

}

else

{

m\_arr.push\_back(m\_arr.back()->next(c));

}

m\_sentence.push\_back(c);

result = m\_arr.back()->get\_hot\_sentences();

}

return result;

}

};