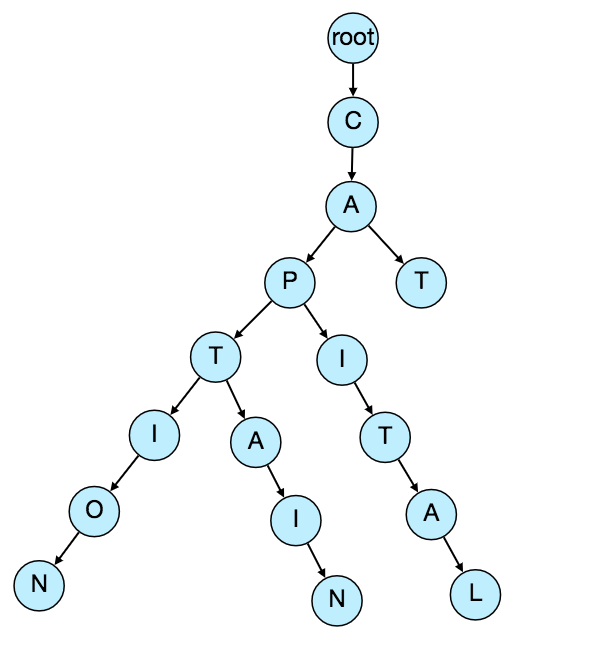
# Trie Search

The Trie data structure is used commonly in string search. The problem is that we want to search a word a huge word list. To avoid that we search words repeatedly we group the words by common prefix, so we can avoid duplicate search for the words with same prefix.

A trie is a tree looks like below.



Because a word can be a prefix of another word, in this case we need to indicate whether the current trie node is an end of word.

## 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:**

* You may assume that all inputs are consist of lowercase letters a-z.
* All inputs are guaranteed to be non-empty strings.

### **Analysis**

**We can treat Trie class as a node and construct itself recursively. Is\_end will indicate the end of the word.**

/// <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));

}

};

};

## Trie and Prefix Hash

Instead of store the Trie as a tree like structure which share the prefix, we can also use prefix hash to implement another similar algorithm. The idea of prefix hash is that you store every prefix of the word as a key in a hash table and search it by the key. This is an O(1) search algorithm. However the issue is that we may use too many space to store the prefix key.

Remember normally the value of such prefix hash is a vector because many words may fall into the same prefix.

## 745. Prefix and Suffix Search

Hard

Given many words, words[i] has weight i.

Design a class WordFilter that supports one function, WordFilter.f(String prefix, String suffix). It will return the word with given prefix and suffix with maximum weight. If no word exists, return -1.

**Examples:**

**Input:**

WordFilter(["apple"])

WordFilter.f("a", "e") // returns 0

WordFilter.f("b", "") // returns -1

**Note:**

1. words has length in range [1, 15000].
2. For each test case, up to words.length queries WordFilter.f may be made.
3. words[i] has length in range [1, 10].
4. prefix, suffix have lengths in range [0, 10].
5. words[i] and prefix, suffix queries consist of lowercase letters only.

### **Analysis**

**For this problem, although we can also use Trie to store the prefix and suffix, but there two problems here, first we have to store all the possible words with prefix and suffix, second search separately on prefix and suffix is not efficient. For the above reason we store the words by the key of prefix and suffix.**

/// <summary>

/// Leet code #745. Prefix and Suffix Search

///

/// Given many words, words[i] has weight i.

/// Design a class WordFilter that supports one function,

/// WordFilter.f(String prefix, String suffix). It will return the word with

/// given prefix and suffix with maximum weight. If no word exists, return -1.

///

/// Examples:

/// Input:

/// WordFilter(["apple"])

/// WordFilter.f("a", "e") // returns 0

/// WordFilter.f("b", "") // returns -1

/// Note:

/// words has length in range [1, 15000].

/// For each test case, up to words.length queries WordFilter.f may be made.

/// words[i] has length in range [1, 10].

/// prefix, suffix have lengths in range [0, 10].

/// words[i] and prefix, suffix queries consist of lowercase letters only.

/// </summary>

class WordFilter {

unordered\_map<string, int> m\_WordMap;

public:

WordFilter(vector<string> words)

{

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

{

for (size\_t i = 0; i <= 10 && i <= words[k].size(); i++)

{

string prefix = words[k].substr(0, i);

for (size\_t j = 0; j <= 10 && j <= words[k].size(); j++)

{

string suffix = words[k].substr(words[k].size() - j, j);

m\_WordMap[prefix + "#" + suffix] = k;

}

}

}

}

int f(string prefix, string suffix)

{

if (m\_WordMap.count(prefix + "#" + suffix) > 0)

{

return m\_WordMap[prefix + "#" + suffix];

}

else

{

return -1;

}

}

};

## 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 '#'). 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:

1. The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
2. 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).
3. If less than 3 hot sentences exist, then just return as many as you can.
4. 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. The length of each sentence including those in the historical data won't exceed 100.
3. Please use double-quote instead of single-quote when you write test cases even for a character input.
4. Please remember to **RESET** your class variables declared in class AutocompleteSystem, as static/class variables are **persisted across multiple test cases**. Please see [here](https://leetcode.com/faq/#different-output) for more details.

### **Analysis**

**For this problem, we also use Trie to store word list, and keep the top 3 frequent words at each node. During the iteration we keep track the sentence, the visited node and then when ‘#’ is input we add the word, if more frequent to each visited node on the way.**

/// <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;

}

};

## 1032. Stream of Characters

Hard

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:**

StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]); // init the dictionary.

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 <= words.length <= 2000
* 1 <= words[i].length <= 2000
* Words will only consist of lowercase English letters.
* Queries will only consist of lowercase English letters.
* The number of queries is at most 40000.

### **Analysis**

**We can reverse the input stream with maximum word length and use Trie to search the reverse words, when hit is\_end, we return true.**

/// <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;

}

};

## 212. Word Search II

Hard

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

**Example:**

**Input:**

**board** = [

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

**words** = ["oath","pea","eat","rain"]

**Output:**["eat","oath"]

**Note:**

1. All inputs are consist of lowercase letters a-z.
2. The values of words are distinct.

### **Analysis**

**To search the word, we use DFS. DFS is better than BFS because we need to keep track on word. We store the words with same prefix in Trie to avoid duplicate word search.**

**The answer also needs to be deduplicated, so we use hash set to store the result.**

struct TrieNode

{

bool is\_end;

vector<TrieNode \*> children;

TrieNode()

{

is\_end = false;

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

};

~TrieNode()

{

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

{

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

}

}

void insert(string word, int i)

{

if (i == word.size())

{

is\_end = true;

}

else

{

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

if (children[k] == nullptr)

{

children[k] = new TrieNode();

}

children[k]->insert(word, i + 1);

}

}

TrieNode \* next(char ch)

{

int k = ch - 'a';

return children[k];

}

};

/// <summary>

/// Leet code #212. Word Search II

/// </summary>

void LeetCodeDFS::wordSearchII(vector<vector<char>>& board, string& word,

TrieNode \* trie\_node, int x, int y, unordered\_set<string> &result)

{

if (x < 0 || x >= (int)board.size() || y < 0 || y >= (int)board[0].size())

{

return;

}

if (board[x][y] == '#') return;

char c = board[x][y];

trie\_node = (TrieNode \*)trie\_node->next(board[x][y]);

if (trie\_node == nullptr)

{

return;

}

board[x][y] = '#';

word.push\_back(c);

if (trie\_node->is\_end) result.insert(word);

wordSearchII(board, word, trie\_node, x - 1, y, result);

wordSearchII(board, word, trie\_node, x + 1, y, result);

wordSearchII(board, word, trie\_node, x, y - 1, result);

wordSearchII(board, word, trie\_node, x, y + 1, result);

board[x][y] = c;

word.pop\_back();

}

/// <summary>

/// Leet code #212. Word Search II

///

/// Given a 2D board and a list of words from the dictionary, find all words

/// in the board.

/// Each word must be constructed from letters of sequentially adjacent cell,

/// where "adjacent" cells are

/// those horizontally or vertically neighboring. The same letter cell may not

/// be used more than once in a word.

/// For example,

/// Given words = ["oath","pea","eat","rain"] and board =

/// [

/// ['o','a','a','n'],

/// ['e','t','a','e'],

/// ['i','h','k','r'],

/// ['i','f','l','v']

/// ]

/// Return ["eat","oath"].

/// Note:

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

/// </summary>

vector<string> LeetCodeDFS::wordSearchII(vector<vector<char>>& board, vector<string>& words)

{

unordered\_set<string> result;

TrieNode \* root = new TrieNode();

for (string word : words)

{

root->insert(word, 0);

}

for (size\_t x = 0; x < board.size(); x++)

{

for (size\_t y = 0; y < board[0].size(); y++)

{

string word;

wordSearchII(board, word, root, x, y, result);

}

}

delete root;

return vector<string>(result.begin(), result.end());

}

## 425. Word Squares

Hard

Given a set of words **(without duplicates)**, find all [word squares](https://en.wikipedia.org/wiki/Word_square) you can build from them.

A sequence of words forms a valid word square if the *k*th row and column read the exact same string, where 0 ≤ *k* < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

b a l l

a r e a

l e a d

l a d y

**Note:**

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

**Example 1:**

**Input:**

["area","lead","wall","lady","ball"]

**Output:**

[

[ "wall",

"area",

"lead",

"lady"

],

[ "ball",

"area",

"lead",

"lady"

]

]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

**Example 2:**

**Input:**

["abat","baba","atan","atal"]

**Output:**

[

[ "baba",

"abat",

"baba",

"atan"

],

[ "baba",

"abat",

"baba",

"atal"

]

]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

### **Analysis**

**We can resolve this problem by DFS.For every row, except for the first row, you will have some prefix decided by the previous rows. The best way is to leverage Trie. Here we need to get all the words with specified prefix then iterate them on the specific row.**

struct TrieNode

{

bool is\_end;

vector<TrieNode \*> children;

TrieNode()

{

is\_end = false;

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

};

~TrieNode()

{

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

{

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

}

}

void insert(string word, int i)

{

if (i == word.size())

{

is\_end = true;

}

else

{

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

if (children[k] == nullptr)

{

children[k] = new TrieNode();

}

children[k]->insert(word, i + 1);

}

}

TrieNode \* next(char ch)

{

int k = ch - 'a';

return children[k];

}

void get\_words(string &word, vector<string>& result)

{

if (is\_end) result.push\_back(word);

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

{

if (children[i] != nullptr)

{

word.push\_back((char)('a' + i));

children[i]->get\_words(word, result);

word.pop\_back();

}

}

}

};

/// <summary>

/// Leet code #425. Word Squares

/// </summary>

void LeetCodeDFS::wordSquares(TrieNode \*root, vector<string>& wordSquare, vector<vector<string>>& result)

{

if ((!wordSquare.empty()) && (wordSquare.size() == wordSquare[0].size()))

{

result.push\_back(wordSquare);

return;

}

string prefix;

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

{

prefix.push\_back(wordSquare[i][wordSquare.size()]);

}

vector<string> words;

TrieNode \* trie = root;

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

{

trie = trie->next(prefix[i]);

if (trie == nullptr) break;

}

if (trie != nullptr)

{

trie->get\_words(prefix, words);

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

{

wordSquare.push\_back(words[i]);

wordSquares(root, wordSquare, result);

wordSquare.pop\_back();

}

}

}

/// <summary>

/// Leet code #425. Word Squares

///

/// Given a set of words (without duplicates), find all word squares you

/// can build from them.

/// A sequence of words forms a valid word square if the kth row and column

/// read the exact same string, where 0 ≤ k < max(numRows, numColumns).

///

/// For example, the word sequence ["ball","area","lead","lady"] forms

/// a word square because each word reads the same both horizontally and

/// vertically.

/// b a l l

/// a r e a

/// l e a d

/// l a d y

///

/// Note:

/// 1.There are at least 1 and at most 1000 words.

/// 2.All words will have the exact same length.

/// 3.Word length is at least 1 and at most 5.

/// 4.Each word contains only lowercase English alphabet a-z.

///

/// Example 1:

/// Input:

/// ["area","lead","wall","lady","ball"]

///

/// Output:

/// [

/// [ "wall",

/// "area",

/// "lead",

/// "lady"

/// ],

/// [ "ball",

/// "area",

/// "lead",

/// "lady"

/// ]

/// ]

///

/// Explanation:

/// The output consists of two word squares. The order of output does not

/// matter

/// (just the order of words in each word square matters).

///

/// Example 2:

/// Input:

/// ["abat","baba","atan","atal"]

///

/// Output:

/// [

/// [ "baba",

/// "abat",

/// "baba",

/// "atan"

/// ],

/// [ "baba",

/// "abat",

/// "baba",

/// "atal"

/// ]

/// ]

///

/// Explanation:

/// The output consists of two word squares. The order of output does not matter

/// (just the order of words in each word square matters).

/// </summary>

vector<vector<string>> LeetCodeDFS::wordSquares(vector<string>& words)

{

vector<string> wordSquare;

vector<vector<string>> result;

TrieNode \*trie = new TrieNode();

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

{

trie->insert(words[i], 0);

}

wordSquares(trie, wordSquare, result);

return result;

}

## 336. Palindrome Pairs

Hard

Given a list of **unique** words, find all pairs of ***distinct*** indices (i, j) in the given list, so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome.

**Example 1:**

**Input:** ["abcd","dcba","lls","s","sssll"]

**Output:** [[0,1],[1,0],[3,2],[2,4]]

**Explanation:** The palindromes are ["dcbaabcd","abcddcba","slls","llssssll"]

**Example 2:**

**Input:** ["bat","tab","cat"]

**Output:** [[0,1],[1,0]]

**Explanation:** The palindromes are ["battab","tabbat"]

### **Analysis**

**At very beginning I tried Trie in this problem and it passed through. But a young man changed my opinion, using simple hash table is more straightforward. We first store all reversed word in the hash table, and then we scan each word from left to right, if the left is match a reverse word and right is palindrome or right match a reverse word and left is palindrome. We intentionally make right possible as empty string to take care of the empty string.**

/// <summary>

/// Leet code #336. Palindrome Pairs

///

/// Given a list of unique words, find all pairs of distinct

/// indices (i, j) in the given list,

/// so that the concatenation of the two words, i.e. words[i] + words[j]

/// is a palindrome.

/// Example 1:

/// Given words = ["bat", "tab", "cat"]

/// Return [[0, 1], [1, 0]]

///

/// Example 2:

/// Given words = ["abcd", "dcba", "lls", "s", "sssll"]

/// Return [[0, 1], [1, 0], [3, 2], [2, 4]]

/// The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]

/// </summary>

vector<vector<int>> LeetCodeString::palindromePairs(vector<string>& words)

{

vector<vector<int>> result;

unordered\_map<string, int> reverse\_map;

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

{

string str = words[i];

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

reverse\_map[str] = i;

}

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

{

string str = words[i];

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

{

string left = words[i].substr(0, k + 1);

string right = words[i].substr(k + 1);

if (left == string(left.rbegin(), left.rend()) &&

(reverse\_map.count(right) > 0))

{

int j = reverse\_map[right];

if (i != j)

{

result.push\_back({ j, i });

if (right.empty()) result.push\_back({ i, j });

}

}

if (right == string(right.rbegin(), right.rend()) &&

(reverse\_map.count(left) > 0))

{

int j = reverse\_map[left];

if (i != j) result.push\_back({ i, j });

}

}

}

return result;

}