<https://leetcode.com/problems/prefix-and-suffix-search/description/>

Design a special dictionary that searches the words in it by a prefix and a suffix.

Implement the WordFilter class:

WordFilter(string[] words) Initializes the object with the words in the dictionary.

f(string pref, string suff) Returns the index of the word in the dictionary, which has the prefix pref and the suffix suff. If there is more than one valid index, return **the largest** of them. If there is no such word in the dictionary, return -1.

**Example 1:**

**Input**

["WordFilter", "f"]

[[["apple"]], ["a", "e"]]

**Output**

[null, 0]

**Explanation**

WordFilter wordFilter = new WordFilter(["apple"]);

wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

**Constraints:**

1 <= words.length <= 10^4

1 <= words[i].length <= 7

1 <= pref.length, suff.length <= 7

words[i], pref and suff consist of lowercase English letters only.

At most 104 calls will be made to the function f.

**Attempt 1: 2024-03-24**

**Solution 1: Two Tries (30 min)**

class WordFilter {

    class TrieNode {

        TrieNode[] children;

        Set<String> words;

        public TrieNode() {

            this.children = new TrieNode[26];

            this.words = new HashSet<>();

        }

    }

    TrieNode prefix\_root;

    TrieNode suffix\_root;

    Map<String, Integer> indexes;

    public WordFilter(String[] words) {

        prefix\_root = new TrieNode();

        suffix\_root = new TrieNode();

        indexes = new HashMap<>();

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

            add(words[i]);

            indexes.put(words[i], i);

        }

    }

    private void add(String word) {

        TrieNode cur\_prefix = prefix\_root;

        TrieNode cur\_suffix = suffix\_root;

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

            // Insert for prefix search

            int index1 = word.charAt(i) - 'a';

            if(cur\_prefix.children[index1] == null) {

                cur\_prefix.children[index1] = new TrieNode();

            }

            cur\_prefix = cur\_prefix.children[index1];

            cur\_prefix.words.add(word);

            // Insert for suffix search

            // Note: Reverse scan, no need to separately

            // open a new for loop

            int index2 = word.charAt(word.length() - 1 - i) - 'a';

            if(cur\_suffix.children[index2] == null) {

                cur\_suffix.children[index2] = new TrieNode();

            }

            cur\_suffix = cur\_suffix.children[index2];

            cur\_suffix.words.add(word);

        }

    }

    public int f(String pref, String suff) {

        TrieNode iter\_prefix = prefix\_root;

        TrieNode iter\_suffix = suffix\_root;

        // Get all words with prefix

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

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

            if(iter\_prefix.children[index] == null) {

                return -1;

            }

            iter\_prefix = iter\_prefix.children[index];

        }

        Set<String> words\_contain\_prefix = iter\_prefix.words;

        // Get all words with suffix

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

            int index = suff.charAt(suff.length() - 1 - i) - 'a';

            if(iter\_suffix.children[index] == null) {

                return -1;

            }

            iter\_suffix = iter\_suffix.children[index];

        }

        Set<String> words\_contain\_suffix = iter\_suffix.words;

        int max\_index = -1;

        for(String word : words\_contain\_prefix) {

            if(words\_contain\_suffix.contains(word)) {

                max\_index = Math.max(max\_index, indexes.get(word));

            }

        }

        return max\_index;

    }

}

/\*\*

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

 \* WordFilter obj = new WordFilter(words);

 \* int param\_1 = obj.f(pref,suff);

 \*/

Time complexity:

WordFilter: O(L + W), where L is the length of the words array and W is the length of each wordinsert: O(W),

f: O(L), at worst case, we have to find the largest index of L words.

Space complexity: O(L \* W^2)

**Refer to**

**Visual Explanation | Double Trie JAVA**

<https://leetcode.com/problems/prefix-and-suffix-search/solutions/2164117/visual-explanation-double-trie-java/>

**Logic:**

The logic behind my approach is quite straightforward! Simply construct two tries: rootP and rootS each representing the root nodes of a prefix trie and a suffix trie.

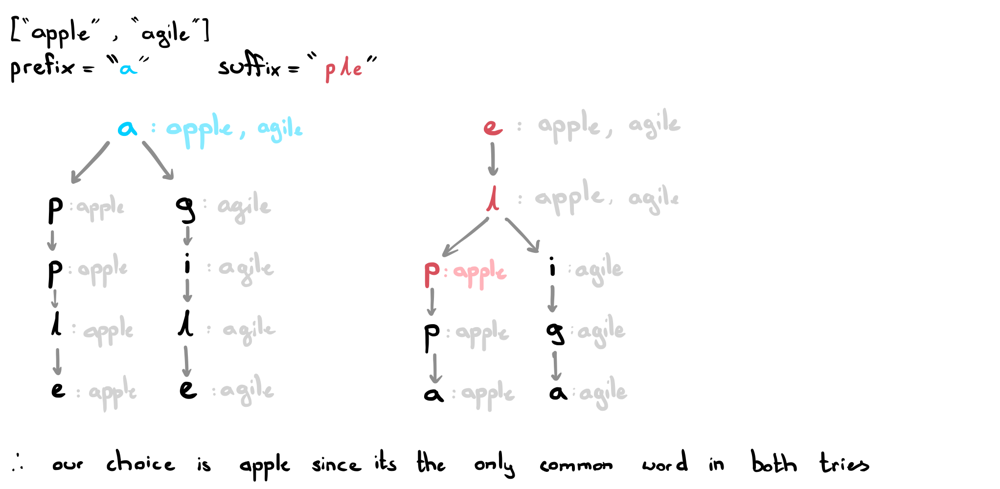
Building these two tries doesn't take too much effort. First, let's discuss how our TrieNode class will work.

TrieNode:

Map<Character, TrieNode> children: stores all letters following the current letter (TrieNodes)

Set<String> words: stores all words that share the current prefix / suffix

The words set is particularly important. Basically, in our prefix trie, at each TrieNode we will store all words associated with the current prefix in the words set. We will do the same for the suffix trie. This is what it would look like for the words ["apple", "agile"] with prefix "a" and suffix "ple":



As you can see, all we really need to do is collect the words associated with the ends of the prefix and suffix and find the common word with the largest index. To keep track of indexes, we'll simply use a map indexes that will store the largest index for each word.

Awesome, now we have everything we need to start coding!

**Code:**

class TrieNode {

public Map<Character, TrieNode> children;

public Set<String> words;

public TrieNode() {

children = new HashMap<>();

words = new HashSet<>();

}

}

class WordFilter {

private TrieNode rootP, rootS;

private Map<String, Integer> indexes;

public WordFilter(String[] words) {

rootP = new TrieNode();

rootS = new TrieNode();

indexes = new HashMap<>();

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

insert(words[i]);

indexes.put(words[i], i);

}

}

private void insert(String word) {

TrieNode ptrP = rootP, ptrS = rootS;

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

// insert as prefixes

char c = word.charAt(i);

if (!ptrP.children.containsKey(c))

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

ptrP = ptrP.children.get(c);

ptrP.words.add(word);

// insert as suffixes

c = word.charAt(word.length()-1-i);

if (!ptrS.children.containsKey(c))

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

ptrS = ptrS.children.get(c);

ptrS.words.add(word);

}

}

public int f(String prefix, String suffix) {

TrieNode ptrP = rootP, ptrS = rootS;

// get all words with prefix

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

char c = prefix.charAt(i);

if (!ptrP.children.containsKey(c)) return -1;

ptrP = ptrP.children.get(c);

}

Set<String> prefixes = ptrP.words;

// get all words with suffix

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

char c = suffix.charAt(suffix.length()-1-i);

if (!ptrS.children.containsKey(c)) return -1;

ptrS = ptrS.children.get(c);

}

Set<String> suffixes = ptrS.words;

int index = -1;

for (String word: prefixes)

if (suffixes.contains(word))

index = Math.max(index, indexes.get(word));

return index;

}

}

**Time complexity:**

WordFilter: O(L + W), where L is the length of the words array and W is the length of each word

insert: O(W),

f: O(L), at worst case, we have to find the largest index of L words.

**Space complexity:** O(L \* W^2)

**Solution 2: Trie + Merge Prefix and Suffix (30 min)**

class WordFilter {

    class TrieNode {

        TrieNode[] children;

        int indexInNums;

        public TrieNode() {

            // 'a' - 'z' and '{'. 'z' and '{' are neighbours in ASCII table

            this.children = new TrieNode[27];

            this.indexInNums = -1;

        }

    }

    TrieNode root;

    public WordFilter(String[] words) {

        root = new TrieNode();

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

            String word = "{" + words[i];

            add(word, i);

            // e.g words = {"apple"}

            // add "apple{apple", "pple{apple", "ple{apple", "le{apple",

            // "e{apple", "{apple" into the Trie Tree

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

                // Skip first '{' in word

                add(word.substring(j + 1) + word, i);

            }

        }

    }

    private void add(String word, int indexInNums) {

        TrieNode cur = root;

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

            // Insert for prefix search

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

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

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

            }

            cur = cur.children[index];

            cur.indexInNums = indexInNums;

        }

    }

    public int f(String pref, String suff) {

        TrieNode iter = root;

        // Get all words with "suffix + { + prefix"

        String target = suff + "{" + pref;

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

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

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

                return -1;

            }

            iter = iter.children[index];

        }

        // Note: no need map to store matched condition string

        // index in original input array, because the later

        // matched condition string's index will overwrite

        // previous smaller index

        return iter.indexInNums;

    }

}

/\*\*

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

 \* WordFilter obj = new WordFilter(words);

 \* int param\_1 = obj.f(pref,suff);

 \*/

Time complexity:

WordFilter: O(L + W), where L is the length of the words array and W is the length of each wordinsert: O(W),

f: O(L), at worst case, we have to find the largest index of L words.

Space complexity: O(L \* W^2)

**Refer to**

<https://leetcode.com/problems/prefix-and-suffix-search/solutions/144432/java-beat-95-just-small-modifications-in-implementing-trie/>

Take "apple" as an example, we will insert add "apple{apple", "pple{apple", "ple{apple", "le{apple", "e{apple", "{apple" into the Trie Tree.

If the query is: prefix = "app", suffix = "le", we can find it by querying our trie for

"le { app".

We use '{' because in ASCii Table, '{' is next to 'z', so we just need to create new TrieNode[27] instead of 26. Also, compared with tradition Trie, we add the attribute weight in class TrieNode.

class TrieNode {

TrieNode[] children;

int weight;

public TrieNode() {

weight = 0;

children = new TrieNode[27];

}

}

class WordFilter {

TrieNode root;

public WordFilter(String[] words) {

root = new TrieNode();

for (int weight = 0; weight < words.length; weight++) {

String word = '{' + words[weight];

insert(root, word, weight);

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

//skip first '{' in word

insert(root, word.substring(i+1) + word, weight);

}

}

}

private void insert(TrieNode root, String word, int weight) {

TrieNode cur = root;

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

int k = c - 'a';

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

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

}

cur = cur.children[k];

cur.weight = weight;

}

}

public int f(String prefix, String suffix) {

TrieNode cur = root;

for(char c : (suffix + '{' + prefix).toCharArray()) {

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

return -1;

}

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

}

return cur.weight;

}

}

**Refer to**

[L211.Design Add and Search Words Data Structure](note://2E59433C67644AD8BE4622BC47F9E4D8)