<https://leetcode.com/problems/stream-of-characters/>

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 <= 200

words[i] consists of lowercase English letters.

letter is a lowercase English letter.

At most 4 \* 10^4 calls will be made to query.

**Attempt 1: 2022-12-24**

**Solution 1: Construct Trie with Reversed Words (30 min)**

class StreamChecker {

    Trie trie;

    StringBuilder sb;

    public StreamChecker(String[] words) {

        sb = new StringBuilder();

        trie = new Trie();

        for(String word : words) {

            trie.insert(word);

        }

    }

    public boolean query(char letter) {

        String s = sb.insert(0, letter).toString();

        return trie.startWithGivenWord(s);

    }

}

/\*\*

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

\* StreamChecker obj = new StreamChecker(words);

\* boolean param\_1 = obj.query(letter);

\*/

class Trie {

    TrieNode root;

    public Trie() {

        this.root = new TrieNode();

    }

    public void insert(String s) {

        TrieNode cur = root;

        for(int i = s.length() - 1; i >= 0; i--) {

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

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

                TrieNode node = new TrieNode();

                cur.children[index] = node;

                cur = node;

            } else {

                cur = cur.children[index];

            }

        }

        cur.isEnd = true;

    }

    public boolean startWithGivenWord(String s) {

        TrieNode cur = root;

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

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

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

                cur = cur.children[index];

                if(cur.isEnd) {

                    return true;

                }

            } else {

                return false;

            }

        }

        return false;

    }

}

class TrieNode {

    public TrieNode[] children;

    public boolean isEnd;

    public TrieNode() {

        this.children = new TrieNode[26];

        this.isEnd = false;

    }

}

Assuming average word length for input string array words is L, length of words is M, length of query buffer size is N.

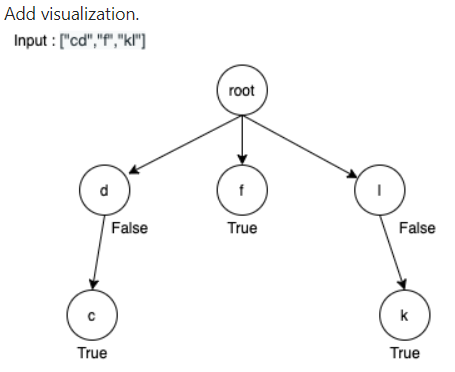
Time complexity: Inititalization -> O(L \* M); query -> O(N)

Space complexity: O(26 \* L \* M) -> O(LM)

**Refer to**

<https://leetcode.com/problems/stream-of-characters/solutions/278769/java-trie-solution/>

Store the words in the trie with reverse order, and check the query string from the end



class StreamChecker {

    class TrieNode {

        boolean isWord;

        TrieNode[] next = new TrieNode[26];

    }

    TrieNode root = new TrieNode();

    StringBuilder sb = new StringBuilder();

    public StreamChecker(String[] words) {

        createTrie(words);

    }

    public boolean query(char letter) {

        sb.append(letter);

        TrieNode node = root;

        for (int i = sb.length() - 1; i >= 0 && node != null; i--) {

            char c = sb.charAt(i);

            node = node.next[c - 'a'];

            if (node != null && node.isWord) {

                return true;

            }

        }

        return false;

    }

    private void createTrie(String[] words) {

        for (String s : words) {

            TrieNode node = root;

            int len = s.length();

            for (int i = len - 1; i >= 0; i--) {

                char c = s.charAt(i);

                if (node.next[c - 'a'] == null) {

                    node.next[c - 'a'] = new TrieNode();

                }

                node = node.next[c - 'a'];

            }

            node.isWord = true;

        }

    }

}

**Refer to**

<https://leetcode.com/problems/stream-of-characters/solutions/278250/python-trie-solution-with-explanation/>

**We say**

W = max(words.length),the maximum length of all words.

N = words.size, the number of words

Q, the number of calls of function query

**Solution 1: Check all words (TLE)**

If we save the whole input character stream and compare with words one by one,

The time complexity for each query will be O(NW),

depending on the size of words.

**Solution 2: Check Query Suffixes (Maybe AC, Maybe TLE)**

While the words.size can be really big,

the number of the suffixes of query stream is bounded.

For example, if the query stream is "abcd",

the suffix can be "abcd", "bcd", "cd", "d".

We can save all hashed words to a set.

For each query, we check query stream's all suffixes.

The maximum length of words is W, we need to check W suffixes.

The time complexity for each query will be O(W) if we take the set search as O(1).

The overall time is O(WQ).

**Solution 3: Trie (Accepted)**

Only a part of suffixes can be the prefix of a word,

waiting for characters coming to form a complete word.

Instead of checking all W suffixes in each query,

we can just save those possible waiting prefixes in a waiting list.

**Explanation**

**Initialization:**

Construct a trie

declare a global waiting list.

**Query:**

for each node in the waiting list, check if there is child node for the new character. If so, add it to the new waiting list.

return true if any node in the waitinglist is the end of a word.

**Time Complexity:**

waiting.size <= W, where W is the maximum length of words.

So that O(query) = O(waiting.size) = O(W)

We will make Q queries, the overall time complexity is O(QW)

Note that it has same complexity in the worst case as solution 2 (like "aaaaaaaa" for words and query),

In general cases, it saves time checking all suffixes, and also the set search in a big set.

**Space Complexity:**

waiting.size <= W, where W is the maximum length of words.

waiting list will take O(W)

Assume we have initially N words, at most N leaves in the trie.

The size of trie is O(NW).

**Python:**

Time: 6000+ms

class StreamChecker(object):

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

T = lambda: collections.defaultdict(T)

self.trie = T()

for w in words: reduce(dict.\_\_getitem\_\_, w, self.trie)['#'] = True

self.waiting = []

def query(self, letter):

self.waiting = [node[letter] for node in self.waiting + [self.trie] if letter in node]

return any("#" in node for node in self.waiting)

**Solution 4: Construct Trie with Reversed Words**

Time: 600 ~ 700ms

Time complexity: O(WQ)

**Python**

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

T = lambda: collections.defaultdict(T)

self.trie = T()

for w in words: reduce(dict.\_\_getitem\_\_, w[::-1], self.trie)['#'] = True

self.S = ""

self.W = max(map(len, words))

def query(self, letter):

self.S = (letter + self.S)[:self.W]

cur = self.trie

for c in self.S:

if c in cur:

cur = cur[c]

if cur['#'] == True:

return True

else:

break

return False

**Refer to**

[L208.Implement Trie (Prefix Tree)](note://68E95CFADD3A4906B6290D7096A7EFAD)

[L336.Palindrome Pairs (Ref.L5,L214)](note://883D658FFA6E4B7587F7F47404965C31)