Implement Magic Dictionary

Implement a magic directory with `buildDict`, and `search` methods.

For the method `buildDict`, you'll be given a list of non-repetitive words to build a dictionary.

For the method `search`, you'll be given a word, and judge whether if you modify **exactly** one character into **another** character in this word, the modified word is in the dictionary you just built.

Example 1:

```
Input: buildDict(["hello", "leetcode"]), Output: Null
Input: search("hello"), Output: False
Input: search("hhllo"), Output: True
Input: search("hell"), Output: False
Input: search("leetcoded"), Output: False
```

Note:

1. You may assume that all the inputs are consist of lowercase letters `a-z`.
2. For contest purpose, the test data is rather small by now. You could think about highly efficient algorithm after the contest.
3. Please remember to **RESET** your class variables declared in class MagicDictionary, as static/class variables are **persisted across multiple test cases**. Please see here for more details.

Solution 1

A word `'apple'` has neighbors `'*pple', 'a*ple', 'ap*le', 'app*e', 'appl*'`. When searching for a target word like `'apply'`, we know that a necessary condition is a neighbor of `'apply'` is a neighbor of some source word in our magic dictionary. If there is more than one source word that does this, then at least one of those source words will be different from the target word. Otherwise, we need to check that the source doesn't equal the target.

```
class MagicDictionary(object):
    def _candidates(self, word):
        for i in xrange(len(word)):
            yield word[:i] + '*' + word[i+1:]

    def buildDict(self, words):
        self.words = set(words)
        self.near = collections.Counter(cand for word in words
                                        for cand in self._candidates(word))

    def search(self, word):
        return any(self.near[cand] > 1 or
                   self.near[cand] == 1 and word not in self.words
                   for cand in self._candidates(word))
```

written by awice original link here

Solution 2

1. For each word in `dict`, for each char, remove the char and put the rest of the word as key, a pair of index of the removed char and the char as `part of` value list into a map. e.g.
   "hello" -> {"ello":[[0, 'h']], "hllo":[[1, 'e']], "helo":[[2, 'l'],[3, 'l']], "hell":[[4, 'o']]}
2. During search, generate the keys as in step 1. When we see there's pair of same index but different char in the value array, we know the answer is true. e.g.
   "healo" when remove `a`, key is "helo" and there is a pair [2, 'l'] which has same index but different char. Then the answer is true;

```
class MagicDictionary {

    Map<String, List<int[]>> map = new HashMap<>();
    /** Initialize your data structure here. */
    public MagicDictionary() {
    }

    /** Build a dictionary through a list of words */
    public void buildDict(String[] dict) {
        for (String s : dict) {
            for (int i = 0; i < s.length(); i++) {
                String key = s.substring(0, i) + s.substring(i + 1);
                int[] pair = new int[] {i, s.charAt(i)};

                List<int[]> val = map.getOrDefault(key, new ArrayList<int[]>());
                val.add(pair);

                map.put(key, val);
            }
        }
    }

    /** Returns if there is any word in the trie that equals to the given word after
modifying exactly one character */
    public boolean search(String word) {
        for (int i = 0; i < word.length(); i++) {
            String key = word.substring(0, i) + word.substring(i + 1);
            if (map.containsKey(key)) {
                for (int[] pair : map.get(key)) {
                    if (pair[0] == i && pair[1] != word.charAt(i)) return true;
                }
            }
        }
        return false;
    }
}
```

written by shawngao original link here

Solution 3

The implementation is a simple Trie, with the method relaxedSearch.

relaxedSearch searches for a word, with one deviation from a normal trie.

If there is a match with the current character, it proceeds as usual in that branch. But for all the non matched characters, it still continues searching, by incrementing the changedTimes variable, which maintains how many times a character was changed in the word search from the root.

Any search that involves changedTimes > 1, is immediately terminated by returning false as we are allowed to change only one character.

The solution is reached, when we find word in the trie and the changedTimes is exactly == 1.

```
class MagicDictionary {

        Trie trie;
        public MagicDictionary() {
            trie = new Trie(256);
        }

        public void buildDict(String[] dict) {
            Arrays.stream(dict).forEach(s -> trie.insert(s));
        }

        public boolean search(String word) {
            return trie.relaxedSearch(word);
        }

        class Trie {
            private int R;
            private TrieNode root;

            public Trie(int R) {
                this.R = R;
                root = new TrieNode();
            }

            public boolean relaxedSearch(String word) {
                return relaxedSearch(root, word, 0);
            }

            private boolean relaxedSearch(TrieNode root, String word, int changedTi
mes) {
                if (root == null || (!root.isWord && word.isEmpty()) || changedTimes
> 1) return false;
                if (root.isWord && word.isEmpty()) return changedTimes == 1;
                return Arrays.stream(root.next).anyMatch(nextNode -> relaxedSearch(
nextNode, word.substring(1),
                        root.next[word.charAt(0)] == nextNode ? changedTimes : chan
gedTimes+1));
            }

            // Inserts a word into the trie.
```

```
            // Inserts a word into the trie.
            public void insert(String word) {
                insert(root, word);
            }

            private void insert(TrieNode root, String word) {
                if (word.isEmpty()) { root.isWord = true; return; }
                if (root.next[word.charAt(0)] == null) root.next[word.charAt(0)] =
new TrieNode();
                insert(root.next[word.charAt(0)], word.substring(1));
            }

            private class TrieNode {
                private TrieNode[] next = new TrieNode[R];
                private boolean isWord;
            }

        }
    }
```

written by johnyrufus16 original link here

From Leetcoder.