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

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;

```java
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 2

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.

```python
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 3

Since the input set is small and the character set is finite, it is acceptable to enumerate through all possible combinations and see if the new word exists in the dictionary and the time complexity for search is O(k) where k is the length of the word.

Funnily enough, the comment for `search` already hints at using a "trie" but it's not needed to pass the contest.

By Yang Shun

```
class MagicDictionary(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.words = None

    def buildDict(self, dict):
        """
        Build a dictionary through a list of words
        :type dict: List[str]
        :rtype: void
        """
        self.words = set(dict)


    def search(self, word):
        """
        Returns if there is any word in the trie that equals to the given word after
modifying exactly one character
        :type word: str
        :rtype: bool
        """
        chars = set(word)
        for index, char in enumerate(word):
            for i in range(26):
                sub = chr(ord('a') + i)
                if sub == char:
                    continue
                new_word = word[:index] + sub + word[index + 1:]
                if new_word in self.words:
                    return True
        return False
```

written by yangshun original link here