288. Unique Word Abbreviation

March 5, 2016 | 15.7K views

Average Rating: 4 (21 votes)

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

```
a) it                      --> it    (no abbreviation)

     1
     ↓
b) d|o|g                   --> d1g

              1    1  1
     1---5----0----5--8
     ↓   ↓    ↓    ↓  ↓
c) i|nternationalizatio|n  --> i18n

              1
     1---5----0
     ↓   ↓    ↓
d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no *other* word from the dictionary has the same abbreviation.

Example:

```
Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

Summary

This problem has a low acceptance rate for a reason. The logic in `isUnique` can be a little tricky to get right due to the number of cases you need to consider. We highly recommend that you practice this similar but easier problem first - Two Sum III - Data structure design.

Solution

Approach #1 (Brute Force)

Let us begin by storing the dictionary first in the constructor. To determine if a word's abbreviation is unique with respect to a word in the dictionary, we check if all the following conditions are met:

1. They are not the same word.
2. They both have equal lengths.
3. They both share the same first and last letter.

Note that Condition #1 is implicit because from the problem statement:

> A word's abbreviation is unique if no ***other*** word from the dictionary has the same abbreviation.

```
public class ValidWordAbbr {
    private final String[] dict;

    public ValidWordAbbr(String[] dictionary) {
        dict = dictionary;
    }

    public boolean isUnique(String word) {
        int n = word.length();
        for (String s : dict) {
            if (word.equals(s)) {
                continue;
            }
            int m = s.length();
            if (m == n
                && s.charAt(0) == word.charAt(0)
                && s.charAt(m - 1) == word.charAt(n - 1)) {
                return false;
            }
        }
        return true;
    }
}
```

Complexity analysis

- Time complexity : $O(n)$ for each `isUnique` call. Assume that n is the number of words in the dictionary, each `isUnique` call takes $O(n)$ time.

Approach #2 (Hash Table) [Accepted]

Note that `isUnique` is called repeatedly for the same set of words in the dictionary each time. We should pre-process the dictionary to speed it up.

Ideally, a hash table supports constant time look up. What should the key-value pair be?

Well, the idea is to *group* the words that fall under the same abbreviation together. For the value, we use a Set instead of a List to guarantee uniqueness.

The logic in `isUnique(word)` is tricky. You need to consider the following cases:

1. Does the word's abbreviation exists in the dictionary? If not, then it must be unique.
2. If above is yes, then it can only be unique if the grouping of the abbreviation contains no other words except *word*.

```
public class ValidWordAbbr {
    private final Map<String, Set<String>> abbrDict = new HashMap<>();

    public ValidWordAbbr(String[] dictionary) {
        for (String s : dictionary) {
            String abbr = toAbbr(s);
            Set<String> words = abbrDict.containsKey(abbr)
                ? abbrDict.get(abbr) : new HashSet<>();
            words.add(s);
            abbrDict.put(abbr, words);
        }
    }

    public boolean isUnique(String word) {
        String abbr = toAbbr(word);
        Set<String> words = abbrDict.get(abbr);
        return words == null || (words.size() == 1 && words.contains(word));
    }

    private String toAbbr(String s) {
        int n = s.length();
        if (n <= 2) {
            return s;
        }
        return s.charAt(0) + Integer.toString(n - 2) + s.charAt(n - 1);
    }
}
```

Approach #3 (Hash Table) [Accepted]

Let us consider another approach using a counter as the table's value. For example, assume the dictionary = `["door", "deer"]`, we have the mapping of `{"d2r" -> 2}`. However, this mapping alone is not enough, because we need to consider whether the word exists in the dictionary. This can be easily overcome by inserting the entire dictionary into a set.

When an abbreviation's counter exceeds one, we know this abbreviation must not be unique because at least two different words share the same abbreviation. Therefore, we can further simplify the counter to just a boolean.

```
public class ValidWordAbbr {
    private final Map<String, Boolean> abbrDict = new HashMap<>();
    private final Set<String> dict;

    public ValidWordAbbr(String[] dictionary) {
        dict = new HashSet<>(Arrays.asList(dictionary));
        for (String s : dict) {
            String abbr = toAbbr(s);
            abbrDict.put(abbr, !abbrDict.containsKey(abbr));
        }
    }

    public boolean isUnique(String word) {
        String abbr = toAbbr(word);
        Boolean hasAbbr = abbrDict.get(abbr);
        return hasAbbr == null || (hasAbbr && dict.contains(word));
    }

    private String toAbbr(String s) {
        int n = s.length();
        if (n <= 2) {
            return s;
        }
        return s.charAt(0) + Integer.toString(n - 2) + s.charAt(n - 1);
    }
}
```

Complexity analysis

- Time complexity : $O(n)$ pre-processing, $O(1)$ for each `isUnique` call. Both Approach #2 and Approach #3 above take $O(n)$ pre-processing time in the constructor. This is totally worth it if `isUnique` is called repeatedly.
- Space complexity : $O(n)$. We traded the extra $O(n)$ space storing the table to reduce the time complexity in `isUnique`.

Rate this article: