入选理由

1. 通过昨天的学习，相信大家已经会手撸前缀树了，那么用前缀树可以解决什么样的题目呢？今天我们就来看看~

标签

- 前缀树

难度

- 中等

题目地址(677. 键值映射)

https://leetcode-cn.com/problems/map-sum-pairs

题目描述

实现一个 MapSum 类里的两个方法，insert 和 sum。

对于方法 insert，你将得到一对（字符串，整数）的键值对。字符串表示键，整数表示值。如果键已经存在，那么原来的键值对将被替代成新的键值对。

对于方法 sum，你将得到一个表示前缀的字符串，你需要返回所有以该前缀开头的键的值的总和。

示例 1:

输入: insert("apple", 3), 输出: Null 输入: sum("ap"), 输出: 3 输入: insert("app", 2), 输出: Null 输入: sum("ap"), 输出: 5

前置知识

- 哈希表
- Trie
- DFS

哈希表

思路

题目说的简单也明白，就是让我们实现两个方法。

方法一：题目既然都说叫“键值映射”了，我们自然而然就可以想到 hashmap，接下来分析是否可行：

- 对于 insert 方法，输入是键值对且键重复覆盖值，hashmap 完美契合。
- 对于 sum 方法，要求是找到所有以给定字符串为前缀的键的值的求和，那我们遍历一遍键不就知道了。

代码

代码支持：Python，Java

```python
class MapSum:

    def __init__(self):
        self.m = {}
    def insert(self, key, val):
        self.m[key] = val

    def sum(self, prefix):
        count = 0
        for key in self.m:
            if key.startswith(prefix):
                count += self.m[key]
        return count
```

```java
class MapSum {

    Map<String, Integer> map;

    public MapSum() {

        map = new HashMap<>();
    }

    public void insert(String key, int val) {

        map.put(key, val);
    }

    public int sum(String prefix) {

        int count = 0;

        for (String key: map.keySet())
            if(key.startsWith(prefix))
```

```
                count += map.get(key);

        return count;
    }
}
```

复杂度分析

- 空间复杂度：$O(N)$，其中 N 是不重复的 key 的个数

- 时间复杂度：插入是$O(1)$，求和操作是$O(N * S)$，其中 N 是目前为止 key 的个数，S 是前缀长度。

前缀树

思路

我们继续考虑，这个 sum 方法是找所有以 xxx 为前缀的字符串，那么就想到了 Trie(关键词：字符串前缀)，那么我们分析一下是否可行：

- 对于 insert 方法，键值映射这事儿 Trie 也可以胜任，因为我们的 Node 节点我们想怎么设定就怎么设定。

- 对于 sum 方法，这事就该交给 Trie 来办，找到指定前缀的结尾所对应 Trie 的 Node，直接把所有分叉全都遍历一遍不就完事了，遇到是键的从对应节点里取我们的值就可以了。

代码：

代码支持：Python，Java

```
class MapSum {

    TrieNode root;

    public MapSum() {

        root = new TrieNode();
    }

    public void insert(String key, int val) {

        TrieNode temp = root;
        for (int i = 0; i < key.length(); i++) {

            if (temp.children[key.charAt(i) - 'a'] == null)
                temp.children[key.charAt(i) - 'a'] = new TrieNode();
```

```
            temp = temp.children[key.charAt(i) - 'a'];

        }
        temp.count = val;
    }

    public int sum(String prefix) {

        TrieNode temp = root;

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

            if (temp.children[prefix.charAt(i) - 'a'] == null)
                return 0;

            temp = temp.children[prefix.charAt(i) - 'a'];
        }

        return dfs(temp);
    }

    public int dfs(TrieNode node) {

        int sum = 0;

        for (TrieNode t : node.children)
            if (t != null)
                sum += dfs(t);

        return sum + node.count;
    }

    private class TrieNode {

        int count; //表示以该处节点构成的串为前缀的个数
        TrieNode[] children;

        TrieNode() {

            count = 0;
            children = new TrieNode[26];
        }
    }
}
```

复杂度分析

- 空间复杂度：参考讲义 Trie 复杂度分析

- 时间复杂度：插入操作是线性复杂度，sum 操作最坏情况是$O(m^n)$（可以理解成从根结点遍历了所有节点，该题可以将 sum 操作的时间优化成线性，避免 dfs 这种搜索操作，大家可以试试）

这里还是给出优化后的代码供大家参考：

```
class MapSum {

    TrieNode root;

    public MapSum() {

        root = new TrieNode();
    }

    public void insert(String key, int val) {

        TrieNode temp = root;

        int oldVal = searchValue(key);

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

            if (temp.children[key.charAt(i) - 'a'] == null)
                temp.children[key.charAt(i) - 'a'] = new TrieNode();

            temp = temp.children[key.charAt(i) - 'a'];

            // update val
            temp.count = temp.count - oldVal + val;
        }

        temp.val = val;
        temp.isWord = true;
    }

    public int searchValue(String key) {

        TrieNode temp = root;
        for (int i = 0; i < key.length(); i++) {

            if (temp.children[key.charAt(i) - 'a'] == null)
                return 0;

            temp = temp.children[key.charAt(i) - 'a'];
        }

        return temp.isWord ? temp.val : 0;
    }

    public int sum(String prefix) {

        TrieNode temp = root;

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

            if (temp.children[prefix.charAt(i) - 'a'] == null)
```

```
                return 0;

            temp = temp.children[prefix.charAt(i) - 'a'];
        }

        return temp.count;
    }

    private class TrieNode {

        int count; //表示以该处节点构成的串为前缀的个数
        int val;
        TrieNode[] children;
        boolean isWord;

        TrieNode() {

            count = 0;
            children = new TrieNode[26];
            isWord = false;
            val = 0;
        }
    }
}
```

上一页 下一页