642. Design Search Autocomplete System

4 2 Add to List

Description Hints Submissions Discuss Solution

Solution (under development)

Approach #1 Brute Force [Time Limit Exceeded]

In this solution, we make use of a HashMap map which stores entries in the form $(sentence_i, times_i)$. Here, $times_i$ refers to the number of times the $sentence_i$ has been typed earlier.

`AutocompleteSystem` : We pick up each sentence from $sentences$ and their corresponding times from the $times$, and make their entries in the map appropriately.

`input(c)` : We make use of a current sentence tracker variable, cur_sen, which is used to store the sentence entered till now as the input. For c as the current input, firstly, we append this c to cur_sen and then iterate over all the keys of map to check if a key exists whose initial characters match with cur_sen. We add all such keys to a $list$. Then, we sort this $list$ as per our requirements, and obtain the first three values from this $list$.

Java

```
public class AutocompleteSystem {
    HashMap < String, Integer > map = new HashMap < > ();
    class Node {
        Node(String st, int t) {
            sentence = st;
            times = t;
        }
        String sentence;
        int times;
    }
    public AutocompleteSystem(String[] sentences, int[] times) {
        for (int i = 0; i < sentences.length; i++)
            map.put(sentences[i], times[i]);
    }
    String cur_sent = "";
    public List < String > input(char c) {
        List < String > res = new ArrayList < > ();
        if (c == '#') {
            map.put(cur_sent, map.getOrDefault(cur_sent, 0) + 1);
            cur_sent = "";
        } else {
            List < Node > list = new ArrayList < > ();
            cur_sent += c;
            for (String key: map.keySet())
                if (key.indexOf(cur_sent) == 0) {
                    list.add(new Node(key, map.get(key)));
                }
            Collections.sort(list, (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);
            for (int i = 0; i < Math.min(3, list.size()); i++)
                res.add(list.get(i).sentence);
        }
        return res;
    }
}

/**
 * Your AutocompleteSystem object will be instantiated and called as such:
 * AutocompleteSystem obj = new AutocompleteSystem(sentences, times);
 * List<String> param_1 = obj.input(c);
 */
```

Approach #2 Using One level Indexing[Accepted]

This method is almost the same as that of the last approach except that instead of making use of simply a HashMap to store the sentences along with their number of occurences, we make use of a Two level HashMap. Thus, we make use of an array arr of HashMapsEach element of this array, arr, is used to refer to one of the alphabets possible. Each element is a HashMap itself, which stores the sentences and their number of occurences similar to the last approach. e.g. $arr[0]$ is used to refer to a HashMap which stores the sentences starting with an 'a'. The process of adding the data in `AutocompleteSystem` and retrieving the data remains the same as in the last approach, except the one level indexing using arr which needs to be done prior to accessing the required HashMap.

Java

```
public class AutocompleteSystem {
    HashMap < String, Integer > [] arr;
    class Node {
        Node(String st, int t) {
            sentence = st;
            times = t;
        }
        String sentence;
        int times;
    }
    public AutocompleteSystem(String[] sentences, int[] times) {
        arr = new HashMap[26];
        for (int i = 0; i < 26; i++)
            arr[i] = new HashMap < String, Integer > ();
        for (int i = 0; i < sentences.length; i++)
            arr[sentences[i].charAt(0) - 'a'].put(sentences[i], times[i]);
    }
    String cur_sent = "";
    public List < String > input(char c) {
        List < String > res = new ArrayList < > ();
        if (c == '#') {
            arr[cur_sent.charAt(0) - 'a'].put(cur_sent, arr[cur_sent.charAt(0) - 'a'].getOrDefault(cur_sent, 0) + 1);
            cur_sent = "";
        } else {
            List < Node > list = new ArrayList < > ();
            cur_sent += c;
            for (String key: arr[cur_sent.charAt(0) - 'a'].keySet()) {
                if (key.indexOf(cur_sent) == 0) {
                    list.add(new Node(key, arr[cur_sent.charAt(0) - 'a'].get(key)));
                }
            }
            Collections.sort(list, (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);
            for (int i = 0; i < Math.min(3, list.size()); i++)
                res.add(list.get(i).sentence);
        }
        return res;
    }
}
```

Approach #3 Using Trie[Accepted]

A Trie is a special data structure used to store strings that can be visualized like a tree. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.

Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

Java

```
public class AutocompleteSystem {
    class Node {
        Node(String st, int t) {
            sentence = st;
            times = t;
        }
        String sentence;
        int times;
    }
    class Trie {
        int times;
        Trie[] branches = new Trie[27];
    }
    public int int_(char c) {
        return c == ' ' ? 26 : c - 'a';
    }
    public void insert(Trie t, String s, int times) {
        for (int i = 0; i < s.length(); i++) {
            if (t.branches[int_(s.charAt(i))] == null)
                t.branches[int_(s.charAt(i))] = new Trie();
            t = t.branches[int_(s.charAt(i))];
        }
        t.times += times;
    }
    public List < Node > lookup(Trie t, String s) {
        List < Node > list = new ArrayList < > ();
        for (int i = 0; i < s.length(); i++) {
            if (t.branches[int_(s.charAt(i))] == null)
                return new ArrayList < Node > ();
            t = t.branches[int_(s.charAt(i))];
        }
        traverse(s, t, list);
        return list;
    }
    public void traverse(String s, Trie t, List < Node > list) {
        if (t.times > 0)
            list.add(new Node(s, t.times));
        for (char i = 'a'; i <= 'z'; i++) {
            if (t.branches[i - 'a'] != null)
                traverse(s + i, t.branches[i - 'a'], list);
        }
        if (t.branches[26] != null)
            traverse(s + ' ', t.branches[26], list);
    }
    Trie root;
    public AutocompleteSystem(String[] sentences, int[] times) {
        root = new Trie();
        for (int i = 0; i < sentences.length; i++) {
            insert(root, sentences[i], times[i]);
        }
    }
    String cur_sent = "";
    public List < String > input(char c) {
        List < String > res = new ArrayList < > ();
        if (c == '#') {
            insert(root, cur_sent, 1);
            cur_sent = "";
        } else {
            cur_sent += c;
            List < Node > list = lookup(root, cur_sent);
            Collections.sort(list, (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);
            for (int i = 0; i < Math.min(3, list.size()); i++)
                res.add(list.get(i).sentence);
        }
        return res;
    }
}
```

Analysis written by: @vinod23