Design Search Autocomplete System

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character `'#'`). For **each character** they type **except '#'**, you need to return the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed. Here are the specific rules:

1. The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
2. The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same degree of hot, you need to use ASCII-code order (smaller one appears first).
3. If less than 3 hot sentences exist, then just return as many as you can.
4. When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Your job is to implement the following functions:

The constructor function:

`AutocompleteSystem(String[] sentences, int[] times):` This is the constructor. The input is **historical data**. `Sentences` is a string array consists of previously typed sentences. `Times` is the corresponding times a sentence has been typed. Your system should record these historical data.

Now, the user wants to input a new sentence. The following function will provide the next character the user types:

`List<String> input(char c):` The input `c` is the next character typed by the user. The character will only be lower-case letters (`'a'` to `'z'`), blank space (`' '`) or a special character (`'#'`). Also, the previously typed sentence should be recorded in your system. The output will be the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed.

Example:
Operation: AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"], [5,3,2,2])
The system have already tracked down the following sentences and their corresponding times:
`"i love you"` : `5` times
`"island"` : `3` times
`"ironman"` : `2` times
`"i love leetcode"` : `2` times
Now, the user begins another search:

Operation: input('i')
Output: ["i love you", "island","i love leetcode"]

Explanation:
There are four sentences that have prefix `"i"`. Among them, "ironman" and "i love leetcode" have same hot degree. Since `' '` has ASCII code 32 and `'r'` has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be ignored.

Operation: input(' ')
Output: ["i love you","i love leetcode"]
Explanation:
There are only two sentences that have prefix `"i "`.

Operation: input('a')
Output: []
Explanation:
There are no sentences that have prefix `"i a"`.

Operation: input('#')
Output: []
Explanation:
The user finished the input, the sentence `"i a"` should be saved as a historical sentence in system. And the following input will be counted as a new search.

Note:

1. The input sentence will always start with a letter and end with '#', and only one blank space will exist between two words.
2. The number of **complete sentences** that to be searched won't exceed 100. The length of each sentence including those in the historical data won't exceed 100.
3. Please use double-quote instead of single-quote when you write test cases even for a character input.
4. Please remember to **RESET** your class variables declared in class AutocompleteSystem, as static/class variables are **persisted across multiple test cases**. Please see here for more details.

Solution 1

Only thing more than a normal `Trie` is added a map of `sentence` to `count` in each of the `Trie` node to facilitate process of getting top 3 results.

```java
public class AutocompleteSystem {
    class TrieNode {
        Map<Character, TrieNode> children;
        Map<String, Integer> counts;
        boolean isWord;
        public TrieNode() {
            children = new HashMap<Character, TrieNode>();
            counts = new HashMap<String, Integer>();
            isWord = false;
        }
    }

    class Pair {
        String s;
        int c;
        public Pair(String s, int c) {
            this.s = s; this.c = c;
        }
    }

    TrieNode root;
    String prefix;


    public AutocompleteSystem(String[] sentences, int[] times) {
        root = new TrieNode();
        prefix = "";

        for (int i = 0; i < sentences.length; i++) {
   add(sentences[i], times[i]);
  }
    }

    private void add(String s, int count) {
        TrieNode curr = root;
        for (char c : s.toCharArray()) {
            TrieNode next = curr.children.get(c);
            if (next == null) {
                next = new TrieNode();
                curr.children.put(c, next);
            }
            curr = next;
            curr.counts.put(s, curr.counts.getOrDefault(s, 0) + count);
        }
        curr.isWord = true;
    }

    public List<String> input(char c) {
        if (c == '#') {
            add(prefix, 1);
            prefix = "";
            return new ArrayList<String>();
            return new ArrayList<String>();
        }

        prefix = prefix + c;
        TrieNode curr = root;
  for (char cc : prefix.toCharArray()) {
   TrieNode next = curr.children.get(cc);
   if (next == null) {
    return new ArrayList<String>();
   }
   curr = next;
  }

        PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> (a.c == b.c ? a.s.compareTo(b.s) : b.c - a.c));
        for (String s : curr.counts.keySet()) {
            pq.add(new Pair(s, curr.counts.get(s)));
        }

        List<String> res = new ArrayList<String>();
        for (int i = 0; i < 3 && !pq.isEmpty(); i++) {
            res.add(pq.poll().s);
        }
        return res;
    }

}
```

written by shawngao original link here

Solution 2

```cpp
class AutocompleteSystem {
    unordered_map<string, int> dict;
    string data;

public:
    AutocompleteSystem(vector<string> sentences, vector<int> times) {
        for (int i = 0; i < times.size(); i++)
            dict[sentences[i]] += times[i];
        data.clear();
    }

    vector<string> input(char c) {
        if (c == '#') {
            dict[data]++;
            data.clear();
            return {};
        }

        data.push_back(c);
        auto cmp = [](const pair<string, int> &a, const pair<string, int> &b) {
            return a.second > b.second || a.second == b.second && a.first < b.first;
        };

        priority_queue<pair<string, int>, vector<pair<string, int>>, decltype(cmp)>
pq(cmp);

        for (auto &p : dict) {
            bool match = true;
            for (int i = 0; i < data.size(); i++) {
                if (data[i] != p.first[i]) {
                    match = false;
                    break;
                }
            }
            if (match) {
                pq.push(p);
                if (pq.size() > 3)
                    pq.pop();
            }
        }

        vector<string> res(pq.size());
        for (int i = pq.size() - 1; i >= 0; i--) {
            res[i] = pq.top().first;
            pq.pop();
        }
        return res;
    }
};
```

written by mzchen original link here

Solution 3

I used Trie Data Structure for this problem and I don't know whether this is the optimal way to do this problem.

Also don't forget to cache the searched string into the database.

```
public class AutocompleteSystem {
    public Trie trie;
    public TrieNode root;
    public String prefix;
    public AutocompleteSystem(String[] sentences, int[] times) {
        trie = new Trie();
        root = trie.root;
        prefix = "";
        for (int i = 0; i < times.length; i++) {
            trie.insert(sentences[i], times[i]);
        }
    }

    public List<String> input(char c) {
        List<String> list = new ArrayList<>();
        PriorityQueue<Pair> pq = new PriorityQueue<>((a, b)-> (a.freq == b.freq) ? (
a.token.compareTo(b.token)) : b.freq - a.freq);
        if (c == '#')   {
            root = trie.root;
            trie.insert(prefix, 1);
            prefix = "";
            return new ArrayList<String>();
        }
        prefix = prefix + c;
        root = trie.searchHelper(prefix);
        trie.addToPQ(root, pq, prefix);
        for (int i = 1; i <= 3; i++){
            if (pq.size() > 0)
                list.add(pq.poll().token);
        }
        return list;
    }

    class Pair {
        String token;
        int freq;
        Pair(String token, int freq){
            this.token = token;
            this.freq = freq;
        }
    }


    class TrieNode{
        int freq;
        TrieNode[] children;
        public TrieNode() {
            freq = 0;
            children = new TrieNode[27];
        }
```

```java
    }

    class Trie {
        public TrieNode root;
        /** Initialize your data structure here. */
        public Trie() {
            root = new TrieNode();
        }

        /** Inserts a word into the trie. */
        public void insert(String word, int f) {
            TrieNode ws = root;
            for (char ch: word.toCharArray()) {
                int id = ch - 'a';
                if (ch == ' ')  id = 26;
                if (ws.children[id] == null)
                    ws.children[id] = new TrieNode();
                ws = ws.children[id];
            }
            ws.freq += f;
        }

        /** Returns if the word is in the trie. */
        public boolean search(String word) {
            TrieNode ws = searchHelper(word);
            return ws != null && ws.freq > 0;
        }

        /** Returns if there is any word in the trie that starts with the given pref
ix. */
        public boolean startsWith(String prefix) {
            return searchHelper(prefix) != null;
        }

        public TrieNode searchHelper(String str) {
            TrieNode ws = root;
            for (char ch: str.toCharArray()){
                int id = ch - 'a';
                if (ch == ' ')  id = 26;
                if (ws == null) return null;
                ws = ws.children[id];
            }
            return ws;
        }

        public void addToPQ(TrieNode root, PriorityQueue<Pair> pq, String prefix) {
            if (root == null)   return;
            if (root.freq > 0)   pq.offer(new Pair(prefix, root.freq));
            for (int i = 0; i < 27; i++) {
                if (root.children[i] != null) {
                    char ch = ' ';
                    if (i != 26)    ch = (char) ('a' + i);
                    addToPQ(root.children[i], pq, prefix + ch);
                }
            }
        }
    }
```

```
}
```

written by MichaelPhelps original link here

From Leetcoder.