Previous Next

642. Design Search Autocomplete System

July 15, 2017 | 72.1K views

Average Rating: 2.57 (49 votes)

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

Approach 1: Brute Force

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_sent, 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_sent and then iterate over all the keys of map to check if a key exists whose initial characters match with cur_sent. 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 Copy

```java
class Node {
  String sentence;
  int times;

  Node(String st, int t) {
    sentence = st;
    times = t;
  }
}

class AutocompleteSystem {
  private HashMap<String, Integer> map = new HashMap<>();
  private String cur_sent = "";

  public AutocompleteSystem(String[] sentences, int[] times) {
    for (int i = 0; i < sentences.length; i++) map.put(sentences[i], times[i]);
  }

  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())
```

Performance Analysis

- `AutocompleteSystem()` takes $O(k*l)$ time. This is because, putting an entry in a hashMap takes $O(1)$ time. But, to create a hash value for a sentence of average length k, it will be scanned atleast once. We need to put l such entries in the map.
- `input()` takes $O\big(n + m \log m\big)$ time. We need to iterate over the list of sentences, in map, entered till now(say with a count n), taking $O(n)$ time, to populate the $list$ used for finding the hot sentences. Then, we need to sort the $list$ of length m, taking $O\big(m \log m\big)$ time.

Approach 2: Using One level Indexing

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 Copy

```java
class Node {
  String sentence;
  int times;

  Node(String st, int t) {
    sentence = st;
    times = t;
  }
}

class AutocompleteSystem {
  private HashMap<String, Integer>[] arr;
  private String cur_sent = "";

  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]);
  }

  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 = "";
```

Performance Analysis

- `AutocompleteSystem()` takes $O(k*l+26)$ time. Putting an entry in a hashMap takes $O(1)$ time. But, to create a hash value for a sentence of average length k, it will be scanned atleast once. We need to put l such entries in the map.
- `input()` takes $O\big(s + m \log m\big)$ time. We need to iterate only over one hashmap corresponding to the sentences starting with the first character of the current sentence, to populate the $list$ for finding the hot sentences. Here, s refers to the size of this corresponding hashmap. Then, we need to sort the $list$ of length m, taking $O\big(m \log m\big)$ time.

Approach 3: Using Trie

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.

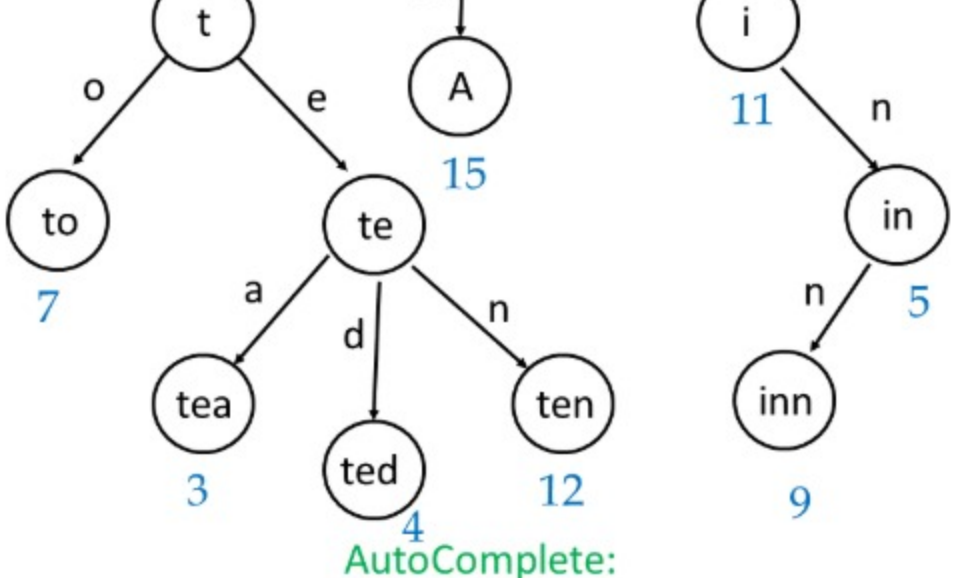
Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

A Trie data structure is very commonly used for representing the words stored in a dictionary. Each level represents one character of the word being formed. A word available in the dictionary can be read off from the Trie by starting from the root and going till the leaf.

By doing a small modification to this structure, we can also include an entry, $times$, for the number of times the current word has been previously typed. This entry can be stored in the leaf node corresponding to the particular word.

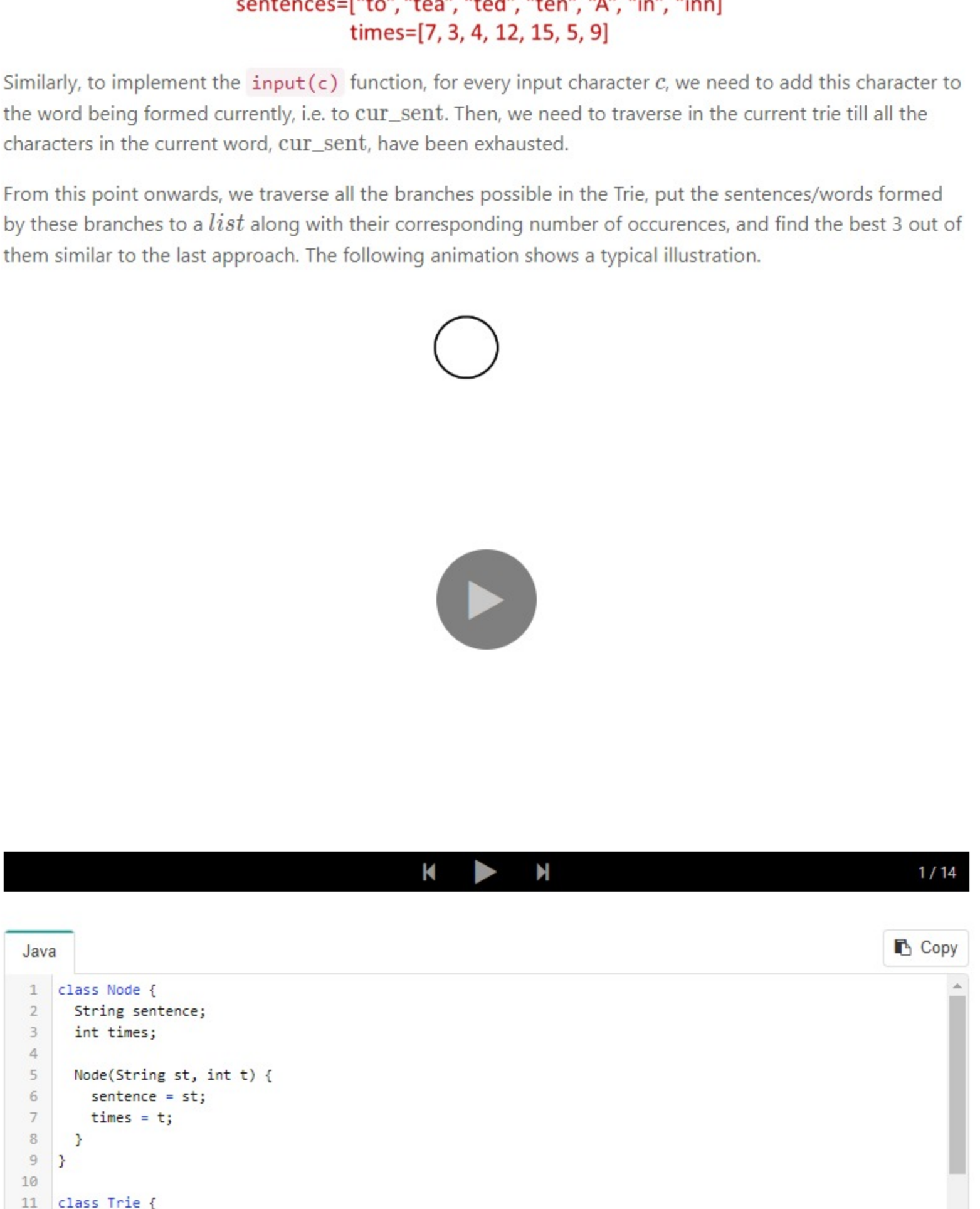
Now, for implementing the `AutoComplete` function, we need to consider each character of the every word given in $sentences$ array, and add an entry corresponding to each such character at one level of the trie. At the leaf node of every word, we can update the $times$ section of the node with the corresponding number of times this word has been typed.

The following figure shows a trie structure for the words "A","to", "tea", "ted", "ten", "i", "in", and "inn", occuring 15, 7, 3, 4, 12, 11, 5 and 9 times respectively.



Similarly, to implement the `input(c)` function, for every input character c, we need to add this character to the word being formed currently, i.e. to cur_sent. Then, we need to traverse in the current trie till all the characters in the current word, cur_sent, have been exhausted.

From this point onwards, we traverse all the branches possible in the Trie, put the sentences/words formed by these branches to a $list$ along with their corresponding number of occurences, and find the best 3 out of them similar to the last approach. The following animation shows a typical illustration.



Java Copy

```java
class Node {
  String sentence;
  int times;

  Node(String st, int t) {
    sentence = st;
    times = t;
  }
}

class Trie {
  int times;
  Trie[] branches = new Trie[27];
}

class AutocompleteSystem {
  private Trie root;
  private String cur_sent = "";

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

  private int toInt(char c) {
```

Performance Analysis

- `AutocompleteSystem()` takes $O(k*l)$ time. We need to iterate over l sentences each of average length k, to create the trie for the given set of $sentences$.
- `input()` takes $O\big(p + q + m \log m\big)$ time. Here, p refers to the length of the sentence formed till now, cur_sent. q refers to the number of nodes in the trie considering the sentence formed till now as the root node. Again, we need to sort the $list$ of length m indicating the options available for the hot sentences, which takes $O\big(m \log m\big)$ time.

Analysis written by: @vinod23

Rate this article:

Previous Next

Comments: 32 Sort By

Type comment here... (Markdown is supported)

Preview Post

lymr 106 January 5, 2019 10:53 AM
The solution code is unreadable! the most weird spacing I ever saw and class members are all over the class. what the heck man ?
98 Share Reply

Cloudson 90 July 1, 2018 6:40 PM
About the third approach, I don't think we need lookup from root to the last character in cur_sent, we could just use an instance variable to record which Trie we are at now.
22 Share Reply
SHOW 2 REPLIES

jsaade 25 April 13, 2018 9:39 PM
Why not save the top 3 items at each trie node.
So when you reach a node you can just return the top 3.
Traverse will be eliminated, lookup would just return t.top3 (which is a saved list).
20 Share Reply
SHOW 4 REPLIES

ramadanizm 22 September 5, 2018 12:43 AM
can some one explain the mergesort here nad the comparable line?
Collections.sort(list, (a, b) -> a.times == b.times ? a.sentence.compareTo(b.sentence) : b.times - a.times);
3 Share Reply
SHOW 1 REPLY

yorkshire 681 December 2, 2017 10:29 AM
@LArch I agree, none of the solutions feels efficient due to repeated sorting and storing duplicated copies of sentences.
How about storing all sentence in a map (sentence to count). Then for a new input create a list sorted by count and repeatedly filter as more chars are seen.
Better explanation and code ... https://discuss.leetcode.com/topic/112705/fast-and-simple-explained-
Read More
2 Share Reply

pintastic 26 August 6, 2017 7:15 AM
For the Trie solution, why don't we:
Remember the node of our last time input, so we don't have to traverse from the root all over again?
2 Share Reply
SHOW 1 REPLY

LArch 34 October 26, 2017 7:19 AM
I think all these 3 solutions are bad.
First one very slow.
Second one weird.
Third one very slow too.
3 Share Reply
SHOW 1 REPLY

azimbabu 108 November 13, 2017 4:18 AM
I think three improvements can be made for the trie solution :

1. avoiding traversal from root for each input invocation by keeping track of current node and current sentence.
2. Using a string buffer instead of string.

Read More
1 Share Reply

nimxor 4 July 28, 2017 3:00 PM
Here we can also optimize the given solution by making a min-priority queue or set of size 3. So that we need not to sort the whole 100 values.
As we need only top 3 most frequent elements. So using a min-priority queue of size 3 is sufficient to do so.
1 Share Reply

mehranangelo 108 December 19, 2018 4:22 AM
Trie with TreeSet fastest Solution Possible

```
class TrieNode{
    TreeSet<Pair> list;
    TrieNode[] child;
```

Read More
1 Share Reply
SHOW 1 REPLY

< 1 2 3 4 >