COMP 3270 SPRING 2020

**Programming Project: Autocomplete**

Name: \_\_\_\_\_\_\_\_\_\_Alexander Lewin\_\_\_ Date Submitted: \_4/8/20\_\_\_\_\_\_\_

1. **Pseudocode**: Understand the strategy provided for *TrieAutoComplete*. State the algorithm for the functions precisely using numbered steps that follow the pseudocode conventions that we use. Provide an approximate efficiency analysis by filling the table given below, for your algorithm.

*Add*

* Pseudocode:   
  **Add**(word: Array [1…n] of chars, double weight>0)   
  1. let curr be a char  
  2. node = RootNode  
  3. node.mySubtreeMaxWeight = Max(weight, node.mySubtreeMaxWeight)

4. for i = 1 to n

5. curr = word[i]

6. if node.hasChild(curr)

7. node = node.getChild(curr)

8. node.mySubtreeMaxWeight = Max(weight, node.mySubtreeMaxWeight)  
9. if i == word.length then

10. node.setWeight(Max(weight, node.getWeight()))

11. node.setWord(word)

12. else

13. node.children.put(curr, Node(curr, node, weight))

14. node = node.getChild(curr)

15. if i == word.length – 1 then

16. node.setWeight(weight)

17. node.setWord(word)

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(n) |
| 5 | O(n) |
| 6 | O(n) |
| 7 | O(n) |
| 8 | O(n) |
| 9 | O(n) |
| 10 | O(n) |
| 11 | O(n) |
| 12 | O(n) |
| 13 | O(n) |
| 14 | O(n) |
| 15 | O(n) |
| 16 | O(n) |
| 17 | O(n) |

Complexity of the algorithm = O(\_n\_)

*topMatch*

* Pseudocode:

**topMatch**(prefix: Array [1…n] of chars)

1. let curr be a char
2. node = myRoot
3. for i = 1 to prefix.length
4. curr = prefix[i]
5. if !node.children.containsKey(curr) then   
    //hashmap -> computes in constant time
6. return ""
7. node = node.getChild(curr)
8. while node.mySubtreeMaxWeight != node.getWeight()
9. for-each temp in node.children.keySet()
10. if node.getChild(temp).mySubtreeMaxWeight == node.mySubtreeMaxWeight then
11. node = node.getChild(temp)
12. return node.getWord()

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(n) |
| 4 | O(n) |
| 5 | O(n) |
| 6 | O(n) |
| 7 | O(n) |
| 8 | O(K), K = Max depth |
| 9 | O(K\*C), C = Max number of children |
| 10 | O(K\*C) |
| 11 | O(1) |
| 12 | O(1) |

Complexity of the algorithm = O(\_n+KC\_)

*topMatches*

* Pseudocode:

**topMatches**(prefix: Array[1…n], k: max number of words to return) {

1. node = myRoot
2. for i = 1 to prefix.length
3. if !node.children.containsKey(prefix[i]) then //hashap -> O(1)
4. return A[ ] //emptyIterator
5. node = node.getChild(prefix[i])
6. let q be an empty priority-queue of nodes that  
    compares mySubtreeMaxWeight of each node
7. q.queue(node) // queue -> add-left, deque -> pop-right
8. count = 0
9. let words[0…k] be a new array of strings
10. while !q.isEmpty() and count < k
11. curr = q.deque()
12. if curr.getWeight() == curr.mySubtreeMaxWeight
13. count++
14. words.queue(curr.getWord())
15. for-each child in curr.children.values()
16. q.queue(child)
17. return words

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(n) |
| 3 | O(n) |
| 4 | O(1) |
| 5 | O(1) |
| 6 | O(1) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(M+k), M = max number of nodes |
| 11 | O(M+k), k = number of words to return |
| 12 | O(M+k) |
| 13 | O(k) |
| 14 | O(k) |
| 15 | O((M+k)\*C), C = max number of children |
| 16 | O((M+k)\*C) |
| 17 | O(1) |

Complexity of the algorithm = O(\_(M+k)\*C\_)

2.**Testing**: Complete your test cases to test the *TrieAutoComplete* functions based upon the criteria mentioned below. **Tests in file called Tester.java**

**Test of correctness:**

Assuming the trie already contains the terms {”ape, 6”, ”app, 4”, ”ban, 2”, ”bat, 3”, ”bee, 5”, ”car, 7”, ”cat, 1”}, you would expect results based on the following table:

|  |  |  |
| --- | --- | --- |
| Query | k | Result |
| ”” | 8 | {”car”, ”ape”, ”bee”, ”app”, ”bat”, ”ban”, ”cat”} |
| ”” | 1 | {”car”} |
| ”” | 2 | {”car”, ”ape”} |
| ”” | 3 | {”car”, ”ape”, ”bee”} |
| ”a” | 1 | {”ape”} |
| ”ap” | 1 | {”ape”} |
| ”b” | 2 | {”bee”, ”bat”} |
| ”ba” | 2 | {”bee”, ”bat”} |
| ”d” | 100 | {} |

3.**Analysis**: Answer the following questions. Use data wherever possible to justify your answers, and keep explanations brief but accurate:

1. What is the order of growth (big-Oh) of the number of compares (in the worst case) that each of the operations in the *Autocompletor* data type make?  
     
   **Add:** O(n), n=size of word  
     
   **TopMatch:** O(n+KC) n=length of prefix, K=Max Depth, C=Max number of children  
     
   **TopMatches:** O((M+K)\*C) M=number of nodes, k=number of words to return, C=max number of children
2. How does the runtime of *topMatches()* vary with k, assuming a fixed prefix and set of terms? Provide answers for *BruteAutocomplete* and *TrieAutocomplete*. Justify your answer, with both data and algorithmic analysis.

* **BruteAutocomplete**:   
  The runtime of topMatches() does not vary with k, when using a brute force approach. The brute force algorithm compares every word in the wordlist, regardless of k.  
  + Time for topKMatches("n", 1) - 0.0068879888583218704
  + Time for topKMatches("n", 4) - 0.0083358745
  + Time for topKMatches("n", 7) - 0.008321767387687189
* **TrieAutocomplete:**The runtime of topMatches() varies with k with a linear relationship, which is evident in the time complexity of topMatches (O(M+K)\*C).  
  + Time for topKMatches("n", 1) - 6.6774E-6
  + Time for topKMatches("n", 4) - 2.31394E-5
  + Time for topKMatches("n", 7) - 4.14381E-5

1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*?

* **topMatch:**   
  Full:
  + Time for topMatch("nenk") - 3.419E-7
  + Time for topMatch("n") - 2.1582E-6
  + Time for topMatch("ne") - 3.6995E-6

Half:

* + Time for topMatch("aenk") - 2.425E-7
  + Time for topMatch("a") - 1.3564E-6
  + Time for topMatch("ae") - 2.2456E-6
* **topMatches:**   
  Source size does affect the runtime of topMatches(). The time complexity of topMatches indicates this because the big O complexity includes the max depth of the trie and the number of children of each node. Both these values are affected by the size of the wordlist.   
  Full:
  + Time for topKMatches("n", 1) - 8.0832E-6
  + Time for topKMatches("ne", 1) - 5.8588E-6
  + Time for topKMatches("notarealword", 1) - 3.467E-7

Half:

* + Time for topKMatches("a", 1) - 5.2135E-6
  + Time for topKMatches("ae", 1) - 3.3605E-6
  + Time for topKMatches("notarealword", 1) - 1.043E-7

4. Graphical Analysis: Provide a graphical analysis by comparing the following:

1. The big-Oh for *TrieAutoComplete* after analyzing the pseudocode and big-Oh for *TrieAutoComplete* after the implementation.  
   Pseudocode:  
   **Add:** O(n), n=Length of word  
   **TopMatch:** O(n+KC) n=length of prefix, K=Max Depth, C=Max number of children  
   **TopMatches:** O((M+K)\*C) M=number of nodes, k=number of words to return, C=max number of children  
     
   Implementation:  **Add:** O(n), n=Length of word  
   **TopMatch:** O(n+KC) n=length of prefix, K=Max Depth, C=Max number of children  
   **TopMatches:** O((M+K)\*C) M=number of nodes, k=number of words to return, C=max number of children
2. Compare the *TrieAutoComplete* with *BruteAutoComplete* and *BinarySearchAutoComplete*.  
   **Note that the Brute solution is the least efficient in every case, followed by the BinarySearch solution. The Trie solution is the most efficient in each case.**