COMP 3270 FALL 2018

**Programming Project: Autocomplete**

Name: Kareith Dyce Date Submitted: 11/26/18

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: String, weight: non-zero double)

1. curr = myRoot;
2. if weight > myRoot.mySubtreeMaxWeight
3. then myRoot.mySubtreeMaxWeight = weight
4. for i to word.length
5. if curr.children.containsKey(word[i]) then
6. if curr.mySubtreeMaxWeight < weight
7. then curr.mySubtreeMaxWeight = weight
8. else
9. n = Node(word[i], curr, weight)
10. curr.children.put(word[i], n)
11. curr = n
12. if i == word.length()-1
13. if curr.isWord
14. curr.mySubtreeMaxWeight = weight
15. curr.setWeight(weight)
16. findMaxWeight(curr)//recursive helper method O(n\*m)
17. curr.setWord(word)
18. curr.setWeight(weight)
19. curr.isWord = true

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(c) |
| 2 | O(c) |
| 3 | O(c) |
| 4 | O(n) |
| 5 | O(c) |
| 6 | O(c) |
| 7 | O(c) |
| 8 | O(c) |
| 9 | O(c) |
| 10 | O(c) |
| 11 | O(c) |
| 12 | O(c) |
| 13 | O(c) |
| 14 | O(c) |
| 15 | O(c) |
| 16 | O(m\*n) |
| 17 | O(c) |
| 18 | O(c) |
| 19 | O(c) |

Complexity of the algorithm = O(n\*m)

*topMatch*

* Pseudocode:

topMatch(prefix: String)

* 1. n = myRoot;
  2. for i to prefix.length
  3. n = n.getChild(prefix[i])
  4. if n == null
  5. return ""
  6. max = n.mySubtreeMaxWeight;
  7. while n!= null
  8. if n.isWord && n.getWeight() == max
  9. return n.getWord()
  10. for-each c in n.children.values()
  11. if max == c.mySubtreeMaxWeight
  12. n = c
  13. break
* Complexity analysis:

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

Complexity of the algorithm = O(m\*n)

*topMatches*

* Pseudocode:

topMatches(prefix : String, k: non-zero integer)

* 1. list = new ArrayList<String>()
  2. q=PriorityQueue<Node>(new Node.ReverseSubtreeMaxWeightComparator())
  3. qT = PriorityQueue<Term>(new Term.ReverseWeightOrder())
  4. look = myRoot
  5. for i in look.length
  6. look = look.getChild(prefix[i])
  7. if look == null{
  8. return list
  9. q.add(look)
  10. while !q.isEmpty() && list.size() < k
  11. n = q.poll()
  12. for-each c in n.children.values()
  13. q.add(c)
  14. if n.isWord&&n.getWeight()!=n.mySubtreeMaxWeight
  15. qT.add(new Term(n.getWord(), n.getWeight()))
  16. else if n.isWord
  17. list.add(n.getWord())
  18. if list.size() == k
  19. return list
  20. if !qT.isEmpty() && !q.isEmpty()
  21. while qT.peek().getWeight()>=q.peek().mySubtreeMaxWeight
  22. list.add(qT.poll().getWord())
  23. if list.size() == k
  24. return list
  25. if qT.isEmpty()
  26. break
  27. return list;
* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(c) |
| 2 | O(c) |
| 3 | O(c) |
| 4 | O(c) |
| 5 | O(c) |
| 6 | O(c) |
| 7 | O(c) |
| 8 | O(c) |
| 9 | O(c) |
| 10 | O(c) |
| 11 | O(c) |
| 12 | O(c) |
| 13 | O(c) |
| 14 | O(c) |
| 15 | O(c) |
| 16 | O(c) |
| 17 | O(c) |
| 18 | O(c) |
| 19 | O(c) |
| 20 | O(c) |
| 21 | O(c) |
| 22 | O(c) |
| 23 | O(c) |
| 24 | O(c) |
| 25 | O(c) |
| 26 | O(c) |
| 27 | O(c) |

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

2.**Testing**: Complete your test cases to test the *TrieAutoComplete* functions based upon the criteria mentioned below.

**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 |
| ”” | - | Car |
| ”a” | - | Ape |
| ”ap” | - | Ape |
| ”b” | - | Bee |
| ”ba” | - | Bat |
| ”c” | - | Car |
| ”ca” | - | Car |
| ”cat” | - | Cat |
| ”d” | - | ”” |
| ” ” | - | ”” |
| ”” | 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?
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.
3. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has all four-letter word combinations from aaaa to mzzz. These datasets provide a very clean distribution of words and an exact 1-to-2 ratio of words in source files.)

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.
2. Compare the *TrieAutoComplete* with *BruteAutoComplete*.