COMP 3270 FALL 2020

**Programming Project: Autocomplete**

Name: Daniel Thomason Date Submitted: 5 November 2020

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: double)

1 current = root

2 for i = 1 to word length

3 if (current’s child does not contain char at i) then

4 temp = new Node(word.charAt(i), current, weight)

5 current.children.put(word.charAt(i), temp)

6 temp.mySubtreeMaxWeight = weight

7 if (current.mySubtreeMaxWeight < weight)

8 current.mySubtreeMaxWeight = weight

9 current = current.children.get(word.charAt(i))

10 current.setWord(word)

11 current.isWord = true

12 current.setWeight(weight)

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(n) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 3 | O(n) |
| 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) |

Complexity of the algorithm = O(n)

*topMatch*

* Pseudocode:

topMatch(prefix: string)

1 if (prefix is null) then throw exception

2 top = “”

3 current = root

4 for i = 1 to prefix length

5 if (current Node’s children doesn’t contain prefix char)

6 return top

7 current = current children prefix at i

8 maxWeight = current subtreeMaxWeight

9 while (current is not a word)

10 for (each Node n that is a child of current)

11 if (n’s subtreeMaxWeight == maxWeight)

12 current = n

13 top = current’s myWord

14 return top

* 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(1) |
| 14 | O(1) |

Complexity of the algorithm = O(n^2)

*topMatches*

* Pseudocode:

topMatches(prefix: string, k: integer where k > 0)

1 if prefix is null then throw exception

2 current = root

3 tops = iterable of strings

4 nodes = priority queue of nodes

5 terms = priority queue of terms

6 for i = 1 to prefix length

7 c = prefix char at i

8 current = current’s child at char c

9 if current is null then return empty list

10 nodes.add(current)

11 while nodes is not empty

12 current = nodes.pop

13 add all of current’s children to nodes

14 if current is a word

15 then term = new Term with current’s weight and word

16 terms.add(term)

17 for each term t in terms

18 if tops’ size <= k then

19 tops.add(t’s word)

20 return tops

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(1) |
| 5 | O(1) |
| 6 | O(n) |
| 7 | O(n) |
| 8 | O(n) |
| 9 | O(n) |
| 10 | O(1) |
| 11 | O(n) |
| 12 | O(n) |
| 13 | O(n) |
| 14 | O(n) |
| 15 | O(n) |
| 16 | O(n) |
| 17 | O(n) |
| 18 | O(n) |
| 19 | O(n) |
| 20 | O(1) |

Complexity of the algorithm = O(n^3)

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 |
| ”” | 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?
   * + BruteAutocomplete: makes O(n) compares
     + BinaryAutocomplete: makes O(n^2) compares
     + TrieAutocomplete: makes O(n^3) compares
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.
   * + For BruteAutocomplete, as k increases, the runtime tends to stay the same. For example, topMatches(“”, 1) has a runtime of 0.00516, topMatches(“”, 4) has a runtime of 0.00587, and topMatches(“”, 7) has a runtime of 0.00553.
     + For TrieAutocomplete, as k increases, the runtime tends to decrease. For example, topMatches(“”, 1) has a runtime of 0.86982, topMatches(“”, 4) has a runtime of 0.78604, and topMatches(“”, 7) has a runtime of 0.79778.
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.)
   * + The prefix size does not increase the time by a substantial amount. However, by decreasing the source size, i.e., the text file that contains the words to be queried, the runtime decreases significantly.

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* and *BinarySearchAutoComplete*.