COMP 3270 FALL 2020

**Programming Project: Autocomplete**

Name: \_\_\_\_\_Carlton Schmieder\_\_\_\_\_\_\_ Date Submitted: \_\_11/11/2020\_\_

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:

1 Node curr = myRoot

2 for i to word.length

3 if weight > curr.mySubtreeMaxWeight

4 curr.mySubtreeMaxWeight = weight

5 if not curr.children.containsKey(word.charAt(i)

6 curr.children.put(word.charAt(i), new Node(word.charAt(i), curr, weight)

7 curr = curr.getChild(word.charAt(i)

8 curr.setWeight(weight)

9 curr.setWord(word

10 curr.isWord equal ture

* Complexity analysis:

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

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

*topMatch*

* Pseudocode:

1 Node curr = myRoot

2 String emptyString = “”

3 for I = 0 to prefix.length

4 if not curr.children.containsKey(word.charAt(i)

5 return emptyString

6 curr = curr.getChild(prefix.charAt(i)

7 double weight = curr.mySubtreeMaxWeight

8 if curr.myWeight = weight

9 return curr.getWord

10 while curr.getWeight != weight

11 for Node tree curr.children.value

12 curr = tree

13 break

14 return curr.getWord()

* Complexity analysis:

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

Complexity of the algorithm = O(\_n^2\_)

*topMatches*

* Pseudocode:

1 Node curr = myRoot

2 PriorityQueue<Node> queue = new PriorityQueue

3 List<String> list = new List

4 for i = 0 to prefix.length

5 if not curr.children.containsKey(prefix.charAt(i)

6 return new ArrayList

7 else

8 curr = curr.getChild(prefix.charAt(i)

9 queue.add(curr)

10 while queue.size() > 0

11 curr = queue.poll()

12 if curr.isWord

13 list.add(new Term(curr.myWord, curr.myWeight)

14 if list < k

15 match.addAll(list)

16 return finish

* Complexity analysis:

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

Complexity of the algorithm = O(\_n^2\_)

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?

Autocompletor is made of three methods which are weightof, topmatch, and topmatches. Weightof is tasked which keeping up with the data and finding the word that is searched. Because of that it stays O(n). Topmatch and topmatches have the same big-Oh because they deal with searching through nodes and arraylist so they require the big-Oh of O(n^2)

1. 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.

With BruteAutocomplete, it is one big array. So, k does not affect the runtime because it searches through the entire array and when it fills up, it compares every element to the first element and does so until the entire list has been searched.

With TrieAutocomplete, k effects how long the runtime will be because as k increases, the runtime increases. Trie uses k to see how many elements can be stored which normally makes the runtime a little longer.

1. 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.)

By increasing the size of the source and increasing the size of the prefic argument actually helps Trie make the matches faster. The bigger the prefix is the less time it takes to find the match that is required. You can see this with the data shown below.

fourletterwords.txt

Table

Description automatically generated

Table

Description automatically generated

fourletterwordshalf.txt

Table

Description automatically generated

Table

Description automatically generated

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.

After I got done with the pseudocode, I determined the big-Oh for it was O(n^2)

Chart, bar chart, histogram

Description automatically generated

1. Compare the *TrieAutoComplete* with *BruteAutoComplete* and *BinarySearchAutoComplete*.

Chart, bar chart

Description automatically generated