COMP 3270 FALL 2018

**Programming Project: Autocomplete**

Name: Paul Chong Date Submitted: 26NOV18 .

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. for i = 1 to wordArray.length

2. if (node.mySubtreeMaxWeight < weight)

3. node.mySubtreeMaxWeight = weight

4. if not (node.children.contains(i))

5. node.children.put(i, child)

6. else

7. node = node.getChild(i)

8. node.setWord(word)

9. node.setWeight(weight)

10. node.isWord = true

* Complexity analysis:

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

Complexity of the algorithm = O(n)

*topMatch*

* Pseudocode:

1. for i = 1 to prefixArray.length

2. if (node.children.contains(i))

3. node = node.getChild(i)

4. else

5. return ""

6. if node.myWeight == largestWordWeight

7. return node.myWord

8. while (node.myWeight != largestWordWeight)

9. for j = 1 to node.children.values

10. if child.mySubtreeMaxWeight==largestWordWeight)

11. node = child

12. return node.myWord

* Complexity analysis:

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

Complexity of the algorithm = O(n2)

*topMatches*

* Pseudocode:

1. PriorityQueue nodeList = new PriorityQueue

2. List<String> wordsList = new List

3. for i = 1 to prefixArray.length

4. if (node.children.contains(i))

5. node = node.getChild(i)

6. else

7. return wordsList

8. nodeList.add(node)

9. while (nodeList.size() > 0)

10. node = nodeList.poll()

11. if (node.isWord)

12. wordsList.add(node.myWord)

13. if (wordsList.size() >= k)

14. break

15. nodeList.addAll(node.children)

16. return wordsList

* Complexity analysis:

|  |  |
| --- | --- |
| Step # 1 | Complexity stated as O(1) |
| Step # 2 | Complexity stated as O(1) |
| Step # 3 | Complexity stated as O(n) |
| Step # 4 | Complexity stated as O(1) |
| Step # 5 | Complexity stated as O(1) |
| Step # 6 | Complexity stated as O(1) |
| Step # 7 | Complexity stated as O(1) |
| Step # 8 | Complexity stated as O(1) |
| Step # 9 | Complexity stated as O(n2) |
| Step # 10 | Complexity stated as O(1) |
| Step # 11 | Complexity stated as O(1) |
| Step # 12 | Complexity stated as O(1) |
| Step # 13 | Complexity stated as O(1) |
| Step # 14 | Complexity stated as O(1) |
| Step # 15 | Complexity stated as O(1) |
| Step # 16 | Complexity stated as O(1) |

Complexity of the algorithm = O(n2)

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

My test cases have been attached as “AutocompleteTest.java” because I did not know what other way I could show my tests.

**Test of correctness:**

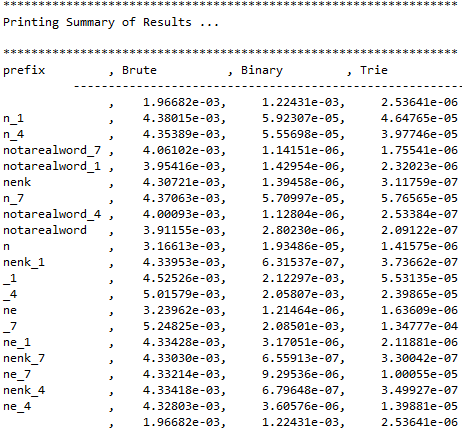
Assuming the trie already contains the terms {”ape, 6”, ”app, 4”, ”ban, 2”, ”bat, 3”, ”bee, 5”, ”car, 7”, ”cat, 1”}, you would expect results based on the following table:

|  |  |  |
| --- | --- | --- |
| 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 | {} |

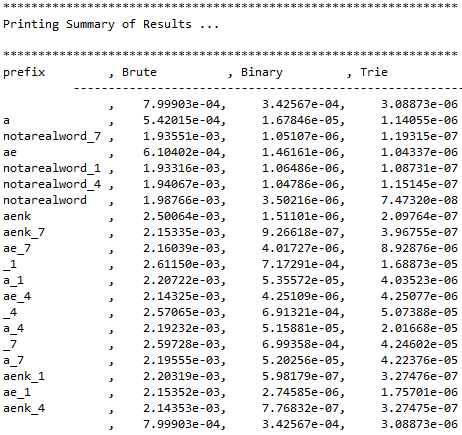
3.**Analysis**: Answer the following questions. Use data wherever possible to justify your answers, and keep explanations brief but accurate:

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?
   * The Autocompleter data structure consists of three methods: topMatches, topMatch, and weightOf. As the number of comparisons or the dataset for these comparisons grows, weightOf stays at O(n) because its only job is to go through at least n number of words and get the weight of the specified word (if it exists). TopMatch and TopMatches grow as O(n2) because regardless of the comparisons, their nature of checking nodes and lists twice or more will always prevent them from reaching O(n) and O(log(n)).
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: It does not matter what k is because it runs through the entire word list regardless. Brute simply has a giant array and goes through each element one by one. Once the list becomes full, it compares the element to the first element on the list since it’s sorted. It does this over and over again so k does not affect it.
   * TrieAutocomplete: As k increases, runtime increases. The trie structure loops looking for k extra elements, so a bigger k would make the time to find the element longer and this would in turn cause the length of time that the loop runs to be longer as more words are added to the word list.
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.)

**fourletterwords.txt**



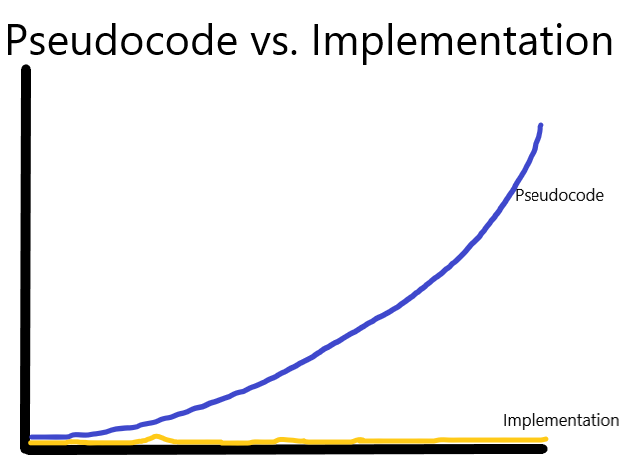
**fourletterwordshalf.txt**



* + By viewing the charts above for both fourletterwords and fourletterwordshalf, we can conclude that increasing the prefix argument and increasing the source size causes TrieAutocomplete to make matches quicker. However, it should be noted that the time differences within the Trie data structures in the diagrams are very slim and almost negligible as they operate within the 10^-6 to 10^-8 numbers unlike brute and binary.

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.
   1. My pseudocode for *TrieAutoComplete* produced a time complexity of O(n2). After the implementation, the graphs that are shown below in 4(ii) seem to show an almost O(1) time complexity. However, by Googling the time complexity of a Trie data structure (source: <https://www.quora.com/What-is-the-complexity-of-Trie>), it seems that the complexity in reality is O(N \* M), where N and M are two separate parts of a dataset. Thus a true depiction of the comparisons between the pseudocode and implementation may be difficult. However, with what we have so far, this is a comparison between the pseudocode and my implementation.



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