COMP 3270 FALL 2021

**Programming Project: Autocomplete**

Name: Mohamad Salaam Date Submitted: 10/28/2021

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. If word = null,
2. throw pointer exception
3. If weight < 0,
4. throw illegal argument exception
5. currentNode = root;
6. for i to word.length
7. index = character at index i
8. nextNode = child of index character
9. If nextNode = null
10. Instantiate nextNode with character index
11. Parent currentNode & weight
12. Add node to children
13. If max weight < weight
14. currentNode.mySubtreeMaxWeight = weight
15. Set word value
16. Set weight
17. Set currentNode as a word

* Complexity analysis:

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

Complexity of the algorithm = O( n )

*topMatch*

* Pseudocode:

1. If word = null,
2. throw pointer exception
3. for i to word.length
4. if currentNode doesn’t contain character at i
5. Return “”
6. currentNode = character at i
7. While (currentNode is not a word & weight < currentNode’s max weight)
8. For i to k
9. If max weight at k index = currentNode’s max weight
10. CurrentNode = node(k)
11. Break
12. Return currentNode’s word

* Complexity analysis:

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

Complexity of the algorithm = O( n^2 )

*topMatches*

* Pseudocode:

1. If word = null,
2. throw pointer exception
3. If k = 0
4. Throw exception
5. currentNode = root
6. new priority queue NodeQ
7. new Arrailist Terms
8. for i to charArray.length
9. if char at i = null
10. Return empty array
11. If currentNode.isWord = true
12. Add the term to nodeQ
13. For x to currentNode.values
14. If x != null
15. Offer node to nodeQ
16. While (nodeQ > 0)
17. currentNode = nodeQ.poll()
18. If currentNode.isWord = true
19. Add term to NodeQ
20. For x to NodeQ.values
21. If x != nul
22. Offer node to NodeQ
23. Sort Terms in reverse order
24. Create a new ArrayList called ‘output’ with term datatypes
25. If k > terms.size,
26. add all terms from ‘terms’ to ‘output’
27. Else,
28. add terms from index 1…k in ‘terms’ to ‘output’
29. Return outputStrings arrayList

* Complexity analysis:

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

Add method: O(n)

Top match: O(N^2)

Top matches 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.

BruteAutocomplete: This stays relatively the same and is not affected significantly by the size of k

TrieAutocomplete: here the size of k and runtime are dependant, as k increases the runtime increases. But, it is still faster than BruteAutocomplete due to the break

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.)

BruteAutocomplete: runtime increases linearly according to size of source while size of prefix has almost no affect

TrieAutocomplete: the runtime decreases in the event of increasing the size of both source and prefix

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*.