COMP 3270 FALL 2019

**Programming Project: Autocomplete**

Name: \_\_\_\_Maggie Blanton\_\_\_\_\_\_\_\_\_\_\_ Date Submitted: \_\_\_11-22-19\_\_\_\_

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, weight)**

1 curr node = root

2 if curr is not word

3 for each char in word

4 get child

5 if child = null

6 add Node with k, curr, weight to children

7 if max weight < weight

8 update MaxWeight to weight

9 set isWord

10 set word

11 set weight

12 else

13 if max weight < weight

14 update MaxWeight to weight

* Complexity analysis:

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

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

*topMatch*

* Pseudocode:

**TopMatch(prefix)**

1 curr node = root

2 for all characters in prefix

3 get child

4 if child is null, return empty string

5 while maxWeight != weight

6 for each child of current

7 if current maxWeight = child maxWeight

8 current = child

9 exit loop

10 return current word

* Complexity analysis:

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

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

*topMatches*

* Pseudocode:

**TopMatches(prefix, k)**

1 curr node = root

2 create empty list

3 if k = 0, return empty list

4 for each char in prefix

5 get child

6 return empty list if child is null

7 PriorityQueue of nodes sorted by mySubtreeMaxHeight (PQ1)

8 PriorityQueue of terms sorted by weight (PQ2)

9 Push current node to PQ1

10 while PQ is not empty and top k matches haven’t been found

11 Pop current node of PQ1

12 if node is word, add to PQ2

13 for each child of current node

14 Push value to PQ1

15 for each value in PQ2

16 add value to LinkedList

17 return list

* Complexity analysis:

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

Add: O(n)

topMatches: O(n^2)

topMatch: 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.

As K increases, Brute runtime shows little change. See below:

for topKMatches("", 1) -  0.002429379496  
for topKMatches("", 4) -  0.00225641581  
for topKMatches("", 7) -  0.002310189872

As K increases, Trie runtime increases. See below:

Time for topKMatches("ae", 1) -  3.926363E-6  
Time for topKMatches("ae", 4) -  4.477321E-6  
Time for topKMatches("ae", 7) -  5.272512E-6

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.)

Fourletterwordshalf.txt ran much faster overall. Ergo, increasing the size of the source slows down both of these methods.

Increasing the size of the prefix increases the number of calls to getChild for both topMatch and topMatches.

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*.

|  |  |  |  |
| --- | --- | --- | --- |
| prefix | Brute | Binary | Trie |
| kh\_4 | 4.25E-03 | 3.63E-06 | 2.21E-05 |
| k\_4 | 4.27E-03 | 8.42e-05 | 8.53E-05 |
| k\_7 | 4.25E-03 | 6.76E-05 | 8.37E-05 |
| notarealword\_7 | 3.84E-03 | 1.07E-06 | 4.97E-07 |
| notarealword\_1 | 3.94E-03 | 2.06E-06 | 4.77E-07 |
| kh\_7 | 4.25E-03 | 3.87E-06 | 4.21E-05 |
| notarealworld\_4 | 3.84E-03 | 1.34E-06 | 4.64E-07 |
| notarealworld | 3.42E-03 | 5.99E-06 | 6.50E-07 |
| khombu | 2.52E-03 | 5.25E-06 | 6.74E-07 |
|  |  |  |  |

Brute is the slowest. Binary and Trie alternate.