COMP 3270 FALL 2021

**Programming Project: Autocomplete**

Name: \_Kirpa Chahal\_\_\_\_\_\_\_\_\_\_\_ Date Submitted: \_\_Oct 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:

Add(String word, doube weight)

1 if word null

2 throw exception

3 if weight<0

4 throw exception

5 current node=tree root

6 next node=null

7 index =’’

8 for k to wrd.length

9 index=word at char index k

10 next node=current node child

11 if next node=null

12 instantiate next node

13 parent current node and weight

14 if current subtree top weight<weight

15 current node subtree top weight=weight

16 current node point to next node

17 set current node word val

18 set current node weight

19 set word is current node

* Complexity analysis:

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

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

*topMatch*

* Pseudocode:

topMatch(prefix String)

1. If prefix==null
2. Throw exception
3. Node Queue= new queue with a
4. Reverse weight comparator
5. Index=’’
6. Bool realWrd=true
7. Empty String=””
8. Top match String=””
9. For j to prefix len
10. Index= char at j in word
11. If current node child has val
12. Current node point to child
13. Else
14. realWrd=false
15. If(realWrd==false)
16. Return empty
17. While current node is wrd and current node weight<current node subtree top weight
18. For current node children
19. Add to node queue
20. Point current node to head of queue
21. topMatch =. Current node word val
22. Return topmatch

* Complexity analysis:

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

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

*topMatches*

* Pseudocode:

topMatches(string prefix, int k)

1 if prefix==null

2 throw exception

3 current node=tree root

4 index =’’

5 bool realWrd = true

6 emptyL=new Array

7 topMatches=new ArrayList nodes

8 topMatchesStr = new string arrList

9 if j=0

10 return emptyL

11 for k to prefix.len

12 index=char of word at k

13 current node point to child with index val

14 if current node==null

15 return emptyL

16 if current node!=null

17 add current node to node queue

18 while node queue no empty

19 current node point to head of node queue & remove from queue

20 for each node in current node children

21 add node to node queue

22 if topMatch size>=j

23 descending order sort of topMatches

24 if current node point at wrd

25 add current node to top match

26 for nodes in top match

27 add nodes word to topMatchStr

28 return topMatchStr

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O (1) |
| 2 | O (1) |
| 3 | O (1) |
| 4 | O (1) |
| 5 | O (1) |
| 6 | O (1) |
| 7 | O (1) |
| 8 | O (1) |
| 9 | O (1) |
| 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 (n^2) |
| 21 | O (n^2) |
| 22 | O (n) |
| 23 | O (n logn) |
| 24 | O (1) |
| 25 | O (n) |
| 26 | O (n) |
| 27 | O (n) |
| 28 | O (n) |
| 29 | 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)

topMatch O(n^2)

topMatches 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.

A picture containing calendar

Description automatically generated

Got data from benchmark class with fourletterword text file.

BruteAutocomplete: topMatches runtime is same for term set and prefix. The runtime is not impacted by k

TrieAutocomplete: topMatches runtime increases as k increases. Larger k values lead to more runtime.

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.)

In data tables below

fourletterwords.txt data(on left), fourletterwordsshalf.txt (on right)

BruteAutoComplete: runtime of topMatch is increased with size. This is linear

TrieAutocomplete: runtime is not changed by increasing size. This is more scalable

A picture containing calendar

Description automatically generatedA picture containing calendar

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.

Before:

Add: O(n)

topMatch: O(n^2)

topMatches: O(n^2)

After:

Add: O(n)

topMatch: O(n^2)

topMatches: O(n^2)

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

TrieAutocomplete is obviously more efficient than the Brute Autocomplete.