COMP 3270 SPRING 2020

**Programming Project: Autocomplete**

Name: Date Submitted: 4/6/2020

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:

Add(word:String, weight:double)

1. if word is null

2. throw an exception

3. if weight < 0

4. throw an exception

5. char indicator = “”

6. Node location = root of the tree

7. For a to word.length

8. Indicator = the character at index a

9. Node next = the child node of location with the

character indicator

10. If next is null

11. Initialize next with the character index, parent location, and given weight

12. Add the new child to the tree

13. If the weight > location’s subtree max weight

14. Location’s subtree max weight = weight

15. Point location to next

16. Set location’s word value

17. Set location’s weight

18. Set location to be an actual word

* 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(n) |
| 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(1) |
| 17 | O(1) |
| 18 | O(1) |

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

*topMatch*

* Pseudocode:

topMatch(prefix: string)

1. if prefix is null

2. throw an exception

3. Boolean match = true

4. char indicator = “”

5. Node location = root of the tree

6. string emptyString = “”

7. string bestMatchFound = “”

8. nodePQ = new priority queue of nodes using

a reverse weight comparator

9. for a to prefix.length

10. indicator = the character in the word at index a

11. if a child of location contains the value of indicator

12. location points to that child

13. else

14. match = false

15. break

16 if match is false

17. return emptyString

18. while location’s subtree max weight > location’s weight and

location is a word

19. for every node within location’s children

20. add that node into nodePQ

21. point location to the head of the queue and then remove the element from the queue

22. bestMatchFound = location’s word value

23. return bestMatchFound

* 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(n) |
| 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(prefix: String; k: int)

1. if prefix is null

2. throw an exception

3. char indicator = “”

4. Node location = root of the tree

5. wordAList = new ArrayList of Nodes

6. finalist = new ArrayList of Strings

7. output3 = new ArrayList of Strings

8. nodePQ = new PriorityQueue of Nodes

9. for a to prefix.length

10. indicator = the character in the word at the index a

11. location points to the child with the value of a

12. if location is null

13. return output3

14. if nodePQ is not null

15. add location to nodePQ

16. while nodePQ is not empty

17. if the size of wordAList >= k

18. sort wordAList in descending order of the values

19. break

20. set location to point to the head of the queue and then remove the element from the queue

21. if location is a word

22. add location to wordAList

23. for each Node within location’s children

24. add the node into nodePQ

25. for b to the smallest value between k and wordAList’s size

26. add the word that is located at b to finalist

27. return finalList

* 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(1) |
| 15 | O(1) |
| 16 | O(n) |
| 17 | O(n) |
| 18 | O(nlog(n)) |
| 19 | O(1) |
| 20 | O(n) |
| 21 | O(n) |
| 22 | O(n) |
| 23 | O(n^2) |
| 24 | O(n^2) |
| 25 | O(n) |
| 26 | O(n) |
| 27 | 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 | {} |

**All test cases pass**

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

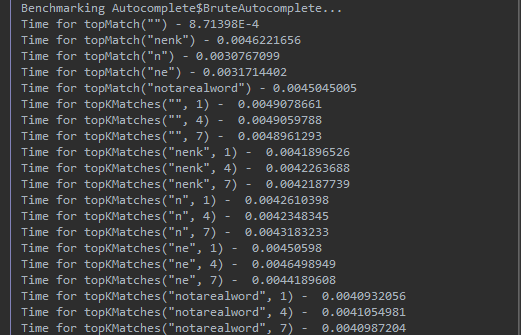
1. What is the order of growth (big-Oh) of the number of compares (in the worst case) that each of the operations in the *Autocompletor* data type make?

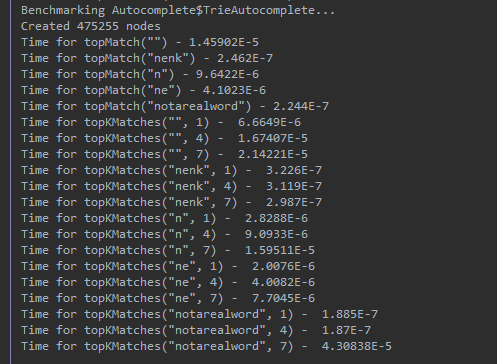
Add: O(n)

topMatch: O(n^2)

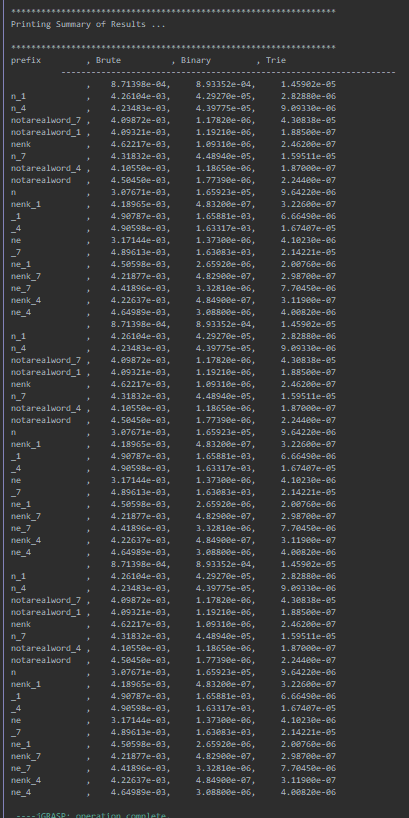
topMatches: O(n^2)

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.

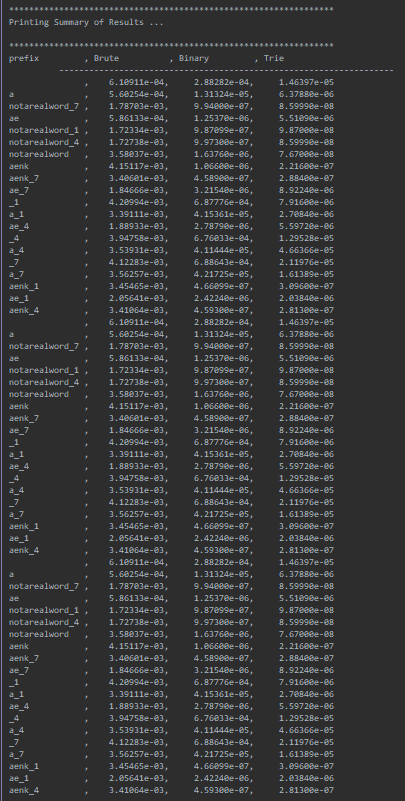




1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has 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.)

**Fourletterwords.txt**

**Fourlettershalf.txt**



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*.

As the charts will show, the overall efficiency of the TrieAutocomplete is much faster and greater than both Brute and Binary Autocomplete