COMP 3270 FALL 2021

**Programming Project: Autocomplete**

Name: **Chase Hopkins**

Date Submitted: **10/29/2021**

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 is null**
2. **throw an exception**
3. **if weight < 0**
4. **throw an exception**
5. **char character = “”**
6. **Node temp = root node of the tree**
7. **For a to word.length**
8. **Character = the character at index a**
9. **Node next = the child node of temp with the**

**character character**

1. **If next is null**
2. **Initialize next with the index of the character, parent temp, and given weight**
3. **Add the new child to tree**
4. **If the weight > temp subtree max weight**
5. **temp subtree max weight = weight**
6. **Point temp to next**
7. **Set temp node word value**
8. **Set temp node weight**
9. **Set temp node to that of a 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:

**1. if prefix is null**

**2. throw an exception**

**3. Boolean matchFound = true**

**4. char character = “”**

**5. Node temp = root node of the tree**

**6. string emptyString = “”**

**7. string bestMatchFound = “”**

**8. priorityQueue = new priority queue of nodes using**

**a reverse weight comparator**

**9. for a to prefix.length**

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

**11. if a child of temp contains the value of character**

**12. temp points to that child**

**13. else**

**14. matchFound = false**

**15. break**

**16 if matchFound is false**

**17. return emptyString**

**18. while temp’s subtree max weight > temp’s weight and**

**temp is a word**

**19. for every node within temp’s children**

**20. add that node into priorityQueue**

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

**22. bestMatchFound = word value of temp node**

**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(n²)** |
| **20** | **O(n²)** |
| **21** | **O(n)** |
| **22** | **O(1)** |
| **23** | **O(1)** |

Complexity of the algorithm = **O(n²)**

*topMatches*

* Pseudocode:

**1. if prefix is null**

**2. throw an exception**

**3. char character = “”**

**4. Node temp = root of the tree**

**5. wordList = new ArrayList of Nodes**

**6. output = new ArrayList of Strings**

**7. noWordsList = new ArrayList of Strings**

**8. priorityQueue = new PriorityQueue of Nodes**

**9. for a to prefix.length**

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

**11. temp points to the child with the value of i**

**12. if temp is null**

**13. return noWordsList**

**14. if priorityQueue is not null**

**15. add temp to priorityQueue**

**16. while priorityQueue is not empty**

**17. if the size of wordList >= k**

**18. sort wordList in descending order of the values**

**19. break**

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

**21. if temp is a word**

**22. add temp to wordList**

**23. for each node within temp nodes children**

**24. add the node into priorityQueue**

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

**26. add the word that is located at b to output**

**27. return output**

* 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(nlogn)** |
| **19** | **O(1)** |
| **20** | **O(n)** |
| **21** | **O(n)** |
| **22** | **O(n)** |
| **23** | **O(n²)** |
| **24** | **O(n²)** |
| **25** | **O(n)** |
| **26** | **O(n)** |
| **27** | **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.

**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 | {} |

**\*\*passed all test cases\*\***

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?
2. **Add: O(n)**
3. **topMatch: O(n²)**
4. **topMatches: O(n²)**
5. 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.

Chart, bar chart

Description automatically generated

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.)

Chart

Description automatically generated

A picture containing bar chart

Description automatically generated

A picture containing chart

Description automatically generated

Chart, table

Description automatically generated with medium confidence

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.

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

* **According to the data shown above, it is apparent that Trie is far more efficient than that of Brute and Binary.**