COMP 3270 FALL 2019

**Programming Project: Autocomplete**

Name: \_\_\_\_\_\_\_\_\_Jiyeon Lee\_\_\_\_\_\_\_\_\_\_\_ Date Submitted: \_\_11/22/2019\_\_

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 == null then throw new NullPointerException

(“Word argument is null.”)

1. if weight < 0 then throw new IllegalArgumentException

(“Weight is not a proper value.”)

1. if myRoot.mySubtreeMaxWeight < weight then

myRoot.mySubtreeMaxWeight = weight

1. myNode = myRoot
2. for i = 1 to word.length()
3. myChar = word.charAt(i)
4. if not myNode.children.containsKey(myChar) then
5. n = new Node(myChar, myNode, weight)
6. myNode.children.put(myChar, n)
7. myNode.myInfo += myChar
8. if myNode.mySubtreeMaxWeight < weight then
9. myNode.mySubtreeMaxWeight = weight
10. myNode = myNode.getChild(myChar);
11. myNode.isWord = true
12. myNode.setWord(word)
13. myNode.myWord = word
14. myNode.setWeight(weight)
15. myNode.myWeight = weight

* Complexity analysis:

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

Complexity of the algorithm = O(n)

*topMatch*

* Pseudocode:

topMatches(prefix: String)

1. If prefix is null then throw new NullPointerException()
2. myNode = myRoot
3. for i = 1 to prefix.length()
4. if myNode.getChild(prefix.charAt(i)) is null then
5. myNode = myNode.getChild(prefix.charAt(i))
6. if myNode.isWord and myNode.getWeight() is

myNode.mySubtreeMaxWeight then

1. return myNode.getWord();
2. else
3. return “”;
4. while(myNode.getWeight() is not myNode.mySubtreeMaxWeight)
5. for each ch is in myNode.children.values()
6. if ch.mySubtreeMaxWeight is

myNode.mySubtreeMaxWeight then

1. myNode = ch
2. break;
3. return myNode.getWord()

* 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(n) |
| 10 | O(n\*m) |
| 11 | O(n\*m) |
| 12 | O(n\*m) |
| 13 | O(n\*m) |
| 14 | O(n\*m) |
| 15 | O(1) |
|  |  |
|  |  |

Complexity of the algorithm = O(n\*m)

*topMatches*

* Pseudocode:

topMatches(prefix: String, k: int)

1. If prefix is null then throw NullPointerException()
2. myNode = myRoot
3. matches = new ArrayList<String>()
4. for i=1 to prefix.length()
5. myNode = myNode.getChilde(prefix.charAt(i))
6. if myNode is null then return matches
7. qC = newPriorityQueue<Node>(k,

new Node.ReverseSubtreeMaxWeightComparator())

1. PriorityQueue<Node> q = new PriorityQueue<Node>(k)
2. qC.add(myNode)
3. while(qC is not empty) {
4. myNode = qC.remove()
5. if myNode.isWord && q.size() and k and q.peek().getWeight() <

myNode.mySubtreeMaxWeight) then

1. q.poll()
2. q.add(myNode)
3. else if myNode.isWord then
4. q.add(myNode)
5. for each ch in myNode.children.values()
6. qC.add(ch)
7. Node[] ns = new Node[q.size()]
8. for int i = 1 to ns.length
9. ns[i] = q.poll()
10. for i = ns.length down to 1 and ns.length – k + 1
11. matches.add(ns[i].getWord())
12. return matches

* Complexity analysis:

|  |  |
| --- | --- |
| * Step Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(1) |
| 4 | O(1) |
| 5 | O(m) |
| 6 | O(m) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(k) |
| 11 | O(k) |
| 12 | O(k) |
| 13 | O(k) |
| 14 | O(k) |
| 15 | O(k) |
| 16 | O(k) |
| 17 | O(m\*k) |
| 18 | O(m\*k) |
| 19 | O(1) |
| 20 | O(k) |
| 21 | O(k) |
| 22 | O(k) |
| 23 | O(k) |
| 24 | O(1) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Complexity of the algorithm = O(m\*k)

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 | {} |

I have tested the TrieAutoComplete functions with the criteria by using the equals() method and printing true or false.

e.g.) with the TrieAutoComplete instance called trie.

trie.topMaches(“a”, 1).toString().equals(“[ape]”) results true

since the largest weight is “ape.”

But, trie.topMaches(“b”, 2).toString().equals(“[bee, bat]”)

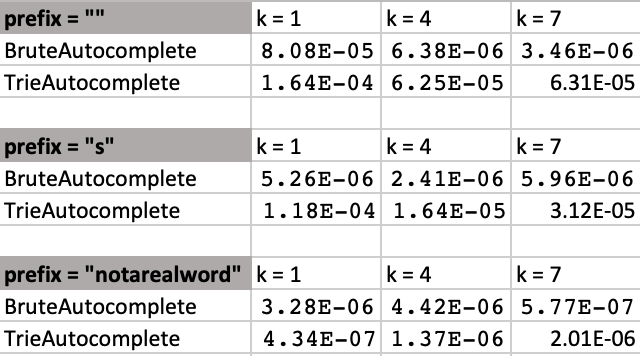
results false because it is supposed to be “[“bat, “ban].”

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) because each node needs to be set the values of myWord, myInfo, isWord, myWeight, and mySubtreeMaxWeight.
* topMatch: O(n\*m) since each node will be checked if the weight and mySubtreeMaxWeight, and for loop finds a child node that has the same mySubtreeMaxWeight as the current node.
* topMatches: O(m\*k) since there is k number capacity and each child is going to be checked until the priority queue is not empty.

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.

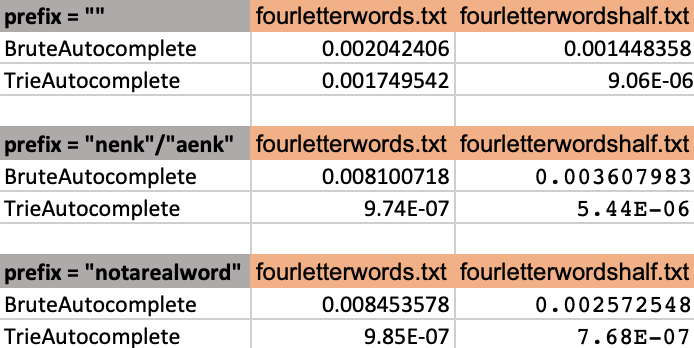


* Answer: When k increases, the time also increase.

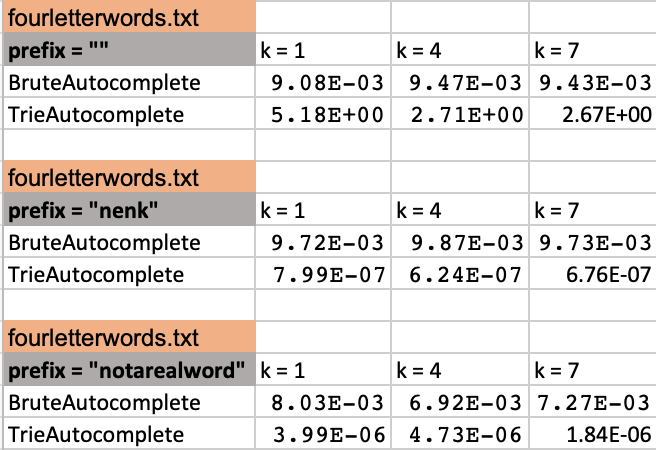
If there is no such prefix exist, and k increases, the TrieAutocomplete time will decrease whereas the BruteAutocomplete time will increase.

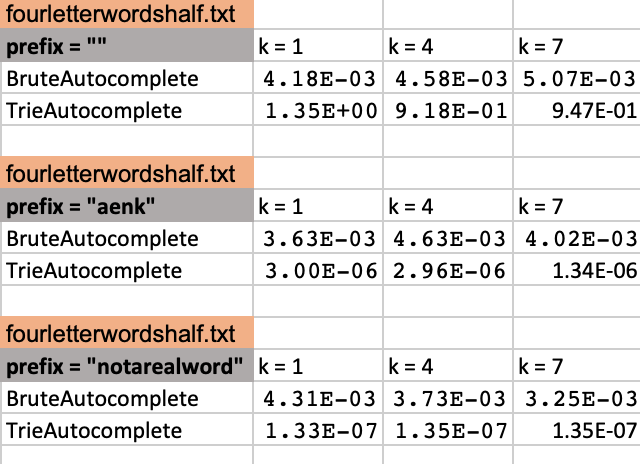
1. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has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

* topMatch



* topMatches

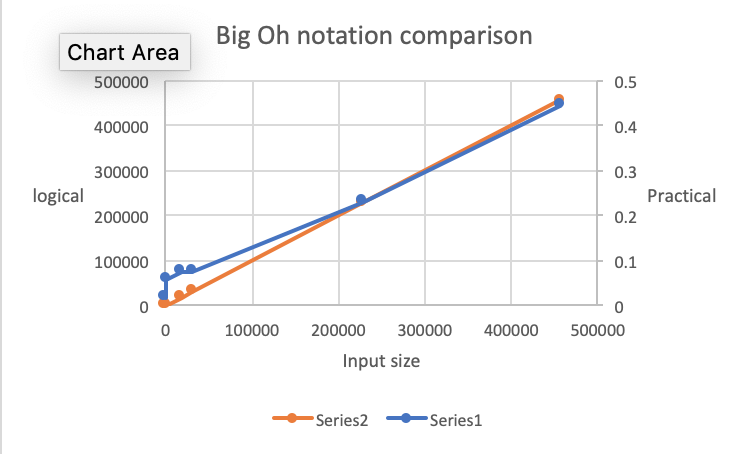




* Answer: When increasing source, the runtime is also increased in BruteAutocomplete. But, the runtime of the TrieAutocomplete is decreased when increasing source and the number of prefix letter.

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.



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