COMP 3270 FALL 2018

**Programming Project: Autocomplete**

Name: \_\_\_Demarcus Campbell\_\_\_\_\_\_ Date Submitted: \_\_\_\_11/23/18\_\_

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. If ( word == null )
   1. throw NullPointerException
2. if ( weight < 0 )
   1. throw IllegalArgumentException
3. Size = 0
4. currentChar = word[0]
5. currentNode = myRoot
6. While ( size < word.length ):
   1. currentChar = word[size]
   2. if (currentNode.getChild( currentChar) == null AND weight > currentNode.weight ):
      1. x = Node(currentChar, currentNode, weight)
      2. currentNode.children.add(currentChar, x)
      3. currentNode = x
   3. else if( currentNode.getChild(currentChar) == null ):
      1. x = Node( currentChar, currentNode, currentNode.weight ):
      2. currentNode.children.add(currentChar, x)
      3. currentNode = x
   4. else:
      1. currentNode= currentNode.getChild(currentChar)
      2. if( currentNode.weight < weight )
         1. currentNode.weight = weight
   5. size += 1
   6. currentNode.word = word
   7. currentNode.isWord = true

* Complexity analysis:

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

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

*topMatch*

* Pseudocode:

topMatch(prefix):

1 if( prefix == null ):

2 throw NullPointerException

3 CurrentRoot = myRoot

4 for( x = 0 to prefix.length ):

5 CurrentRoot = CurrentRoot,getChild( prefix[x] )

6 if( CurrentRoot == null ):

7 return “”

8 PriorityQueue finish

9 PriorityQueue visit = CurrentRoot.children.values()

10 while ( visit.size() > 0 ):

11 temp = visit.pop()

12 for-each( x in temp.children.values() ):

13 if( x.isword ):

14 if( finish.size() < 1 ):

15 finish.push(x)

16 else-if( finish.peek() < x ):

17 finish.pop()

18 finish.push(x)

19 if( x.children.size() > 0 ):

20 visit.push( x.children.values() )

21 if( finish.size() < 1 ):

22 return “”

23 return finish.peek().myWord

* Complexity analysis:

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

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

*topMatches*

* Pseudocode:

topMatches(prefix, k):

1 if( prefix == null ):

2 throw NullPointerException

3 CurrentRoot = myRoot

4 for( x = 0 to prefix.length ):

5 CurrentRoot = CurrentRoot,getChild( prefix[x] )

6 if( CurrentRoot == null ):

7 return “”

8 PriorityQueue finish

9 PriorityQueue visit = CurrentRoot.children.values()

10 while ( visit.size() > 0 ):

11 temp = visit.pop()

12 for-each( x in temp.children.values() ):

13 if( x.isword ):

14 if( finish.size() < k ):

15 finish.push(x)

16 else-if( finish.peek() < x ):

17 finish.pop()

18 finish.push(x)

19 if( x.children.size() > 0 ):

20 visit.push( x.children.values() )

21 for-each( x : finish )

22 ArrayList temp.add(x)

23 return temp

* Complexity analysis:

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

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

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 |
| ”” | - | Car |
| ”a” | - | Ape |
| ”ap” | - | Ape |
| ”b” | - | Bee |
| ”ba” | - | Bat |
| ”c” | - | Car |
| ”ca” | - | Car |
| ”cat” | - | Cat |
| ”d” | - | ”” |
| ” ” | - | ”” |
| ”” | 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?

The largest Big-Oh of the Autocompletor is n2, while most operations are Big-Oh of n.

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.

For BruteAutocomplete, as k increases the runtime shows a slight decrease as seen in the Benchmark. This is supported by the fact that in the algorithm for TopMatches within the BruteAutocomplete class, it minimizes the amount of iterations needed for the final loop, and this starts to make large differences when k is larger than the actual list to be returned.  
  
For TrieAutocomplete, as k increases the runtime will typically decrease, then show a leveling off. This can also be seen in the runtimes given by the Benchmark. This should be supported algorithmically, because of my usage of ArrayLists and for-each loops, which means only the exact amount of loop iterations for the final loop will be executed.

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.)

The runtime of TopMatch and TopMatches does not change too much even though the source amount is increased.

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*.

The TrieAutocomplete has a larger Big-Oh because at worse case it is of n2 where as BruteForce is linear.