COMP 3270 FALL 2018

**Programming Project: Autocomplete**

Name: \_\_\_\_\_\_\_\_\_\_\_\_pengfei yao\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date Submitted: \_\_\_\_\_\_\_\_\_\_\_\_\_

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. private void add(String word, double weight) {
2. // TODO: Implement add
3. if (word == null) throw new NullPointerException("word is null!");
4. if (weight < 0) throw new IllegalArgumentException("we dont wanna weight be negative");
5. //set weight to the root, if weight > curr weight
6. if(myRoot.mySubtreeMaxWeight < weight) {
7. myRoot.mySubtreeMaxWeight = weight;
8. }
9. //updata weight of root
10. Node curr = myRoot;
11. for (int i = 0; i < word.length(); i++){
12. //change weight for each chil nodes in for loop
13. if(curr.mySubtreeMaxWeight < weight) {
14. curr.mySubtreeMaxWeight = weight;
15. }
16. //once we change the weight of current node
17. // we need check wether curr exits
18. //if curr doesnt exits chil node with next char, we need create new one.
19. if(!curr.children.containsKey(word.charAt(i))){
20. curr.children.put(word.charAt(i), new Node(word.charAt(i),curr, weight));
21. curr.myInfo = curr.myInfo + word.charAt(i);
22. }
23. // move to next node
24. curr = curr.children.get(word.charAt(i));
25. }// change, add for loop above
26. /\*if(curr.mySubtreeMaxWeight < weight) {
27. curr.mySubtreeMaxWeight = weight;
28. }\*/
29. // set value of isWord, myWord and setWord for all word nodes
30. curr.isWord = true;
31. curr.myWord = word;
32. curr.setWord(word);
33. //this curr is very last one letter
34. if(curr.isWord && curr.myWeight > weight){
35. //change curr weight to given weight from last one to the first one
36. curr.myWeight = weight;
37. curr.setWeight(weight);
38. while(curr != null) {
39. double maxWeight = -1;
40. for(Character chr : curr.children.keySet()){
41. if(curr.children.get(chr).mySubtreeMaxWeight > maxWeight){
42. maxWeight = curr.children.get(chr).mySubtreeMaxWeight;
43. }
44. }
45. curr.mySubtreeMaxWeight = maxWeight;
46. curr = curr.parent;
47. }
48. }
49. else {
50. //make sure to set to correct values.
51. curr.isWord = true;
52. curr.setWord(word);
53. curr.setWeight(weight);
54. curr.myWeight = weight;
55. curr.myWord = word;
56. }
57. }

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 3 | O(1) |
| 4 | O(1) |
| 6 | O(1) |
| 7 | O(1) |
| 10 | O(1) |
| 11 | O(W = word.length) |
| 13 | O(1) |
| 14 | O(1) |
| 19 | O(1) |
| 20 | O(1) |
| 21 | O(1) |
| 24 | O(1) |
| 29 | O(1) |
| 30 | O(1) |
| 31 | O(1) |
| 33 | O(1) |
| 35 | O(1) |
| 36 | O(1) |
| 37 | O(1) |
| 38 | O(1) |
| 39 | O(c = number of characters) |
| 40 | O(1) |
| 41 | O(1) |
| 44 | O(1) |
| 45 | O(1) |
| 50 | O(1) |
| 51 | O(1) |
| 52 | O(1) |
| 53 | O(1) |
| 54 | O(1) |

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

*topMatch*

* Pseudocode:

1. public String topMatch(String prefix) {
2. // TODO: Implement topMatch
3. if (prefix == null) {
4. throw new NullPointerException("prefix can not be null");
5. }
6. Node curr = myRoot;
7. // find prefix
8. for(int i = 0; i < prefix.length();i++) {
9. if(curr.children.containsKey(prefix.charAt(i))){
10. curr = curr.children.get(prefix.charAt(i));
11. }
12. else {
13. return "";// if no, return noting
14. }
15. }
16. if(curr.mySubtreeMaxWeight == curr.getWeight() && curr.isWord) {
17. return curr.getWord();
18. }
19. //if a given string does not match in pre-tire, create new branch and set weight
20. while(curr.mySubtreeMaxWeight != curr.myWeight && !curr.isWord) {
21. for(Character ch: curr.children.keySet()) {
22. if(curr.children.get(ch).mySubtreeMaxWeight == curr.mySubtreeMaxWeight) {
23. curr = curr.children.get(ch);
24. break;
25. }
26. }
27. }
28. return curr.myWord;
29. }

* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 3 | O(1) |
| 4 | O(1) |
| 6 | O(1) |
| 8 | O(P = prefix.length) |
| 9 | O(1) |
| 10 | O(1) |
| 13 | O(1) |
| 16 | O(1) |
| 17 | O(1) |
| 20 | O(1) |
| 21 | O(C = nember of character) |
| 22 | O(1) |
| 23 | O(1) |
| 24 | O(1) |
| 28 | O(1) |

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

*topMatches*

* Pseudocode:
  + 1. public Iterable<String> topMatches(String prefix, int k) {
    2. // TODO: Implement topKMatches
    3. if (prefix == null) {
    4. throw new NullPointerException("prefix can not be null");
    5. }
    6. Node curr = myRoot;
    7. for(int i = 0; i < prefix.length(); i++){
    8. if(curr.children.containsKey(prefix.charAt(i))) {
    9. curr = curr.children.get(prefix.charAt(i));
    10. }
    11. else {
    12. return new ArrayList<String>();
    13. }
    14. }
    15. //as descreption, we need build priorityQueue to hold the first Kth topmatches
    16. //most diffcult part
    17. PriorityQueue<Node> pQueue = new PriorityQueue<Node>(k, new Node.ReverseSubtreeMaxWeightComparator());
    18. PriorityQueue<Node> min = new PriorityQueue<Node>(k);
    19. pQueue.add(curr);
    20. while(!pQueue.isEmpty()) {
    21. curr = pQueue.remove();
    22. if(min.size() == k && min.peek().myWeight < curr.myWeight && curr.isWord) {
    23. //Retrieves and removes the head of this queue, or returns null if this queue is empty
    24. min.poll();
    25. min.add(curr);
    26. }
    27. else if (curr.isWord){
    28. min.add(curr);
    29. }
    30. for (Character chr : curr.children.keySet()){
    31. pQueue.add(curr.children.get(chr));
    32. }
    33. }
    34. ArrayList<String> aList = new ArrayList<String>();
    35. Node[] nodes = new Node[min.size()];
    36. for(int i = 0; i < nodes.length; i ++){
    37. nodes[i] = min.poll();
    38. }
    39. int lengthOfList = nodes.length - 1;
    40. for (int i = 0; i < nodes.length && i < k; i++){
    41. int index = lengthOfList - i;
    42. aList.add(nodes[index].getWord());
    43. }
    44. return aList;
    45. }
* Complexity analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 3 | O(1) |
| 4 | O(1) |
| 6 | O(1) |
| 6 | O(P = prefix.length) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| 12 | O(1) |
| 17 | O(1) |
| 18 | O(1) |
| 19 | O(1) |
| 20 | O(1) |
| 21 | O(1) |
| 22 | O(1) |
| 24 | O(1) |
| 25 | O(1) |
| 27 | O(1) |
| 28 | O(1) |
| 30 | O(c = number of characters) |
| 31 | O(1) |
| 34 | O(1) |
| 35 | O(1) |
| 36 | O(N = node.length) |
| 37 | O(1) |
| 39 | O(1) |
| 40 | O(N = node.length) |
| 41 | O(1) |
| 42 | O(1) |
| 44 | O(1) |

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

2.**Testing**: Complete your test cases to test the *TrieAutoComplete* functions based upon the criteria mentioned below.

**Test of correctness:**

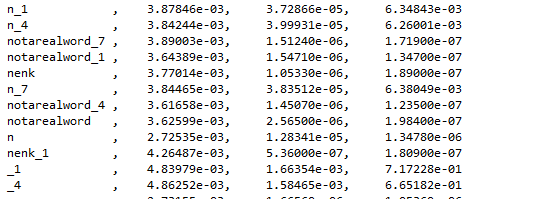
Assuming the trie already contains the terms {”ape, 6”, ”app, 4”, ”ban, 2”, ”bat, 3”, ”bee, 5”, ”car, 7”, ”cat, 1”}, you would expect results based on the following table:

|  |  |  |
| --- | --- | --- |
| 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 | {} |

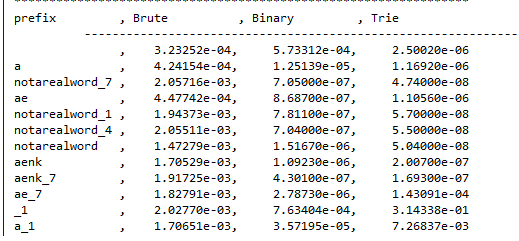
3.**Analysis**: Answer the following questions. Use data wherever possible to justify your answers, and keep explanations brief but accurate:

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?
   1. ADD: O(W)
   2. TopMatches: O(P)
   3. TopMatch: O(P)
2. 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. For runtime of TireAutocomplete, this is will increase by a very tiny amount, for runtime of ButeAutocomplete, this is will exponentially increase, since it will go through all data.
3. 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



Fourletterwordshalf.txt



As we shown in the first question, the complexity of TopMatches will not change and runtime is change a little, however, Burte will change much faster in BIG oh, since this algorithm will go entire data.

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*.
   1. As we see above, Tire is smaller than Brete. So the Tire algorithm is faster