CSE 4304-Data Structures Lab. Winter 2022-23
Batch: CSE 21
Lab: 10
Date: October 23, 2023.
Target Group: All
Topic: AVL Trees, Trie

<u>Instructions</u>:

- You must submit the solutions in the Google Classroom despite finishing the lab tasks within lab time. If I forget to upload the tasks there, CR should contact me. The deadline will always be 11:59 PM on the day the lab has occurred.
- Task naming format: fullID_T01L07_2A.c/cpp.
- If you find any issues in the problem description/test cases, comment in the Google Classroom.
- If you find any tricky test case that I didn’t include, but others might forget to handle, please comment! I’ll be happy to add.
- Use appropriate comments in your code. This will help you to easily recall the solution in the future.
- Obtained marks will vary based on the efficiency of the solution.
- Do not use <bits/stdc++.h> library.
- Modified sections will be marked with BLUE color.

Group	**Tasks**
2B	1 2 3 4
2A	1 2 3 5
1B	1 2 4 6
1A	1 2 4 6
Assignment	**The tasks not covered in your lab**

<u>Task-1:</u> **Calculating the Balance Factor of different nodes**

A series of values are being inserted in a BST. Your task is to show the balance factor of every node after each insertion.

balance_factor= height_of_LeftSubtree - height_of_RightSubtree

The following requirements must be addressed:

- Continue taking input until -1.
- After each insertion, print the nodes of the tree in an in-order fashion. Show the balance_factor beside each node within a bracket.
- Each node has the following attributes: data, left_pointer, right_pointer, parent_pointer, and height. (store balance_factor if needed, but optional)
- Your code should have the following functions:
    - void insertion (key): iteratively inserts a key into the BST.
    - void Update_height(node): update the height of a node after each insertion. Note that only the ancestors are affected after the insertion of a new key.
    - int height(node): returns the height of a node
    - int balance_factor(node): returns balance factor of a node

Sample Input	**Sample Output**

12
8
5
11
20
4
7
17
18
-1
	12(0)
	8(0) 12(1)
	5(0) 8(1) 12(2)
	5(0) 8(0) 11(0) 12(2)
	5(0) 8(0) 11(0) 12(1) 20(0)
	4(0) 5(1) 8(1) 11(0) 12(2) 20(0)
	4(0) 5(0) 7(0) 8(1) 11(0) 12(2) 20(0)
	4(0) 5(0) 7(0) 8(1) 11(0) 12(1) 17(0) 20(1)
	4(0) 5(0) 7(0) 8(1) 11(0) 12(0) 17(-1) 18(0) 20(2)

<u>Task-2</u>: Balancing a BST

Utilize the functions implemented in Task-1 to provide a complete solution for maintaining a ‘Balanced BST’. The program should continue inserting values until it gets -1. For each insertion, it checks whether the newly inserted node has imbalanced any node or not. If any imbalanced node is found, ‘rotation’ is used to fix the issue.

Your program must include the following functions:

- void insertion (key): iteratively inserts a key into the BST.
- void Update_height(node): update the height of a node after each insertion. Note that only the ancestors are affected after the insertion of a new key.
- int height(node): returns the height of a node
- int balance_factor(node): returns the balance factor of a node
- left_rotate(node)
- right_rotate(node)
- check_balance(node): check whether a node is imbalanced and call relevant rotations if needed.
- print_avl(root): print the tree using inorder traversal. The balance factor is printed beside each node.

Sample Input	**Sample Output**

12	12(0)
	Balanced
	Root=12

9	**9**(0) 12(1) Balanced Root=12
5	5(0) **9**(1) 12(2) Imbalance at node: 12 LL case **right**_rotate(12) Status: 5(0) **9**(0) 12(0) Root=**9**

11	5(0) **9**(-1) 11(0) 12(1)
	Balanced
	Root=**9**

20	5(0) **9**(-1) 11(0) 12(0) 20(0)
	Balanced
	Root=**9**

15	5(0) **9**(-2) 11(0) 12(-11) 15(0) 20(1)
	Imbalance at node: **9**
	RR case
	Left_rotate(**9**)
	Status: 5(0) **9**(0) 11(0) 12(0) 15(0) 20(1)
	Root=12

7	5(-1) 7(0) **9**(1) 11(0) 12(1) 15(0) 20(1)
	Balanced
	Root=12

3	3(0) 5(0) 7(0) **9**(1) 11(0) 12(1) 15(0) 20(1)
	Balanced
	Root=12

6	3(0) 5(-1) 6(0) 7(1) **9**(2) 11(0) 12(1) 15(0) 20(1)
	Imbalance at node: 9
	LR Case
	Left_rotate(5), right_rotate(9)
	3(0) 5(0) 6(0) 7(0) **9**(-1) 11(0) 12(1) 15(0) 20(1)
	Root=12

27	3(0) 5(0) 6(0) 7(0) **9**(-1) 11(0) 12(1) 15(0) 277(0) 20(0)
	Balanced
	Root=12

-1	Status: 3(0) 5(0) 6(0) 7(0) **9**(-1) 11(0) 12(1) 15(0) 20(0) 27(0)

Note:
- **Do not** use any recursive implementation

The status of the tree is finally supposed to be like this:

```
                                 12(1)
                              /       \
                             7(0)       20(0)
                            /  \      /    \
                          5(0)   9(0) 15(0)  27(0)
                         /  \      \
                      3(0)   6(0)  11(0)
```

```
12(0)  ->    12(1)  ->    12(2) -> LL Case                       9(0)
             /            /         Right_rotate(12)            /    \
          9(0)          9(1)                                 5(0)    12(0)
                       /
                      5(0)
-----------------------------------------------------------------------------
      9(-1)               9(-1)                     9(-2)
    /    \                /    \                     /    \
   5(0)  12(1)  ->    5(0)  12(0)        ->      5(0)   12(-1)
        /                  /   \                      /   \
      11(0)               11(0) 20(0)             11(0)   20(1)
                                                          /
                                                        15(0)
-----------------------------------------------------------------------------
->imbalance at(9)                12(0)                   12(1)
  RR Case                     /      \                  /      \
  Left_rotate(9)           9(0)      20(1)  ->       9(1)      20(1)
                           /  \       /              /  \       /
                        5(0)  11(0)  15(0)        5(-1)  11(0) 15(0)
                                                     \
                                                     7(0)
-----------------------------------------------------------------------------
             12(1)                            12(1)
          /      \                         /      \
        9(1)      20(1)                 9(2)      20(1)
      /  \       /                      /  \      /
   5(0)  11(0)  15(0)     ->         5(-1)  11(0)  15(0)
  /    \                            /    \
3(0)    7(0)                      3(0)    7(1)
                                          /
                                       6(0)
-----------------------------------------------------------------------------
Imbalance at node(9), LR Case                 12(1)
Left_rotate(5)                             /       \
                                        9(2)       20(1)
                                        /   \     /
                                      7(2)  11(0) 15(0)
                                  /
                                    5(0)
                                   /  \
                                 3(0)  6(0)
-----------------------------------------------------------------------------
Right_rotate(9)                  12(1)
                              /      \
                             7(0)      20(1)
                            /  \      /
                          5(0)   9(-1) 15(0)
                        /  \       \
                      3(0)   6(0)  11(0)
-----------------------------------------------------------------------------

                                 12(1)
                              /       \
                             7(0)       20(0)
                            /  \      /    \
                          5(0)   9(-1) 15(0) 7(0)
                        /  \       \
                      3(0)   6(0)  11(0)

-----------------------------------------------------------------------------
```

<u>Task 3</u>: Basic operations of Trie data structure

Implement the basic operations of the 'Trie' data structure by implementing the following functions:

- void **Insert**(): Inserts a string in a trie
- boolean **Search**(): Returns if the query string is a valid word.
- void **Display**(): Shows all the words that are stored in the Trie in lexicographically sorted order.

The first line of input contains space-separated words that need to be inserted in the Trie. Once the words are inserted, display all of them.
The following line contains another collection of query words. Print **T/F** based on their presence/absence.

Sample Input	**Sample Output**
toy algo algorithm to tom also algea tommy toyota toy toyo al also algorithm algorithmic	algea algo algorithm also to tom tommy toy toyota T F F T T F

<u>Task 4</u>: Find the number of words starting with a certain prefix

Suppose a set of words is stored in a Dictionary. Given a *prefix*, your task is to find out how many words start with it.

The first input line will contain *N* and *Q*, where *N* represents the number of words in the dictionary, and Q is the number of queries. Print the number of words starting with each corresponding prefix.

Sample Input	**Sample Output**
10 10	
Beauty	
Beast	
Beautiful	
Amazing	
Amsterdam	
Beautify	
Banana	
Xray	
Glorifying	
A	
Am	2
AM	2
Beauty	2
Beaut	1
Beast	3
Ing	1
AMS	0
Be	1
B	4
	5

Note: Convert every string/prefix in lowercase before storing/ searching.

Task 5: Search Suggestions

You are given a set of ‘products’ and a string ‘searchWord’. Design a solution that suggests **at most three** products after each character of searchWord is typed. Suggested products should have a common prefix with searchWord. If there are more than three products with a common prefix, follow the lexicographical order.

Input (products)	**Output**	**Explanation (searchWord)**
mobile mouse moneypot monitor mousepad mouse	mobile moneypot monitor mobile moneypot monitor mouse mousepad mouse mousepad mouse mousepad	**‘m’** **‘mo’** **‘mou’** (only 2 matches) **‘mous’** (only 2 matches) **‘mouse’** (only 2 matches)
havana havana	havana havana havana havana havana	**‘h’** **‘ha’** **‘hav’** **‘hava’** **‘havana’**
juice jeerapani icecream jelly jam jackfruit jalapeno jeans	jackfruit jalapeno jam Jelly jeerapani Null Null Null	**‘J’**: 6 words matched. Printed only the first 3 in lexicographical order. **‘Je’**: 2 matches No match found for **‘jea’**, **‘jean’**, **‘jeans’**. Hence null.

Task 6: Chaotic Reading

As you will learn in the Human-Computer Interaction (HCI) course, the human information processing system possesses an impressive ability for text recognition, called Typoglycaemia, which makes us capable of deciphering sentences presented in a severely distorted manner, as illustrated below:

The ACM Itrenntaoial Clloegaite Porgarmmnig Cnotset (IPCC) porvdies clolgee stuetnds wtih oppriuntetiis to itnrecat wtih sutednts form ohetr uinevsrtieis.

The comprehension of these sentences typically adheres to the following principle: The ***initial and last letters of each word remain unchanged,*** while the ***internal letters may be shuffled around freely***. Your task is simple – Given a sentence and a lexicon of words, how many distinct sentences could be constructed and potentially mapped to the same encoding?

Input

The first line of the input is an integer n, which denotes the number of words in the given lexicon. Each of the following n lines consists of a word present in the lexicon. After this, there is a line containing the number m, which is the number of sentences that should be tested with the preceding lexicon. Each of the following m lines constitute those sentences.

Output

For each case, print the case number first. Then for each sentence, output **the number of sentences** that can be formed in a separate line.

Sample Test Case(s)

Input	Output
8 baggers beggars in the blowed bowled barn bran 2 beggars bowled in the barn blowed beggars the bran baggers in the barn	8 32
7 repairer rearpier rarepier drawer reward coronavirus carnivorous 1 rearpier reward coronavirus	6

Note: Sorting a certain portion of the word can be helpful.

<u>**Explanation**</u>:

Case 1: ‘beggars bowled in the barn’

- beggars : can be compared with 2 words (baggers, beggars)
- bowled: blowed, bowled
- in : in
- the: the
- barn: barn, bran
- So total interpretable sentences possible = 2x2x1x1x2 = 8
- The sentences can be interpreted by the reader in the following ways. Note that, the position of a word is not permuted. Rather, only the alternative words are considered.
    - beggars bowled in the barn
    - beggars bowled in the bran
    - beggars blowed in the barn
    - beggars blowed in the bran
    - baggers bowled in the barn
    - baggers bowled in the bran
    - baggers blowed in the barn
    - baggers blowed in the bran
- Similarly, for the sentence **‘blowed beggars the bran baggers in the barn’** we find 2x2x1x2x2x1x1x2=32 possible interpretations.

Case 2:

- ‘rearpier reward coronavirus’
- rearpier: can be interpreted in 3 ways: repairer, rearpier, rarepier
- reward: only 1 way (reward). Note that, ‘drawer’ also consists of the same letters in different permutations. However, we don’t consider it as the first and last letters are not the same.
- coronavirus: 2 ways (coronavirus, carnivorous)
- Hence the sentence can be interpreted in 3x1x2=6 ways
    - rearpier reward coronavirus
    - rearpier reward carnivorous
    - repairer reward coronavirus
    - repairer reward carnivorous
    - rarepier reward coronavirus
    - rarepier reward carnivorous