CST 370 – Homework 3

Due: **11:59pm on March 17, 2019**

Final Submission Deadline: **2:00pm on March 26, 2019**

Changelog:

- 3/12/19: Extended initial due date. Replaced exclaimation marks.
- 3/11/19: Removed exclaimation mark from output of Khan's
- 3/07/19: Added correct hacker rank links

About

For this shorter homework assignment, you have one graph problem and one problem related to self-balancing trees.

Submissions

https://www.hackerrank.com/contests/cst370-s19-hw3/challenges

The problems for this homework will be hosted on HackerRank where automated tests will run. You may submit code on HackerRank as many times as you'd like. Once you are comfortable with your code, you can copy the submission link for that problem and include it in the ilearn assignment for me to grade. **I will only grade hackerrank links submitted on iLearn.** iLearn has videos describing how to get a submission link.

After the due date, you may continue working on problems on HackerRank. If you submitted something at the initial deadline, you may submit HackerRank links again on the iLearn assignment for resubmissions. Resubmissions will be assessed a 10% deduction.

Scoring

Your score on HackerRank or Kattis will give you *an approximation* of what your grade will be; however, I will read your code and determine your final grade myself, as detailed on the syllabus.

Problems

1	TreeMaps	25 points
2	Cycle Detection	15 points

This page is intentionally left blank.

1. TreeMaps (25 points)

https://www.hackerrank.com/contests/cst370-s19-hw3/challenges/avltreemap

For this problem, you will implement an AVL tree using nodes. Your AVL tree will hold two values at each node: a key and a value. This data structure is called a tree map and is used like a hash table to quickly store and retrieve values based on a key!

For example, in C++, the node and class definitions would look like:

```
struct Node {
    int key;
    string value;
    Node *left;
    Node *right;
    int height;
};
class AVLTreeMap {
    Node *root;
};
```

You will need to implement the following operations:

`put(key, value)` which inserts a key/value pair into the tree map.

`get(key)` which returns `nullptr` or the value associated with that key.

`remove(key)` which removes an item from the map.

`levelorder` which prints out a levelorder traversal of the tree map.

For example, in C++, the function headers would be the following:

```
class AVLTreeMap {
    Node *root;

  private:
    Node *rotateLeft(Node *curr) {
            // ...
    }

    Node *rotateRight(Node *curr) {
            // ...
    }

    int getHeight(Node *curr) {
            // ...
    }

    int getBalance(Node *curr) {
            // ...
    }

  public:
    AVLTreeMap() {
        // ...
    }
    string get(int key) {
        // ...
    }
    void put(int key, string value) {
        // ...
    }
    void remove(int key) {
            // ...
    }
    void levelOrder() {
        // ...
    }
};
```

Input

- The first line of the input will be an integer n indicating how many commands follow it.
- The next n lines will consist of one of 4 commands ("put", "get", "remove", and "levelorder").
- The command "put" will be followed by a space, then an integer representing the key, followed by a space, then a string representing the value.
- The command "get" will be followed by a space then an integer representing the key whose value we want to retrieve.
- The command "remove" will be followed by a space then an integer representing the key whose node we want to remove. NOTE: the key may not always be in the tree map. If it's not there just do nothing.

At the end of the input there will be a blank line. Your program should initialize an empty AVL tree and perform the commands given by the input, in sequence, on it.

Constraints

You can assume there will be no invalid input. You can also assume the keys will consist only of positive integers with no duplicates.

Output

- For the "get" command, print the value corresponding to the key we want to retrieve or "not found" if the key isn't in the tree.
- For the "levelorder" command, print a line containing thespace-separated traversal of the map. For each node, print the key, followed by a colon(":"), followed by the value with the balance of each node in parentheses after the value (the line will end in a space).

See the sample output section and hackerrank for concrete examples

Sample Input 1	Sample Output 1
17 put 5 a levelorder put 3 b levelorder put 2 c levelorder put 9 d levelorder put 1 e levelorder put 7 f levelorder put 0 g levelorder get 5 get 6 get 7	5:a(0) 5:a(1) 3:b(0) 3:b(0) 2:c(0) 5:a(0) 3:b(-1) 2:c(0) 5:a(-1) 9:d(0) 3:b(0) 2:c(1) 5:a(-1) 1:e(0) 9:d(0) 3:b(0) 2:c(1) 7:f(0) 1:e(0) 5:a(0) 9:d(0) 3:b(0) 1:e(0) 7:f(0) 0:g(0) 2:c(0) 5:a(0) 9:d(0) a Not Found f

2. Cycle Detection (15 points)

https://www.hackerrank.com/contests/cst370-s19-hw3/challenges/cycledetect

For this problem, you will practice translating the description of an algorithm into code, starting with an algorithm we've briefly mentioned before: Cycle Detection with Khans!

Given an adjacency list representation of a graph, you can use Khan's algorithm to detect if there are any cycles in the graph. This is the algorithm:

- First we compute the in-degrees of each node and store it in a map where the keys are GraphNode pointers and the values are integers representing the in-degree for that graph node.
- Next we begin khan's algorithm by creating a queue to hold the GraphNodes we can visit. These will be GraphNodes with in-degree 0.
- While the queue is not empty, take the GraphNode at the front of the queue and put it into a visited set. For each of the node's neighbors, decrement that neighbor's in-degree by 1. If this would change the neighbor's in-degree to 0, add that neighbor into the queue.
- Continue until the queue is empty. Then compare the size of the visited set with the size of the graph. If the two are not equal, then there is a cycle. Otherwise there is no cycle.

Input

The input describes a directed graph where an edge from a to b means there is a directed edge connecting a to b.

- The first line contains an integer, n, the number of nodes.
- The next n lines each contain a string. This makes up an array of nodes.
- The next line contains two space-separated integers: n e
- The next e lines each contain two space-separated integers a and b describing an edge from the vertex indexed at a to the vertex indexed at b.

Constraints

In this problem, the graph will always be directed and you can assume the data in each node is unique.

Output

- Print "Cycle!" if there is a cycle in the given graph
- Print "No Cycle!" if there is no cycle in the given graph

Sample Input 1

```
3
Rock
Paper
Scissors
3 3
0 1
1 2
2 0
```

Sample Output 1

```
Cycle!
```

Sample Input 2

```
11
Mario
Wario City
Toad Harbor
Start City
Delfino Square
Cap Kingdom
Goomba Village
Bowser's Castle
Rose Town
Metro Kingdom
Peach
11 11
0 1
0 2
1 3
1 4
5 4
2 6
6 8
6 7
7 9
9 10
8 10
```

Sample Output 2

```
No Cycle!
```