CSSE 230 – Data Structures and Algorithms

Exam 2 Winter, 2013-14 Name: ___SOLUTION____ Section: 01 02 03

You may use a calculator, a writing implement, and 1 double-sided 8.5" x 11" note page. Blank paper is also available to use as scratch paper. You must turn in this written part before you use your computer, textbook, or any other resources.

You may not communicate with any person other than your instructor about this exam until allowed by your instructor. You may not use any electronic devices such as calculators, phones, or any device with headphones or earbuds.

Part 1 scores (for instructor use):

Problem	Possible	Score
1	16	
2	10	
3	4	
4	18	
Total	**48**	

Computer part = 72 points

Total	**120**	

Part 2 (computer) rules

You may use any printed or written materials that you brought with you, code that you wrote yourself or with a partner before the exam, anything on the CSSE230 Moodle and web sites, and the Java API documentation (on your laptop or at http://download.oracle.com/javase/7/docs/api/).

You may not otherwise search the internet or communicate with any person other than your instructor or a student assistant. You may not communicate with anyone else about this exam until allowed by your instructor.

You will sign a statement to certify that you neither received help from others nor gave it on either part of the exam.

You must actually get this problem working on your computer. All or most of the credit for this problem will be for code that actually works.

1. (16 points) T/F/IDK. Below you will find several statements. A statement is true (T) if it is always true. It is false (F) if there is at least one counterexample (sometimes false). You may also choose IDK to indicate that you do not know the answer. **Point values:** Correct answer: 2, incorrect answer: -1, IDK: 1, blank: 0. **Circle at most one answer for each part.** If you change your mind, make sure it is very clear which one you marked.

   a. T F IDK Any non-empty binary tree contains at least 1 node such that the sum of (the height of that node in the tree) and (the depth of the same node in the same tree) always equals the height of the tree.

   b. T F IDK In a hash table, the purpose of quadratic probing is to solve the problem of clustering that occurs with linear probing.

   c. T F IDK Hash tables work most time-efficiently when the load factor is high.

   d. T F IDK The worst-case time for insertion into a hash table with N elements is O(1).

   e. T F IDK After a deletion from an AVL tree, sometimes after we do a single or double rotation, additional rotations are still needed higher up in the tree in order to restore the height-balanced property.

   f. T F IDK A left rotation at node N in an AVL tree may cause the rank of N to change.

   g. T F IDK A single right rotation at node N in an AVL tree may cause the rank of N to change.

   h. T F IDK If a recursive algorithm on a binary tree of size N always recurses on both children and its leaves are base cases, the runtime must be $\Omega(N)$.

2. (10 points) A binary tree with height 4…

   (a) can have at most how many nodes? _____**31**

   (b) can have at most how many leaf nodes? _____**16**

   (c) must have at least how many nodes? _____**5**

   (d) must have at least how many leaf nodes? _____**1**

   (e) must have at least how many nodes if it is also height-balanced? _____**12**

3. (4 points) In the EditorTrees project, a Node object (as we gave it to you) has only 5 fields:

```
char element;
Node left;
Node right;
int rank;
Code balance;
```

Note that *size* of p (the number of nodes in the subtree rooted at p) is not stored. Write a method that, given a reference to node p, calculates the size of p in an EditorTree **in O(log N) time**. You will earn no credit if the runtime is not O(log n).

```
// p might be null or a NULL_NODE.
public static int size(Node p){
        if (p == NULL_NODE)
            return 0;

        return 1 + p.rank + size(p.right);
```

Don't use rank: -4

Forget base case: -2

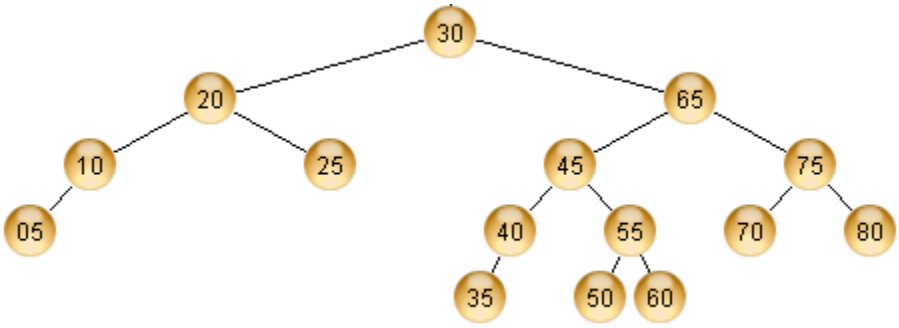
Forget +1 for root: -2

Somehow try to use rank instead of size() for right: -3

Use this.rank instead of p.rank, or same for right: -1

Try to define and use an instance variable instead of a local variable: -3

4. (18 points) Consider the following AVL tree. **Each insertion or deletion starts with this original tree** (that is, insertions/deletions are **not** cumulative). Use the standard AVL algorithms for determining where rotations are needed and which type of rotation fixes the imbalance.



a. Fill in the following table for insertions. For each number given, suppose we want to insert that number into the original tree. (**Remember, start with the original tree for each insertion**.) Fill in the entries in that number's row. If no rotation is needed, write ***NONE*** in the "where rotation needed" column, and leave the rest of that row blank.

Number to insert into the tree	New node's parent after insertion, before rotation	Node where rotation happens (or NONE)	Single or double rotation?	Left or right?	New root of rotated subtree	Left child of that new root	Right child of that new root
2	**5**	**10**	**Single**	**Right**	**5**	**2**	**10**
7	**5**	**10**	**Double**	**Right**	**7**	**5**	**10**
47	**50**	**65**	**Double**	**Right**	**55**	**45**	**65**

b. After inserting node 47, how many balance codes in the tree are different from their original value before the insertion + rotation? ___**2**____ (50: SAME changes to LEFT, 65 : LEFT changes to RIGHT)

c. Fill the in the following lines for deletion (same instructions as above).

Number to delete from the tree	Node's parent after insertion, before rotation	Node where rotation happens (or NONE)	Left or right rotation?	Single or double?	New root of rotated subtree	Left child of that new root	Right child of that new root
5	N/A	**30**	**Double**	**Left**	**45**	**30**	**65**
70	N/A	**NONE**					

Grading: 33 table entries @ ½ point each + 1.5 points for part b = 18 points.

Our code for the Computer part:

Mirror:

```
    public static BinaryNode mirror(BinaryNode t) {
        if (t == null) { return null; }

        BinaryNode newLeft = mirror(t.right);
        BinaryNode newRight = mirror(t.left);
        return new BinaryNode(t.element, newLeft, newRight);

        // Super-concise:
//      return  t == null ?
//              null :
//              new BinaryNode(t.element, mirror(t.right), mirror(t.left));
```