CSSE 230 – Data Structures and Algorithms

Exam 2 Winter, 2013-14 14 (modified for practice) Name: _____ Section: 01 02 03

You may use a calculator, a writing implement, and a 1-sided 8.5" x 11" note page. 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 other electronic devices such as phones or any device with headphones or earbuds.

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 oracle.com).

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 may **not** use other electronic devices such as phones or any device with headphones or earbuds.

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.

> **Example**: 4 correct, 2 incorrect, 3 IDK, 1 blank yields 8 – 2 + 3 + 0 = 9

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 whose height is 4…

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

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

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

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

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

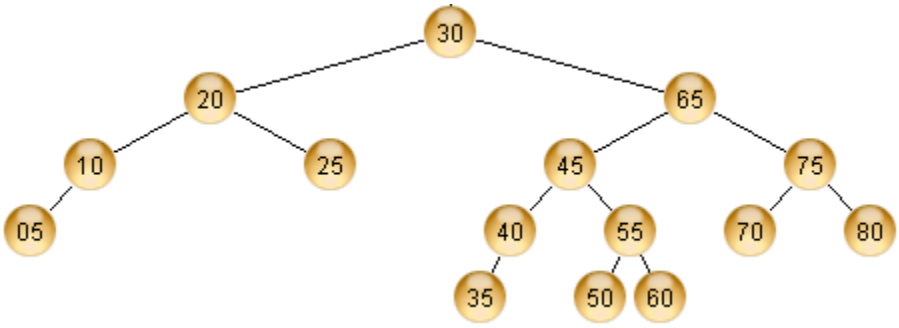
3. (4 points) In the EditorTrees project, a Node object (as we gave it to you) has only 5 fields:

```
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){
```

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							
7							
47							

b. After inserting node 47, how many balance codes in the tree are different from their original value before the insertion + rotation? _____

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						
70	N/A						

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

Instructions.

- 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 anyone other than your instructor or a student assistant. You may not communicate with anyone else about this exam until allowed by your instructor.
- You may not use any electronic devices (calculators, phones, Google Glasses etc.). No headphones or earbuds.
- All code you write should be **correct, efficient, and use good style.** However, no documentation is required, because of time constraints.
- Check out the Exam2 project from your csse230 SVN repository**.**
- **If you want to use the debugger, you may remove or increase the "timeout" time in the unit tests so that they won't timeout in the middle of debugging.**

Practice note: The first two problems are ones from other old exams that only let you recurse through the tree once to earn the efficiency points. I include them with a BIG hint on process for #1 and #2. The second two are the ones that were actually from the current exam.

1. Write **double averageValue()** that finds the average numeric value of every node in this tree. For example, a tree with root = 10 and children = 5 and 20 would return (10+5+20)/3 = 11.666666. Note: we use a BST of **Integer** rather than a BST of generic type T here for simplicity.

   You earn the efficiency points by making sure that your method runs in O(n) time, where n is the size of the tree, by using a single recursive traversal that does not (i) visit any node more than once, or (ii) use any extra data structure whose size depends on the size of the tree (other than the stack that Java uses to keep track of recursive calls, of course). You may not include any other fields like balance codes in the BinaryTree or BinaryNode classes. You also may not modify the given insert() method in any way. However, you may add helper methods with parameters and return values of any type, including a class that you define.

[[[Study hint for problems that only let you recurse through the tree once: you usually want to return a Wrapper object.

On this problem, focusing on correctness first will probably lead you to a solution where you write a recursive method to find the sum of the values and second method (like size()) to find the count of the nodes. You'd call them each and then divide the results. But that would visit each node more than once. To overcome this, think like this.

(a) Since we only get **one** recursive helper call, we ask, "what is ALL the info a node needs from each of its children to figure out this problem?". Here, it is both the sum and node count of each child's subtrees. So we make a Wrapper class and put these things in it as fields.
(b) Then we ask "What Wrapper object should be returned when we hit the base case?" If it's possible to let the NULL_NODE be the base case, that's usually easiest. For this, sum=0 and count=0 both work fine.
(c) Then, once we have called the recursive method on left and right children and have a couple of wrapper objects, how do we combine them to make a wrapper object to return to our parent? Here, a node adds the left and right sums plus its own data to get the whole sum of its subtree, and the left and right counts + 1 for itself to get the count of nodes in its subtree.
(d) Finally, the root node will return a wrapper object to the tree method that called the recursive helper. The tree method uses what it needs to answer the question. Sometimes it only needs one of the fields.]]]

2. In the `heightbalance` package, write **boolean isHeightBalanced()** that determines if the given tree is *height-balanced*. (You can refer to Day 12 slides or section 19.4 of the text if you need a refresher on the definition.) I use character as the data type here for convenience in writing tests, but your method shouldn't need to look at the node data at all.

   The efficiency requirements are the same as question #1, and the study hint there applies to this one as well!

3. For simplicity, we simply represent a binary tree as a reference to the root node, so we do not need a separate BinaryTree class. Write the details of a BinaryNode method named *mirror* that creates a new tree that is a mirror image of the tree it is given. It cannot mutate the original tree. Include any helper methods that you may need.

```
/**
 * Creates a mirror reflection (left to right) of
 * the entire tree rooted at node t.
 *
 * @param t The root of a tree.
 * @return The root of a new tree that is a mirror
 * image of the tree  whose root is node t.
 */
public static BinaryNode mirror(BinaryNode t) {
```