600.363 Introduction to Algorithms / 600.463 Algorithms I **Fall 2014**
Homework #4 **Due:** October 7, 2014, 1:30pm

Please start each problem on a new page, and include your name on each problem. You can submit on blackboard, under student assessment.

Remember: you may work in groups of up to three people, but must write up your solution entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. You may use the internet to look up formulas, definitions, etc., but may not simply look up the answers online.

Please include proofs with all of your answers, unless stated otherwise.

1 More Search Trees (33 points)

We saw in class how to use binary search trees as dictionaries, and in particular how to use them to do *insert* and *lookup* operations. Suppose we want to modify the basic binary search tree to also let us perform the following operation (assuming that all keys are distinct): Given a key x, do a version of $lookup(x)$ that tells us how many keys are less than x (we call this the rank of x)

Note that if our data was fixed, this would be easy. We could simply use a sorted array A. Then to find the rank of x we simply do a binary search to find x and then return its position minus 1.

But if we want to handle inserts as well, then one way to do this is through search trees. The goal of this problem is to modify normal binary search trees (nothing fancy like B-trees or treaps) so that the above operation can be done in $O(depth)$ time, and inserts can also be done in $O(depth)$ time. In particular, you should do the following:

(a) Describe an extra piece of information that you will store at each node of the tree.

(b) Describe how to use this extra information to do the above rank operation in $O(depth)$ time.

(c) Describe how to maintain this information in $O(depth)$ time when a new node is inserted (note that there are no rotations on an insert – it's just the regular binary search tree insert, but you need to update information appropriately).

For example, a bad way to do this would be for every node to store the rank of its key. This information would let us do the rank operation quickly, but maintaining it on an insert might require updating all of the other nodes (which might be much larger than the depth).

2 Hashing (33 points)

We saw in class that universal hashing lets use give guarantees that hold for arbitrary (i.e. worst case) sets S, in expectation over our random choice of hash function. Let's work out some of those guarantees.

(a) Describe an explicit universal hash family from $U = \{0, 1, 2, 3, 4, 5, 6, 7\}$ to $\{0, 1\}$. Hint: you can do this with four functions.

(b) Let H be a universal hash family from U to a table of size m. Let $S \subseteq U$ be a set of m elements which we want to hash. Prove that if we choose h from H uniformly at random, the expected number of pairs $x, y \in S$ that collide is at most $\frac{m-1}{2}$.

(c) Prove that with probability at least $3/4$, no bin gets more than $1 + 2\sqrt{m}$ elements.

Hint 1: use part (b).

Hint 2: You should use "Markov's Inequality". This is actually a pretty obvious fact: if X is a nonnegative random variable with expectation $\mathbf{E}[X]$, then $\mathbf{Pr}[X > k\mathbf{E}[X]] < 1/k$ for any $k > 0$. For example, the probability that X is more than 100 times its expectation is less than $1/100$. You should be able to prove to yourself that this is true from the definition of expectation.

3 Balanced Binary Trees (33 points)

We saw in class that B-trees are exactly balanced (all leaves are the same depth from the root), but we pay for this by making the tree non-binary. We will now figure out a data structure which maintains an almost perfectly balanced binary search tree.

Given a node x in a binary tree, define $height(x)$ to be the maximum over all of the descendants y of x of the distance from x to y. So a leaf (with no children) has height 0, a node whose children are leaves (or whose single child is a leaf) has height 1, etc. For any node x, define its *balance factor* to be the height of its left child minus the height of its right child. We say that a binary tree is *height-balanced* if for every node x, the balance factor of x is either $1, 0$, or -1. (i.e. the heights of its children differ by at most 1). If a node has only one child, let's pretend that its nonexistent child has rank -1, and thus the real child must be a leaf.

(a) Prove that in any height-balanced tree with n nodes, the height of the root is $O(\log n)$. Hint: prove that a height-balanced tree of height k has at least F_k nodes, where F_k is the kth Fibonacci number ($F_0 = 1, F_1 = 2, F_2 = 3, F_3 = 5, F_4 = 8, F_5 = 13, \dots$).

Now let's see how to actually maintain such a tree. We first modify the basic binary search tree construction to force every node to record its height, i.e. for each node x there is a field $x.height$ which will always be equal to $height(x)$. If we do a normal binary search tree insert, we can update all of these values efficiently (since we will only need to change them for nodes on the path down to the new nodes), but we might lose the height-balanced property. We cannot lose it by much, though: clearly the height of any node only increased by at most 1, so for any node the balance factor is at least -2 and at most 2.

Suppose that the balance factor of x is 2 (so the height of $x.left$ is two more than the height of $x.right$). Further suppose that the trees rooted at $x.left$ and $x.right$ are height-balanced, so all nodes in them have balance factor between -1 and 1.

(b) Suppose even further that the balance factor of $x.left$ is either 1 or 0 (so it doesn't "lean right"). Show how to make the entire the subtree rooted at x height-balanced, i.e. how to change the tree so all nodes that were originally in the subtree rooted at x (including x itself) have balance factor between -1 and 1. Hint: think about rotations, and note that x does not have to stay as the root of the subtree.

(c) Now let's handle the case of $x.left$ having balance factor of -1 (so it "leans right", i.e. the height of $x.left.right$ is one more than the height of $x.left.left$. Show how to make the subtree rooted at x height-balanced. Hint: use rotations and part (b).

Together, parts (b) and (c) imply that if the balance factor of x is 2 then we can fix its subtree to be height-balanced. The case of x having height-factor -2 is analogous, so we will just assume that we can also fix the subtree rooted at x when x has balance factor -2. This gives a very simple insertion algorithm: insert a new element as in a normal binary search tree, then fix subtrees that need to be fixed (from the bottom up).

(d) Prove that the total running time of this new insertion algorithm is $O(\log n)$.