Exam 3

CS 3510A, Spring 2014

These solutions are being provided **for your personal use only**. They are not to be shared with, or used by, anyone outside this class (Spring 2014 section of Georgia Tech CS 3510A). Deviating from this policy will be considered a violation of the GT Honor Code. **Instructions:**

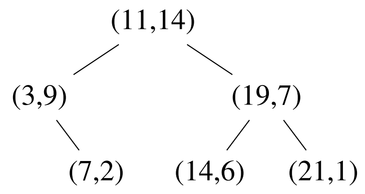
- **Do not open this exam packet until you are instructed to.**
- **Write your name** on the line below.
- You will have **50 minutes** (from 11:05 to 11:55) to earn **50 points**. Pace yourself!
- You may use one double-sided 8.5-by-11" study sheet. **You may not use any other resources, including electronic devices.**
- Do your best to fit your final answers into the space provided below each question. You may use the backs of the exam pages, and the final page of the exam, as scrap paper.
- **You may refer to any facts from lectures or the textbook without re-deriving them.**
- Your work will be graded on correctness and clarity. Please write legibly!

Question	Points	Score
1	15	
2	10	
3	25	
Total:	50	

Your name: ______________________________

1. (15 points)

   (a) Consider the following binary tree on (value, priority) pairs.



   Is this tree a legal *search* tree on the values? A legal *2-3-tree* on the values? A legal *heap* on the priorities? A legal *treap*? For each answer, give a *very brief* explanation.

   Solution: Search tree on values: yes. This is because for every node, all the values in its left/right subtrees are less/greater than the value at the node.

   2-3 tree on values: no. In a legal 2-3 tree, every internal (non-leaf) node must have one more child than the number of values stores in it. But here the (3,9) node is missing a left child. (An incorrect explanation here is that "all leaf nodes must be at the same level;" while 2-3 trees do have that requirement, it holds for this tree!)

   Heap on priorities: no. Every parent has a higher priority than its children, and all leaves are at the same level, but the leaves are not filled from left to right. This is needed so that a heap can be represented by an array (without pointers).

   Treap: yes. It is a legal search tree and the priorities obey the heap condition. (In a treap, the leaves do not have to be filled from left to right.)

   (b) ACME corporation claims to have a comparison-based algorithm SortHeap that does the following: given any heap on n distinct elements, SortHeap outputs the elements in sorted order, in $o(n \log n)$ time. Prove that ACME's claim cannot be true. (*Hint:* show that if ACME's claim was true, then we could use SortHeap to do something impossible.)

   Solution: If ACME's claim were true, then we could comparison-based sort any given array $A[1 \dots n]$ of elements in $o(n \log n)$ time (thus violating the sorting lower bound), as follows: first, call BuildHeap(A) to arrange A into a heap in $O(n)$ time, then call SortHeap(A) to output the elements in sorted order. The total runtime is $O(n) + o(n \log n) = o(n \log n)$.

   Note that it is *not* correct to simply claim that "SortHeap sorts in $o(n \log n)$ time, which violates the sorting lower bound." The sorting lower bound applies to sorting an input array that may be presented in *any* order, but heaps have additional structure that could conceivably make it possible to sort them faster. For example, we know that it is possible to sort any given binary search tree in just $O(n)$ time (with zero comparisons!), using an in-order walk.

2. (10 points) Consider a hash table with $m = \Theta(n)$ slots for n elements, using simple chaining to resolve hash collisions. Recall that assuming an "ideal" hash function, all hash table operations run in *expected* $O(1)$ time. But in the *worst case*—when very unlucky hash values lead to many collisions—some operations can take as much as $\Theta(n)$ time.

   To deal with this problem, your friend suggests using balanced binary search trees to resolve collisions. He claims that in terms of asymptotic runtimes, this change never has any "downside." More precisely, he says that *no matter what the hash values happen to be*, none of the hash table operations can become asymptotically slower when using balanced BSTs instead of chaining, and in some cases they can become asymptotically faster.

   Is your friend's claim correct? Explain why or why not.

   Solution: Your friend is almost correct, but not quite. While element lookups and deletions could only improve (asymptotically), from $\Theta(n)$ with lists to $\Theta(\log n)$ with balanced BSTs, *insertion* can become asymptotically slower: with lists it is always just $\Theta(1)$ time no matter what the hash values are (since we just prepend the new element to the list), whereas with balanced BSTs it would be $\Theta(\log n)$ time if all the elements collide on one hash value. This one counterexample is enough to demonstrate that your friend's claim is false.

3. (25 points) In a set S of distinct values, a "closest pair" is two distinct values $v, v' \in S$ such that $|v - v'|$ is as small as possible. Note that there can be multiple closest pairs. If S has fewer than two elements, then there is no closest pair.

   Suppose you are using a balanced binary search tree to maintain a dynamic set of distinct values. You also need a function $\mathsf{ClosestPair}(T)$, which should return some closest pair in the set represented by T (if one exists).

   (a) Letting n be the number of nodes in the tree, describe a $\Theta(n)$-time implementation of $\mathsf{ClosestPair}$ that uses only the basic data stored in the BST (no augmentation).

   Solution: We first start with an observation: a closest pair of elements must be *adjacent* when we consider all the elements in sorted order. For if v, v' are not adjacent, then there is some $w \in S$ strictly between them, and $|v - w| < |v - v'|$, so v, v' cannot be a closest pair.

   To find a closest pair in $\Theta(n)$ time, we can simply do an in-order tree walk, keeping track of the differences between adjacent elements, and return a pair whose difference is smallest.

   (b) In a binary search tree rooted at T, there must be a closest pair among four candidate pairs:

   1. a closest pair of the T.left subtree (if one exists),
   2. a closest pair of the T.right subtree (if one exists),
   3. T.value and $\underline{???}$,
   4. T.value and $\underline{???}$.

   Describe the missing values above, and briefly argue why there must be a closest pair among these four pairs.

   Solution: The two missing values are: the maximum of the elements in T's left subtree, and the minimum of the elements in T's right subtree (when they exist). Another correct but slightly different answer is T's predecessor and successor in the *full* tree. These are not quite equivalent answers because T's predecessor/successor may not actually belong to the subtree rooted at T (e.g., when T's left/right child is nil), but may be somewhere else in the tree.

   These answers are correct for the following reason: since a closest pair v, v' of the subtree rooted at T must be adjacent, there are three possibilities: either v, v' are both in the left/right subtree of T, in which case they must be a closest pair of the left/right subtree. Or, one of them is T.value and the other is T's predecessor or successor in the subtree rooted at T, which are simply the max/min of T's left/right subtrees. It is impossible for v, v' to belong to different child subtrees of T and still be a closest pair, because then T.value would be between them.

(c) To support a fast ClosestPair operation, you decide to augment each node in the tree with some extra attributes. Describe attributes that have the following properties, and give brief justifications:

- they require only $O(1)$ extra storage per node;
- given a node and the correct attributes for its children, the node's attribute values can be computed in $O(1)$ time;
- they allow ClosestPair to be implemented in $O(1)$ time.

Solution: The previous part suggests which data we should keep at each node. Specifically, we store the minimum and maximum values in the subtree rooted at the node, and a closest pair in the subtree. Clearly these four values take just $O(1)$ space per node, and ClosestPair can be implemented in $O(1)$ time by just returning the closest pair stored at the root node. Moreover, by the previous part we can set the attributes at a node in $O(1)$ time given the (correct) attributes of its children, by examining the stored closest pairs of the left/right child subtrees, and the node's value together with the max/min of the left/right child subtrees.

(d) Following the insertion of a new element x into the tree, describe which nodes in the tree might need to have their attributes updated. Briefly explain why these updates do not increase the asymptotic runtime of Insert.

Solution: After inserting x, we only need to update the attributes of the nodes in x's insertion/search path. This is because none of the subtrees "hanging off" the insertion path have been changed, so their attributes must remain the same. By the previous part, we can update the attributes from x up to the root in $O(1)$ time per node, which does not increase the asymptotic $O(h)$ runtime of Insert.

Although we did not ask about this, it is also true that following a rotation, only two nodes need their attributes to be updated, and this can be done without increasing the asymptotic $O(1)$ runtime of rotation. So the $O(\log n)$ runtimes of all the BST operations can be preserved while maintaining the attributes correctly.

Scrap paper — no exam questions here.