Problem Set 3

All parts are due March 17, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 3-1. [25 points] **AVL trees**

An AVL tree, as presented in Lecture 6, is a certain kind of balanced Binary Search Tree. Between operations, an AVL tree satisfies the invariant that, for every internal node u in the tree, the heights of u's subtrees differ by at most 1. In particular, if we use the common convention that the height of an empty tree is -1, then if node u has only one subtree, that subtree must have height 0, that is, it must consist of just a single node.

The AVL tree maintains this invariant during an Insert operation by inserting the new entry at a new leaf node, then moving up the tree from the new leaf node, rebalancing as long as the difference in heights between a node's subtrees is greater than 1. A key fact is that, when a subtree rooted at a node u is rebalanced during an Insert operation, the height of the subtree after rebalancing is either the same as it was just before the rebalancing, or is reduced by exactly 1. This is important to ensure that rebalancing at one node cannot cause changes that could make it impossible to rebalance at higher nodes.

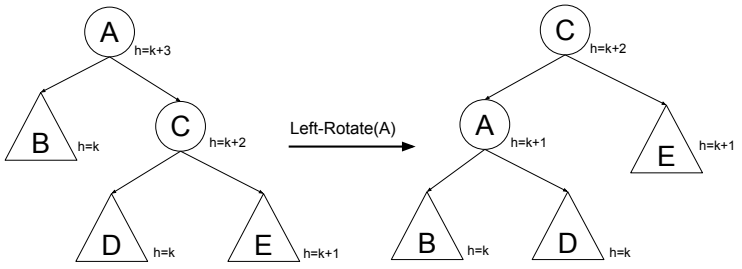

Figure 1: After rebalancing, the subtree rooted at A is now rooted at C, and the height is reduced by 1 from $k+3$ to $k+2$.

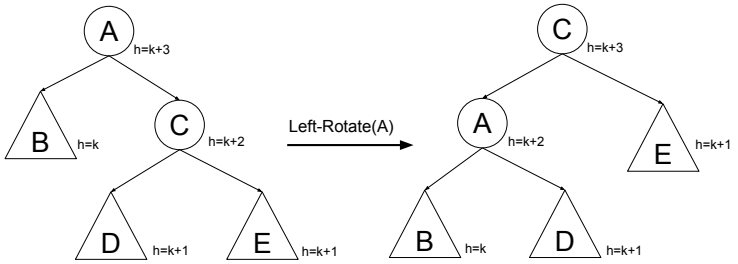

Figure 2: After rebalancing, the subtree rooted at A is now rooted at C, and the height remains the same, $k + 3$.

One might wonder whether AVL trees do too much work by rebalancing every time the tree gets even a little out of balance. In this problem, we will consider allowing a little more leeway, maintaining the invariant that the heights of each node's subtrees differ by at most 2. (In particular, if u has only one subtree, it must have height at most 1.) Let's call the new type of tree an *AVL-2 tree*.

(a) [4 points] Describe in words an algorithm based on rebalancing to implement the Insert operation on an AVL-2 tree.

(b) [6 points] Show that, when a subtree is rebalanced during your AVL-2 Insert algorithm, the height of the subtree after rebalancing is either the same as it was just before the rebalancing, or is reduced by exactly 1.

(c) [5 points] Give an example sequence of insertions, starting from an empty tree, for which an AVL tree rebalances at least twice but an AVL-2 tree does not rebalance at all.

(d) [5 points] Prove that AVL-2 trees always have height $O(\log n)$, where n is the number of nodes in the tree.

(e) [5 points] We can generalize the construction of AVL-2 trees to AVL-c trees, where c is any positive integer. Namely, define an *AVL-c tree* to be the same as an AVL tree except that the invariant is that the heights of each node's subtrees differ by at most c. (In particular, if u has only one subtree, it must have height at most $c - 1$.) Prove that, for any positive integer c, AVL-c trees always have height $O(c \log n)$, where n is the number of nodes in the tree.

Problem 3-2. [15 points] **Lower Bound on Sorting**

Suppose that we are given a sequence A of $n = mk$ elements to sort, with no duplicate elements. But now, the input sequence is not completely arbitrary: It consists of m successive subsequences, $A_1, A_2, \ldots, A_m$, each containing exactly k elements. Each of the subsequences is unsorted. However, for every i and j, $i \neq j$, either all elements of A_i are less than all the elements of A_j, or vice versa.

For example, A might be the sequence $[10, 12, 11, 5, 4, 6, 9, 7, 8, 2, 3, 1]$, with $n = 12$, $m = 4$, $k = 3$, $A_1 = [10, 12, 11]$, $A_2 = [5, 4, 6]$, $A_3 = [9, 7, 8]$, and $A_4 = [2, 3, 1]$.

(a) [3 points] Explain briefly why the given constraints imply that there is some way of sorting A in which all the elements of each A_i appear consecutively.

(b) [4 points] Considering all possible inputs A satisfying the given constraints, exactly how many different possible permutations of the indices in A can occur in the sorted outputs?

(c) [4 points] Prove an asymptotic lower bound on the number of comparisons needed (in the worst case) to sort all arrays A satisfying the given constraints.

(d) [4 points] Describe (in words) an algorithm that sorts an input sequence A that is known to satisfy the given constraints. The values n, m, and k are parameters of the algorithm, so the algorithm can use knowledge of those. Analyze the time complexity of your algorithm, and make your runtime as tight as possible, in particular, try to make it match the lower bound on comparisons that you proved in part (c).

Problem 3-3. [20 points] **Alphabetizing the Phone Book**

Design an algorithm to alphabetize an unsorted list of names and phone numbers. Specifically, assume that you are given an arbitrarily long list L of entries that are triples of the form (last name, first name, phone number). Last names and first names consist of letters of the alphabet, and the distinction between upper and lower case is not significant. The list contains no duplicate triples, but it may contain the same full name with different phone numbers. Your program should output a new list containing the same entries, but the names should be alphabetized in the usual way, according to last name and for the same last name, according to first name. Moreover, all the triples containing the same full name (and different phone numbers) should appear in the same order in the output list as in the input list.

(a) [10 points] Describe carefully, in words, an algorithm that, given a list L of n triples of the form (last name, first name, phone number) having no duplicate triples, sorts the list based on last names. Moreover, all the triples containing the same last name should appear in the same order in the output list as in the input list. Your algorithm should run in time $O(k_l n)$, where k_l is an upper bound on the lengths of any last name in the list.

(b) [6 points] Now describe carefully, in words, an algorithm for the main problem: given a list L of n triples of the form (last name, first name, phone number) having no duplicate triples, it should sort the list based on last names, and for the same last name, according to first name. All the triples containing the same full name (both first and last name) should appear in the same order in the output list as in the input list. Your algorithm should run in time $O(kn)$, where k is an upper bound on the lengths of any last names and any first names in the list.

(c) [4 points] Analyze the running times of your algorithms from parts (a) and (b).

Problem 3-4. [20 points] **Analysis of Hashing With Chaining**

Analysis of hashing with chaining usually assumes a "simple uniform hashing" property of the operation sequence: regardless of what has happened in the past, the key that appears in the next operation in the sequence is equally likely to be one that hashes to any location in the table. This assumption makes analysis easier, and is approximately achieved by real hash functions.

This problem considers some issues involved in analyzing the behavior of hash tables with chaining, under the simple uniform hashing assumption and some variations. In all cases, we assume that the hash table has m locations numbered $0, \ldots, m-1$. We are considering scenarios in which n Insert operations occur for successive, distinct keys $k_1, k_2, \ldots, k_n$, followed by a single Search operation.

(a) [3 points] Suppose that the n Inserts are all for keys that hash to table location 0. This is unlikely according to the simple uniform hashing assumption, but it could happen. Then the Search operation requests a key randomly, in such a way that the resulting hash value is equally likely to be any location in the hash table. Give a good upper bound on the expected time for the Search operation to complete, as a function of n and m. (Use exact bounds rather than asymptotic bounds where possible.)

(b) [3 points] Now consider a different pattern of Insert locations. Suppose that, out of the n Inserts, half are for keys that hash to location 0 and half for keys that hash to location 1. Again, the Search operation requests a key randomly, so that the resulting hash value is equally likely to be any location in the hash table. Again, give a good upper bound on the expected time for the Search operation to complete, as a function of n and m.

(c) [3 points] Generalize the result and your analysis for parts (a) and (b) to any other possible pattern of Insert locations.

However, the situation becomes different if we consider only successful Search operations:

(d) [3 points] As in part (a), suppose that the n Inserts are all for keys that hash to location 0. But now suppose that we know that the Search is successful, that is, it requests a key that is actually in the table. Suppose that it is equally likely to be any key in the table. Give a good asymptotic upper bound on the expected time for the Search operation to complete.

(e) [8 points] But bad tables as in part (d) are not so likely, under the simple uniform hashing assumption. So now let's consider tables that arise as a result of random choices of hash locations for the n Insert operations. Namely, assume that, for each Insert operation, regardless of what has happened in the past, the key that appears in the operation is equally likely to hash to any location in the table.

Suppose again, as in part (d), that we know that the Search is successful, that is, it requests a key that is actually in the table, and suppose that it is equally likely to be any key in the table. Now prove an upper bound of $\frac{n}{m} + O(1)$ on the expected time for

the Search operation to complete.
Hint: For each inserted key k_i, define a random variable CH_i representing the number of inserted keys that hash to the same location as k_i, counting k_i itself, that is, the final length of the chain in the location that k_i hashes to. Analyze the expected value of CH_i.

Problem 3-5. [20 points] **Ancient Scrolls**

At the *Centre for the Study of Ancient Documents*, researchers examine ancient scrolls written in ancient languages and try to determine who wrote them, and what they mean. When you arrive for your summer internship at CSAD, you are given a large pile of long scrolls written in an unknown language over a fairly small, unfamiliar character alphabet A. Your assigned job is to try to figure out the author of each scroll. Actually, you are supposed to write a program that will allow CSAD researchers to efficiently determine the authorship of such scrolls in the future.

Fortunately, the researchers at CSAD have already assembled a list of m possible authors. Each author has a "signature", which is a string of symbols from A that appears in each of his or her documents. For some unknown cultural reason, or coincidentally, all signatures are of exactly the same length k. Unfortunately, a signature string may appear anywhere in a scroll.

(a) [12 points] Describe (in words and pseudocode) an efficient algorithm that searches through a scroll to find the first match of any of the signatures, if any appear in the scroll. If there is no such occurrence, then your algorithm should return "none". Your searching algorithm should run in time $O(n + k)$, normally, where n is the length (number of characters) in the scroll and k is the (fixed, as described above) length of a signature. This time does not include any time for preprocessing the list of signatures, but you should try to keep the preprocessing time to $O(mk)$.

(b) [4 points] Analyze the time complexity of your preprocessing algorithm for the list of signatures.

(c) [4 points] Analyze the time complexity of your main algorithm, which searches the string looking for matches.

Part B

There is no Part B this time!