Problem Set 4

This problem set is due in lecture on Monday, October 23, 2006. The homework lab for this problem set will be held 2–4 P.M. on Sunday, October 22, 2006, in room 32-124.

Reading: Chapters 12, 13, and 18.

Both exercises and problems should be solved, but only the problems should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

Exercise 4-1. Do Exercise 12.2-2 on page 260 of CLRS.

Exercise 4-2. Do Exercise 12.3-5 on page 264 of CLRS.

Exercise 4-3. Do Exercise 18.2-3 on page 447 of CLRS.

Exercise 4-4. Do Exercise 18.3-2 on page 452 of CLRS.

Problem 4-1. AVL Trees. An AVL tree is a binary search tree that is height balanced, i.e., for each node \( x \), the heights of the left and right subtrees of \( x \) differ by at most 1. To implement an AVL tree, we maintain an extra field in each node: \( h[x] \) is the height of the node \( x \). Our first goal is to prove that BST-SEARCH runs in \( O(lg n) \) time on an AVL tree.
Problem Set 4

(a) Prove that an AVL tree with \( n \) nodes has height \( \Theta(\log n) \). (Hint: Prove that an AVL tree of height \( h \) has at least \( F_h \) nodes, where \( F_h \) is the \( h \)th Fibonacci number.)

Our remaining goal is to show how to insert a node into an AVL tree while maintaining the height-balance invariant. To insert a key into an AVL tree, we first place a node with that key in the appropriate place in the binary search tree order, using BST-INSERT. After this insertion, the tree may no longer be height balanced. Specifically, the heights of the left and right children of some node may differ by 2.

(b) Describe a procedure BALANCE\( (x) \) that takes a node \( x \), whose left and right child subtrees are height balanced and have heights that differ by at most 2, i.e.,

\[
|h[right[x]] - h[left[x]]| \leq 2,
\]

and alters the subtree rooted at \( x \) to be height balanced. (Hint: Use rotations.)

(c) Using part (b), describe a procedure AVL-INSERT\( (T, z) \) that, given an AVL tree \( T \) and a newly created node \( z \) (whose key has already been filled in), inserts \( z \) into \( T \).

(d) Prove that AVL-INSERT, run on an \( n \)-node AVL tree, takes \( \Theta(\log n) \) time and performs \( O(1) \) rotations.

Problem 4-2. Merging of data structures. In this problem, you will be asked to design efficient algorithms to merge various kinds of data structures. When we talk about the running time of an algorithm for merging two structures of sizes \( n_1 \) and \( n_2 \), we mean the running time as a function of the sum \( n = n_1 + n_2 \) of the inputs lengths. In particular, a merging algorithm runs in linear time if it takes \( O(n) = O(n_1 + n_2) \) time.

(a) Give a linear-time algorithm to merge two sorted linked lists \( L_1 \) and \( L_2 \) into a new sorted linked list \( L \).

(b) Call a binary search tree \( T \) nearly balanced if there exists some integer \( k \) such that every path from the root of \( T \) to a leaf of \( T \) has length either \( k \) or \( k + 1 \). Give a linear-time algorithm to merge two binary search trees \( T_1 \) and \( T_2 \) into a nearly balanced binary search tree \( T \). You can assume that all keys in the binary trees are distinct.

(c) Give a linear-time algorithm to merge two well formed red-black trees into a new well formed red-black tree. You can assume that all keys in the red-black trees are distinct.

(d) Give an \( O(\log n) \)-time algorithm to merge two 2-3-4 trees \( T_1 \) and \( T_2 \) into a new 2-3-4 tree \( T \), assuming that all keys stored in \( T_1 \) are smaller than all keys stored in \( T_2 \). You can assume that all keys in the 2-3-4 trees are distinct.