15-451 Algorithms, Fall 2003

Homework # 3 **due: Tuesday October 7, 2003**

Please hand in each problem on a separate sheet and put your **name** and **recitation** (time or letter) at the top of each sheet. You will be handing each problem into a separate box, and we will then give homeworks back in recitation.

Remember: written homeworks are to be done **individually**. Group work is only for the oral-presentation assignments.

Problems:

(35 pts) 1. **Hashing.** As discussed in class, the notion of *universal* hashing gives us guarantees that hold for *arbitrary* (i.e., worst-case) sets S, in expectation over our choice of hash function. In this problem, you will work out what some of these guarantees are.

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

(b) Let H be a universal family of hash functions from some universe U into a table of size m. Let $S \subseteq U$ be some set we wish to hash. Prove that if we choose h from H at random, the expected number of pairs (x, y) in S that collide is $O(|S|^2/m)$.

(c) Prove that for some constant c, with probability at least 3/4, no bin gets more than $1+c|S|/\sqrt{m}$ elements. (So, if $|S| = m$, you are showing that with probability 3/4 no bin gets more than $1 + c\sqrt{m}$ elements.) Hint: use part (b).

To solve this question, you should use "Markov's inequality". Markov's inequality is a fancy name for a pretty obvious fact: if you have a non-negative random variable X with expectation $\mathbf{E}[X]$, then for any $k > 0$, $\mathbf{Pr}(X > k\mathbf{E}[X]) \leq 1/k$. For instance, the chance that X is more that 100 times its expectation is at most 1/100. You can see that this has to be true just from the definition of "expectation".

(30 pts) 2. **Treaps and amortized analysis.** Suppose you have an array of n keys that is already sorted, and you want to convert it into a treap (e.g., so that you can later do additional inserts). Here is a procedure for converting the array into a treap in linear time, no matter what the priorities are — we won't be relying on the priorities being chosen randomly here. The procedure walks down the array, inserting the elements one at a time in a special way. Your job is to show that the amortized cost per insert for this procedure is $O(1)$.

First of all, in addition to keeping a pointer to the root node, we will also keep a pointer to the rightmost node of the treap. (The rightmost node is the one with the largest key so far). Also, every node will have a parent pointer in addition to left-child and right-child pointers.

Algorithm. Let A be the input array, where the ith key and priority appear in $A[i].key$ and $A[i].prio$ respectively, and the keys are in sorted order. We will insert the elements one by one, into an initially empty treap T.

We insert element i into the treap T made of elements $1 \cdots (i-1)$ as follows:

(a) if $A[i].prio$ is less than the priority of the root of T, then i becomes the new root and T is made into its left child;

(b) if $A[i].prio$ is greater than the priority of the rightmost node in the treap, then element i is made into the right child of this node;

(c) if $A[root].prio < A[i].prio < A[right].prio$, then element i is temporarily made the right child of the rightmost node, and the heap property of the treap is then restored by successive rotations of the newly inserted node. (Note: $A[right]$ is really the same thing as $A[i-1]$ since the keys are in sorted order.)

Cases (a) and (b) above are clearly constant-time. The problem is that case (c) could involve a lot of rotations. You job is to show that nonetheless, the amortized time per operation is $O(1)$.

(35 pts) 3. **BSTs and dynamic programming.**

Consider a binary search tree storing a set of keys $x_1 < x_2 < x_3 < \ldots < x_n$. Let's define the *cost* of handling a request for some key to be the number of comparisions made in searching for it (one plus the distance of the node from the root of the tree). For example, if the root is requested, the cost is 1.

Given a particular sequence of requests, one can calculate the cost that would be incurred on that sequence by different possible binary search trees. The tree that attains the minimum cost is called the *optimal binary search tree* for that sequence.

(a) For a fixed tree, the cost of a given sequence of requests clearly only depends on the number of times each key is requested, not on their order. Suppose that $n = 4$ and that x_1 is accessed once, x_2 is accessed 9 times, x_3 is accessed 5 times, and x_4 is accessed 6 times. Find an optimal binary tree for this set of requests. (There is more than one possible answer.)

(b) In general, suppose the optimal binary search tree has x_i at the root, with L as its left subtree and R as its right subtree. Prove that L must be an optimal binary search tree for elements $x_1, \ldots, x_{i-1}$ and R must be an optimal binary search tree for elements $x_{i+1}, \ldots, x_n$.

(c) Give a general algorithm for constructing the optimal binary tree given a sequence of counts $c_1, c_2, \ldots, c_n$ (c_i is the number of times x_i is accessed). The running time of your algorithm should be $O(n^3)$. Hint: use dynamic programming.

Note #1: the notion of an optimal binary search tree is a lot like the notion of a Huffman tree, except that we also require the keys to be in search-tree order. This requirement is the reason that the greedy Huffman-tree algorithm doesn't work for finding optimal BSTs.

Note #2: it's actually possible to improve the running time to $O(n^2)$ by a simple modification to this dynamic-programming solution. But proving correctness for this faster version is a real pain.