15-451 Algorithms, Fall 2009

Homework # 3 **Due: Thursday October 8, 2009**

Please hand in each problem on a separate sheet and put your **name**, **andrew id**, and **recitation** (time or letter) at the top of each sheet. You will be handing each problem into a separate stack, so it is important to have your name on each sheet to make sure you get credit for your work!

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

(25 pts) 1. **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 $O(1)$.

(25 pts) 2. **Hashing.** Some short-answer questions on hashing.

(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 $H = \{h_1, \ldots, h_k\}$ be a universal family of hash functions from some universe U ($|U| \geq 2$) into $\{0, 1\}$. Could it be that some function $h_i \in H$ maps all of U to 0? Explain.

(c) Let $H = \{h_1, \ldots, h_k\}$ be a universal family of hash functions from some universe U ($|U| \geq 2$) into $\{0, 1\}$. Let C_i be the number of pairs $x, y \in U$ ($x \neq y$) such that $h_i(x) = h_i(y)$. So, C_i is at most $\binom{|U|}{2}$. If all $h_i \in H$ have $C_i > \frac{1}{2}\binom{|U|}{2}$, could H be universal? Explain.

(25 pts) 3. **The List-Update Problem.** Suppose we have n data items $x_1, x_2, \ldots, x_n$ that we wish to store in a linked list in some order. Let's say the cost for performing a *lookup*(x) operation is \$1 if x is in the head of the list, \$2 if x is the second element in the list, and so on.

For instance, say there are 4 items and it turns out that we end up accessing x_1 3 times, x_2 5 times, x_3 once, and x_4 twice. In this case, in hindsight, the best ordering for a linked list would have been (x_2, x_1, x_4, x_3) with a total cost of \$21.

The *Move-to-Front* (MTF) strategy is the following algorithm for organizing the list if we don't know in advance how many times we will access each element. We begin with the elements in their initial order $(x_1, x_2, \ldots, x_n)$. Then, whenever we perform a *lookup*(x) operation, we move the item accessed to the front of the list. Let us say that performing the movement is free. For instance, if the first operation was *lookup*(x_3), then we pay \$3, and afterwards the list will look like $(x_3, x_1, x_2, x_4 \ldots)$.

(a) Suppose $n = 4$ and we use MTF starting from the order (x_1, x_2, x_3, x_4). If we perform the following 4 operations:

$$lookup(x_4), lookup(x_2), lookup(x_4), lookup(x_2).$$

What does the list look like in the end and what was the total cost?

(b) Your job is to prove that the total cost of the MTF algorithm on a sequence of m operations (think of m as much larger than n) is at most $2C_{static} + n^2$ where C_{static} is the cost of the best static list in hindsight for those m operations (like in our first example). We will prove this in two steps.

i. First prove the somewhat easier statement that the cost of Move-to-Front is at most $2C_{initial}$ where $C_{initial}$ is the cost of the original ordering $(x_1, x_2, \ldots, x_n)$.

Hint: If $i < j$ but x_j is in front of x_i in the MTF list, let's say that x_j has "cut in line" in front of x_i. Now, imagine that each element x_i has a piggy bank with \$1 for everyone that is currently cutting in line in front of it.

ii. Now prove the $2C_{static} + n^2$ bound.

Note: one nice use of this is for *data compression*. You store each ascii character in a list in this way, and then when reading a string of text, for each character you output its index i in the list before moving the character to the front (this requires only $O(\log i)$ bits, which will be small if the item was close to the front of the list).

(25 pts) 4. **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 (1 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.[1]

(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 for a given set of requests 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 the requests to elements $x_1, \ldots, x_{i-1}$ and R must be an optimal binary search tree for the requests to 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 very tricky.

[1] This is the exactly analogous to the "best static list in hindsight" in the previous problem. The big difference is that for *lists*, the best static list in hindsight is very simple: you just put the most-requested item in the front, then the next-most-requested, etc. For *trees*, computing the optimal binary search tree will be more complicated, requiring dynamic programming, which is what this whole problem is about.