15-451 Algorithms, Fall 2007

Homework # 3 **due: Tuesday October 9, 2007**

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.

(25 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 a set of m elements 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 $\leq \frac{m-1}{2}$.

(c) Prove that with probability at least 3/4, no bin gets more than $1+2\sqrt{m}$ 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".

(25 pts) 2. **Doing more with search trees.** Often, by adding extra information to the nodes of a binary search tree, it is possible to perform other operations you might be interested in. In particular, suppose we want to be able to do the following operations efficiently:

- Given k, output the kth smallest element (let's assume all keys are distinct).
- Given a key x, do a version of lookup(x) that tells us the rank of x (how many keys are smaller than x).

Note that if we had a fixed set of data, this would be easy: we just make a sorted array. If we are asked to produce the kth smallest element, we just output $A[k]$. To get the rank of x, we just do binary search to find it, and then output which location it's in.

How could we do this with search trees, if we want to do inserts too? In particular, describe a piece of additional information you could store at each node of the search tree such that:

(1) it allows us to perform the above operations in time O(depth of tree).

(2) the values can be maintained efficiently (time proportional to the depth of the tree) when a new node is inserted.

For example, a *bad* way of solving the problem would be to have each node store the rank of its key. This is bad because if you insert a new key that is smaller than everything else, you may have to update *everyone's* rank, so we don't have property (2). For this problem you should:

(a) Describe the extra information you will store at each node of the tree.

(b) Describe how you can use this to find the kth smallest element efficiently, and how you can use this to find the rank of a given key efficiently.

(c) Describe how the information can be updated efficiently when a new node is inserted into the tree. (You can assume for this that we are doing simple binary search tree insertion, though if you like you can also describe how it is updated when rotations are performed).

(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.