15-451 Algorithms, Fall 2011

Homework # 3 **Due: October 11, 2011**

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. **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) 3. **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 (in addition to solving a few short-answer questions).

(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 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 $\leq \frac{m-1}{2}$.

(d) 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 with probability at least $3/4$, no bin gets more than $1 + 2\sqrt{m}$ elements. Hint: use part (c).

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) 4. **Knapsack revisited.**

Recall from class that in the knapsack problem we have n items, where each item i has an integer value v_i and an integer size s_i, and we also have a knapsack of size S. The goal is to find the maximum total value of items that can be placed into the knapsack. In particular, out of all sets of items whose total size is at most S, we want the set of highest total value. In class, we gave a dynamic programming algorithm to solve this problem whose running time was $O(nS)$.

One issue, though, is that if the sizes are large, then $O(nS)$ may not be so good. In this problem, we want you to come up with an alternative algorithm whose running time is $O(nV)$, where V is the value of the optimal solution. So, this would be a better algorithm if the sizes are much larger than the values.

Note: your algorithm should work even if V is not known in advance, but you may want to first assume you are given V up front and then afterwards figure out how to remove that requirement.

Hint: it might help to look at the algorithm from class and think about what the subproblems were. Then think about what subproblems you want to use for the new goal.