15-451 Algorithms, Fall 2000

Homework # 3 **due: October 10, 2000**

Groundrules: You should try to do the assignment as much as possible by yourself. If you get stuck you may ask for help from friends or TAs or the instructors. In any event, you should do your writeup by yourself, in your own words. If you did get help from others, please list their names at the top of the first page of your assignment. Your assignment is due at the beginning of class on the due date.

Problems:

(15 pts) 1. [bits and trits] Suppose we have a binary counter such that the cost to increment the counter is equal to the number of bits that need to be flipped. We saw in class that if the counter begins at 0, and we increment the counter n times, the amortized cost per increment is just $O(1)$. Equivalently, the total cost to perform all n increments is $O(n)$.

Suppose that we want to be able to both increment *and* decrement the counter.

(a) Show that even without making the counter go negative, it is possible for a sequence of n operations starting from 0, allowing both increments and decrements, to cost as much as $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

(b) To fix the problem from part (a), consider the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or −1. The value of the number $t_{k-1}, \ldots, t_0$ (where each t_i is a trit) is defined to be

$$\sum_{i=0}^{k-1} t_i 2^i.$$

The process of incrementing a ternary number is analogous to that operation on binary numbers. One is added to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. One is subtracted from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated.

The cost of an increment or a decrement is the number of trits that change in the process. Starting from 0, a sequence of n increments and decrements are done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e., the total cost for the n operations is $O(n)$). Hint: think about a "bank account" or "potential function" argument.

(10 pts) 2. [Building a treap from a sorted array] A treap is a special kind of binary search tree (BST) for storing keys that have associated *priorities*. The (key, priority) pairs are arranged in a treap acording to two properties:

- The keys satisfy the BST invariants (left keys $\leq$ parent key $\leq$ right keys).
- The priorities form a *heap* (priority of a node is $\geq$ priority of its children).

It is not obvious that a treap always exists, but it does. Below we give an algorithm that builds a treap given the pairs $(key, prio)$, sorted in order by the key component. Your task is the following:

(a) Draw the treap produced using this algorithm from the input sequence:
`((a 5), (b 7), (c 2), (d 4), (e 1), (f 6), (g 9), (h 3), (i 8))`.

(b) Prove that the algorithm spends an $O(1)$ amortized cost per insert. I.e., the total cost to perform n insertions is $O(n)$.

Algorithm. Let's say the input is given in an array A, 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, so at each step, we will be inserting a key into the treap made of the smaller keys. In order to do that, we will keep track of the root node $root$ and the rightmost node $right$ of the treap.

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

- if $A[i].prio$ is greater than the priority of the root of T, then i becomes the new root and T is made into its left child;
- if $A[i].prio$ is less than the priority of the rightmost node in the treap then element i is made into the right child of this node;
- if $A[right].prio < A[i].prio < A[root].prio$, 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.)

Here is the pseudocode for the algorithm: (all parameters are transmitted by reference.)

BuildTreap(A, n)

1. $root = 1$;
2. for $i = 2$ to n: InsertKey$(root, i-1, i)$

InsertKey$(root, right, i)$

1. if $A[i].prio \geq A[root].prio$ then
   $A[i].leftchild = root$; $A[root].parent = i$;
   $root = i$;
2. if $A[i].prio \leq A[right].prio$ then

$A[right].rightchild = i$; $A[i].parent = right$;

3. if $A[right].prio < A[i].prio < A[root].prio$ then

   $A[right].rightchild = i$; $A[i].parent = right$;

   while $(A[i].prio > A[A[i].parent].prio)$ Rotate(i);

Rotate(i) performs a left rotation, as presented in class for Splay trees. It "lifts" the given node to the parent position and makes the parent become its left child:

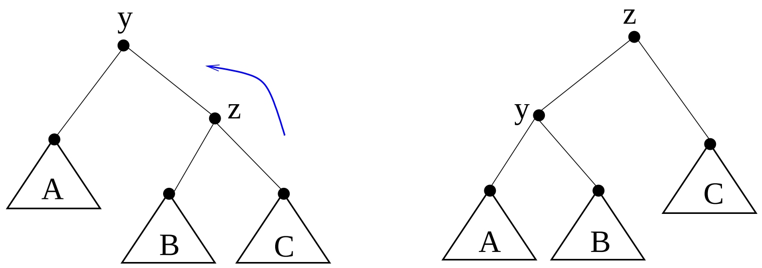
Rotate(z)

$y = A[z].parent$;

$A[y].rightchild = A[z].leftchild$;

$A[z].leftchild = y$;

Make y's parent point to z instead, and fix up parent pointers;



(15 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.

(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 $c|S|/\sqrt{m}$ elements.

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".