Michael Goldwasser
Loyola University Chicago

Homework #2: Recurrences, Amortization, Comparison Based Bounds
Due Date: Tuesday, 11 February 2003

Guidelines

Please make sure you adhere to the policies on collaboration and academic honesty as outlined in Handout #1.

Reading

Review Ch. 4 and read Ch. 8.1, 9, and 17 of CLRS.

Practice

These exercises are purely for your own practice. You should not turn them in, and you are free to discuss them fully with others.

- Do CLRS 4.2-4, 4.2-5
- Do CLRS 4.3-1
- Do CLRS 17.1-2
- Do CLRS 17.1-3 and 17.2-2 and 17.3-2
- Do CLRS 17.2-3
- Do CLRS 17.4-3

Problems

Problem A (20 points) "Work entirely on your own."

Give asymptotic <u>upper</u> and <u>lower</u> bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and prove your answers.

For some of these, you may use the Master Theorem as a proof so long as you justify that it applies. For others, you should prove your bounds by induction, using the substitution method.

i. $T(n) = 2T(n/2) + n^3$.
ii. $T(n) = T(9n/10) + n$.

iii. $T(n) = 16T(n/4) + n^2$.

iv. $T(n) = 7T(n/3) + n^2$.

v. $T(n) = 7T(n/2) + n^2$.

vi. $T(n) = 2T(n/4) + \sqrt{n}$.

vii. $T(n) = T(n-1) + n$.

viii. $T(n) = T(\sqrt{n}) + 1$.

Problem B (40 points) "You may discuss ideas with other students."

Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the field $size[x]$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \le \alpha < 1$. We say that a given node x is α-**balanced** if

$size[left[x]] \le \alpha \cdot size[x]$

and

$size[right[x]] \le \alpha \cdot size[x]$.

The tree as a whole is α-**balanced** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

i. A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in time $\Theta(size[x])$, and it can use $O(size[x])$ auxiliary storage.

ii. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than 1/2. Suppose that `Insert` and `Delete` are implemented as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then the subtree rotted at the highest such node in the tree is "rebuilt" so that it becomes 1/2-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

iii. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

iv. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

v. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

Problem C (20 points) "You may discuss ideas with other students."

Assume that we are interested in a data structure which supports two operations: SEARCH and INSERT. If we store the items in an unordered array, then INSERT can be done in $O(1)$ time but SEARCH would require $\Omega(n)$ time, where n is the current number of items.

Alternatively, if we keep the array sorted, then we can use *binary search* to implement SEARCH in $O(\lg n)$ time (if you are not familiar already with binary search, please consult one of the recommended readings or see a brief discussion in Exercise 2.3-5 of CLRS). Unfortunately, if we insist on keeping the array sorted, INSERT will require $\Omega(n)$ time in the worst case, as we may have to shift many items around.

In this problem, we are going to develop a way to accomplish these tasks while better balancing the time required for SEARCH and INSERT. Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We can keep k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, there is no particular relationship between elements in different arrays.

i. Describe how to perform the SEARCH operation for this data structure in $O(\lg^2 n)$ worst-case time. (justify the bound on the running time)

ii. Describe how to perform the INSERT operation for this data structure in $O(\lg n)$ *amortized* time. (justify the amortized bound on the running time).

Problem D (20 points) "You may discuss ideas with other students."

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y. (*Hint:* If $X[k]$ is the median of array X, how quickly can you determine whether it is also the median of the combined $2n$ elements? If it is not, did you learn any new information about the identity of the true median?)

Problem E **(EXTRA CREDIT – 10 points)**
"You may discuss ideas with other students."

Consider what happens if we wish to include a DELETE operation in the data structure developed in Problem C. (we will assume that the parameter to DELETE is a reference to the exact location in the structure which holds the item to be deleted – that is we will not need to perform a search to find the item being deleted.)

i. Argue that if we insist on using a structure where all of the arrays are either empty or full, there will always be a sequence of t operations, for any t, which require $\Omega(tn)$ time, and thus $\Omega(n)$ amortized time.

ii. If we allow you to relax the restriction that all arrays are either full or empty, show how to implement DELETE in $O(1)$ amortized time, while still maintaining the previous time bounds for SEARCH and INSERT. Make sure that you justify not only the analysis of DELETE, but also that you re-justify the previous bounds for the other operations on the new technique for the data structure.