TEL AVIV UNIVERSITY
Department of Computer Science
0368.4281 – Advanced topics in data structures
Spring Semester, 2007/2008

Final homework

Due Sunday 3.8.2008. Please keep a copy.

1. We insert elements into an initially empty binary search tree, while maintaining at each node v the height of the node, denoted by $h(v)$ and the size of the subtree rooted by v, denoted by $s(v)$. We insert an item as a leaf, then we walk up along the path to the root and check for each node v along this path whether $h(v) \leq 2 * \log_2(s(v))$. If we encounter a node v where $h(v) > 2 * \log_2(s(v))$ then we completely reorganize the subtree rooted at v as follows: We put the median at the root and recursively reorganize the items smaller than the median into the left subtree and the items larger than the median into the right subtree. (If the $s(v)$ is even we arbitrarily pick one of the medians.) Prove that a sequence of n insertions takes $O(n \log n)$ time.

2. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of distinct items totally ordered such that $a_i < a_j$ iff $i < j$. Assume that over a long sequence of m accesses to items in A, a_i is accessed $q(i) \geq 1$ times. Describe an algorithm to find a binary search tree T where each item a_i resides in a leaf of T such that if we serve the accesses using T, each time going from the root to the corresponding leaf, the total time its takes to serve the whole sequence is minimized. Prove
1) that your algorithm indeed constructs a tree that minimizes the total access time. 2) As tight upper bound as you can, on the total time it takes to serve the sequence using T. 3) An upper bound on the running time of your algorithm for constructing T. (Any polynomial time algorithm would be fine.)

3. Given a set S of n points in space (R^3) it is known how to build in $O(n^2 \log n)$ time a data structure D such that given a point q we can find the point in S which is closest to q in $O(\log n)$ time. (This is called a nearest neighbor query.) This data structure is static and does not allow to insert points to S or delete points from S.

Find a way to use D to obtain a data structure for the same problem that performs a nearest neighbor query in $O(\log^2 n)$ time and allows to insert a point in $O(n \log(n))$ amortized time, where n is the number of point in the data structure when we insert the new one.

4. We want to implement a binary counter that supports both increment and decrement using two regular counters c_1 and c_2 that support only increment. The value of the counter at any time is $c_1 - c_2$. We increment the counter by incrementing c_1 and we decrement the counter by incrementing c_2. We represent c_1 and c_2 as singly linked lists where each item in the list corresponds to a digit. The list starts with the the least significant and terminates with the most significant. A value of 0 is represented by an empty list and for a non zero value the last digit is the most significant "1". We start with $c_1 = c_2 = 0$. Complete the implementation and analyze it such that the following hold:

1. When the value of the counter is 0 then $c_1 = c_2 = 0$.
2. For $k > 0$, the length of the longest among c_1 and c_2 is k only if the maximum absolute value of the counter ($|c_1 - c_2|$) had been larger than 2^{k-1}.
3. The amortized time per operation is $O(1)$.

5. Consider the move to root heuristic on a tree containing the integers 1 to n in symmetric order (i.e. for every node v all items in the left subtree are smaller than the item at v, and the item at v is smaller than all items at the right subtree). Recall that in the move to root heuristic, after you access an item you perform rotations bottom up on the edge between the accessed item to its parent until the item is at the root. Assume $i < j$.
a) Prove that i is an ancestor of j if the most recent request for i has occured since the most recent request for any of $i + 1, \ldots, j$. Similarly, j is an ancestor of i if the most recent request for j has occured since the most recent one for any of $i, \ldots, j - 1$.
b) Prove that any tree T containing these items in symmetric order can be produced from any other tree containing these items in symmetric order by a sequence of at most n requests under the move to root heuristics.
c) For any m sufficiently large show a sequence of accesses such that serving it with the move to root heuristics takes $\Omega(n)$ time on average per access. (The initial tree is arbitrary.)

6. Show how to extend dynamic trees to support the operation $lca(u, v)$. This operation assumes that u and v are in the same actual tree and returns the lowest common ancestor of u and v in their tree.
Try to get a running time of $O(\log n)$ without hurting the running time of the other operations.

7. We propose the following data structure for the union-find problem. We represent each set by a tree of depth exactly 2. Each leaf contains an item which is at distance exactly 2 from the root. Each internal node in the tree maintains 1) the number of children it has, 2) a pointer to its parent, 3) a pointer to a singly linked list containing its children. We maintain the following invariants:

1. Each node other than the root has between 1 and $2 \log n$ children.
2. Say a child of the root is *full* if it has at least $\log n$ children. At most one child of the root is not full.

A find returns the root and takes $O(1)$ time. We implement union of two sets with roots x and y as follows. Assume without loss of generality that x has at least as many children as y. We make each full child v of y, a child of x. (This requires inserting v into the list of children of x, changing its parent to be x, and updating the number of children of x.) Let x' and y' be the non-full children of x and y, respectively. If either x' or y' does not exist, we make the other non-full child the non-full child of x. Otherwise, assume without loss of generality that x' has at least as many children as y'. We make every child of y' a child of x'. Node x' remains a child of x which is possibly not full.

Prove that a sequence of m union and find operations on a sequence of n elements takes $O(m + n \log \log n)$ time.