Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #2

due Sunday night, Oct. 7

Write your solutions in LaTeX using the template provided on the Moodle and web sites and upload your PDF file through Moodle by 4:00 am Monday morning, Oct. 8.

Question 1. (20 points)
A binary heap is a heap data structure based on a *complete binary tree*; that is, all levels of the tree, except possibly the last level are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right. Assume the root of the binary heap stores the smallest element. A binary heap supports two operations:

- `insert`, which adds a new element to the heap in a *bubble-up* way: first add the element to the leftist empty position in the bottom level of the heap. Then restore the heap order by iteratively comparing this new added element with its parent: if they are in the correct order, stop; if not, swap the new element with its parent.
- `delete`, which delete the smallest element of the heap in a *bubble-down* way: first, replace the root of the heap with the rightist element in the bottom level. Then restore the heap order by iteratively comparing the new element with its two children: if both are in the correct order, stop; if not, swap the element with one of its children that smaller than it.

Consider a binary tree with n elements.

1. Prove that both `insert` and `delete` have worst-case running time $O(\log n)$.
2. Propose a potential function such that the amortized cost of `insert` is still $O(\log n)$ while the amortized cost of `delete` is $O(1)$.

Question 2. (40 points)
Consider a binary search tree data structure that supports `find` and `add` operations. The `find` operation behaves the same as the general binary search tree. Let $1/2 < c < 1$ and $b = -1/\log c$ be two constants. The `add` operation which inserts x to the data structure takes the following procedure:

1. `find` the insert point of x and insert x to the tree. Record the depth of x, denoted as $d(x)$. If $d(x) \leq b \cdot \log n$, stop.
2. Traverse the path from x back to the root to find the first two consecutive vertices u and v on this path satisfying $w(u) \geq c \cdot w(v)$, where v is the parent vertex of u and $w(v)$ represents the number of vertices in the subtree rooted at v.

3. Denote the subtree rooted as v as $T(v)$. Build a new subtree T' and replace $T(v)$ with T' as follows: sort all the vertices in $T(v)$; choose the median of the sorted list and set it as the root of T' and build the left subtree of T' using the left part of the sorted list and build the right subtree of T' using the right part of the sorted list recursively.

Suppose the tree is empty in the beginning.

1. Prove if $d(x) > b \cdot \log n$, we can always find u and v satisfying $w(u) > c \cdot w(v)$ in step 2.
2. Prove the height of the tree is always bounded by $b \cdot \log n$.
3. Analyze the worst-case running time of `find` and `add`.
4. Propose a potential function to prove that both `find` and `add` have amortized running time of $O(\log n)$.

Question 3. (40 points)
A Lazy Leftist Tree data structure supports four operations: `meld`, `insert`, `delete-min` and `delete`. For each node of the tree, in addition to the length of the shortest path from this node to a leaf, it also need to store a bit to indicate whether this node has been deleted. The `meld` and `insert` operations behave the same as the general leftist tree. The `delete` operation removes a specified node in a *lazy* way: it just marks this node deleted by setting the bit of the specified node to `true`. (Note that we do not need to search for the specified node. You can assume that we are directly given the pointer of the to-be-deleted node.) The `delete-min` operation takes the following procedure: it first removes necessary nodes marked deleted to expose the nondeleted minimum element by recursively traversing the tree from the root and physically removing every node marked deleted, stopping at each nondeleted node (thus nodes marked deleted that have a nondeleted node on the path to the root are not removed); it then searches among all nondeleted roots resulting from this operation for the minimum key, removes it, and finally melds all subtrees produced.

1. Prove that the worse-case running time for `delete-min` is $O(k \log(n/k))$, where n is the size of the tree and k is the number of nodes that will be removed in this `delete-min` operation, i.e., k is the number of nodes marked deleted satisfying that all the nodes on the path to the root (including the root) are marked deleted.
2. Propose a potential function to prove the amortized running time for `insert`, `delete` and `delete-min` is $O(\log n)$. (You do not need to show the $O(\log n)$ amortized running time for `meld`.)