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 $O(1)$.

Solution:

1. The running time for both `insert` and `delete` is proportional to the height of the tree. Now we prove that a complete binary tree with n nodes has height $O(\log n)$. Suppose the height of this tree is h. Since the first $h-1$ levels are fully filled, we have $n > \sum_{k=1}^{h-1} 2^{k-1} = 2^{h-1} - 1$. Thus, $h \leq 1 + \log(n+1) = O(\log n)$.
2. Since the actual cost for a `delete` is $O(\log n)$, we need to design a potential function such that the decrease of the potential for `delete` is $O(\log n)$. Comparing the data structure with that before performing `delete`, we find that only a node in the bottom level is removed, whose length is $\log n$! Thus, we can define the potential as the sum of depth of all the nodes. For a `delete`, its actual cost is $O(\log n)$ while the decrease of the potential is also $O(\log n)$, which yields a constant amortized running time. For a `insert`, which adds a node in the bottom level of the tree, both its actual cost and the increase of the potential are $O(\log n)$; thus, the amortized running time is still $O(\log n)$.

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 $O(\log n)$.

Solution: This data structure is called *Scapegoat Tree* (we do not consider `delete` operation in this problem), which provides worst-case $O(\log n)$ search time, and $O(\log n)$ amortized insertion and deletion time. The advantage of Scapegoat Tree is that there is no additional fields are needed to be stored in each vertex.

1. Suppose that no such u and v exists, i.e., $w(u) \leq c \cdot w(v)$ for all consecutive vertices on the path from x to the root. Applying this formula along this path yields $1 < c^{d(x)} \cdot n$. Thus we have $d(x) \leq \log_c 1/n = b \cdot \log n$, which contradicts with the condition that $d(x) > b \cdot \log n$.

2. Since `find` is a read-only operation, we only need to show that each `add` operation cannot break this bound. Suppose an insertion of x increases the height of tree by 1 and violate the bound. Note that now x becomes the deepest vertex (and x is the only vertex in the deepest layer), and $d(x)$ is indeed the height of the tree. As stated in the procedure, $T(v)$ will be replaced by a new tree T'. We now prove that the height T' must be smaller than the height of $T(v)$, which consequently means this `add` operation will not increase the height of the tree and thus the bound still holds. To show that, we need two facts: first, easy to see that T' has the minimum height among all subtrees that contains vertices in $T(v)$; second, $T(v)$ do not have the minimum height, for the reason that $w(u) > c \cdot w(v)$, which means the size of subtree rooted at the sibling of u is much smaller than that of $T(u)$, thus we can always move x to the the subtree rooted at the sibling of u and decrease the height of $T(v)$.

3. The worst-case running time of `find` is proportional to the height of the tree, which is $O(\log n)$. For the `add` operation without rebuilding process, its running time is the same as `find`. For the `add` operation with rebuilding process, calculating $w(\cdot)$ for each vertex on the path from x to the v can be done in $O(w(v))$ time, and rebuilding the new subtree can also be done in $O(w(v))$ time. Thus, the worst-case running time for `add` operation with rebuilding process is $O(\log n + w(v))$, which is loosely $O(n)$.

4. Consider the case of rebuilding the subtree. Notice that before that vertex v is poorly balanced, i.e., the difference of the sizes of the two subtrees of v is at least $(2c-1)w(v)$, whereas after rebuilding, all vertices in the new subtree T' become completely balanced, i.e., the difference of the sizes of the two subtrees of any vertex is at most 1. This fact inspires us to use the sum of the *imbalance* of all vertices in the tree as the potential function. Formally, we define the imbalance of vertex u as $I(u) = \max\{|w(L(u)) - w(R(u))| - 1, 0\}$, where $L(u)$ and $R(u)$ represent the left children and right children of u respectively. and define the potential function as $\sum_u I(u)$.

   For the `find` operation, its real cost is $O(\log n)$ and it does not change the potential, thus its amortized running time is $O(\log n)$. For the `add` operation without rebuilding process, its real cost is $O(\log n)$ and imbalance change only occurs on those vertices on the path from x to the root. The number of vertices on that path is $O(\log n)$ and for each such vertex, the imbalance change is at most 1. Thus, it also has amortized running time of $O(\log n)$. For the `add` operation with rebuilding process, the actual cost is $O(w(v) + \log n)$. Consider the potential of $T(v)$: before rebuilding the potential of $T(v)$ is at least $(2c-1)w(v)$, and after rebuilding it is 0. This potential decrease can be used to pay for the $w(v)$ term in the actual cost. As for the imbalance change on the vertices that locate on the path from v to the root, again the number of these vertices is $O(\log n)$ and for each such vertex, the imbalance change is at most 1. Thus, it also has 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`.)

Solution:

1. The `delete-min` operation first removes some nodes marked deleted, which costs $O(k)$. Easy to see that this step yields $k+1$ subtrees. Then it searches and removes the node with the minimum key among these $k+1$ subtrees, for which the actual cost is also $O(k)$. Now we have $k+2$ subtrees, and in next step they are melded. We know that running time for melding is proportional to the sum of the length of the rightmost path of each subtree, which is $O(\sum_{i=1}^{k+2} \log s_i)$, where s_i is the size of the i-th subtree. Notice that all these subtrees are disjoint and all the removed nodes are not contained in any subtree, we have $\sum_{i=1}^{k+2} s_i \leq n-k-1$. By applying the inequality of arithmetic and geometric means, we have $\sum_{i=1}^{k+2} \log s_i = \log \prod_{i=1}^{k+2} s_i \leq (k+2) \log \frac{\sum_{i=1}^{k+2} s_i}{k+2} \leq (k+2) \log \frac{n-k-1}{k+2} = O(k \log(n/k))$. The running time of this step dominates the others. Thus, the total actual cost for this operation is $O(k \log(n/k))$.

2. For a tree with total n nodes among which k' nodes are marked deleted, define its potential as $c \cdot k' \log n$, where c is a constant to be determined.
   The actual cost of an `insert` operation is $O(\log n)$. Because it does not change the number of nodes marked deleted, the potential change is $c \cdot k' \log(n+1) - c \cdot k' \log n$, which is $O(1)$ since $k' = O(n)$. Thus, the amortized running time for `insert` is $O(\log n)$.
   The actual cost of a `delete` is $O(1)$, and the potential change is $c \cdot (k'+1) \log n - c \cdot k' \log n = O(\log n)$. Thus, the amortized running time is also $O(\log n)$.
   The total actual cost for `delete-min` operation is $O((k+2) \log(n-k-1))$. (Here we relax the above result.) Now consider the potential change, which is $c \cdot (k'-k) \log(n-k-1) - c \cdot k' \log n$. By setting c the same value with the constant factor in the actual running time, we have the amortized running time is $c \cdot (k'-k) \log(n-k-1) - c \cdot k' \log n + c \cdot (k+2) \log(n-k-1)$, which is $O(\log n)$.