Design and Analysis of Algorithms April 4, 2012
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Dana Moshkovitz and Bruce Tidor Handout 17

Problem Set 6 Solutions

This problem set is due **at 9:00pm** on **Wednesday, April 4, 2012**.

Problem 6-1. Amortization in Balanced Binary Trees

In a binary search tree, we say that a node x is a descendant of y if x is equal to y, or if the parent of x is a descendant of y. The size of the subtree rooted at y is equal to the number of descendants of y. Most self-balancing binary search trees (such as AVL trees and red-black trees) keep the tree fairly balanced, thereby ensuring that the tree has depth logarithmic in the number of elements. In this problem, we will develop an alternative self-balancing binary search tree using amortization.

Suppose that we have a binary tree augmented with subtree sizes. Define the *imbalance* of a node x to be the difference between the number of descendants of the left child of x and the number of descendants of the right child of x. More formally, we define:

$$imbalance(x) = |x.left.size - x.right.size|$$

If either child of x is null, then the imbalance is equal to the size of the non-null child. If both children are null, then the imbalance is equal to 0.

(a) Give an algorithm that takes as input a pointer to some node x in a BST and rebalances the subtree rooted at x. Your algorithm should run in $\Theta(\ell)$ time, where ℓ is the number of descendants of x. Your algorithm should not move any node in the tree that is not a descendant of x, but should reorganize or reconstruct the subtree rooted at x so that each node in the resulting subtree has imbalance at most 1.

Solution: We will first turn the elements in the tree rooted at x into a sorted array in $\Theta(\ell)$ time, and then convert the sorted array into a new BST with imbalances of 0 or 1 on all its nodes, again taking $\Theta(\ell)$ time.

To do the first step, we can simply do an in-order traversal of the tree rooted at x, obtaining a sorted array A with the elements, this clearly takes $\Theta(\ell)$ time.

We then construct a BST on the elements of A as follows: make a root node whose key is equal to the element at index $\lfloor A.length/2 \rfloor$ (since we have an array, this can be accessed in $O(1)$ time), and recursively construct a BST on the elements of the left half of the array and a BST on the elements of the right half of the array. Then make the roots of those two BSTs equal to the left and right children of our root node. Since there are ℓ elements total, we do a total work of $\Theta(\ell)$.

Once this recursive construction is done, remove the subtree rooted at x and replace it with the tree we just constructed.

(b) Suppose that you pick some constant c and modify your code for INSERT and DELETE so that you rebalance a node x as soon as $imbalance(x) > c$. Show that if you start with a tree of size n such that every node has imbalance 0, there exists a sequence of INSERT and DELETE operations such that the total runtime is not $O((\#\text{ of ops}) \cdot \lg n)$. Explain why this implies that this rebalancing scheme is bad.

Solution: If you perform $c+1$ INSERT operations, always adding an element strictly larger than all elements in the tree, each element you add will be added as a descendant of the right child of the root. As a result, after $c+1$ such operations, the root will have imbalance $c+1$, at which point the root will be rebalanced. (Even if other rebalances occurred beforehand, none of them would affect the root — the rebalance operation only affects descendants, and the root is only a descendant of itself.) Rebalancing the root takes $\Theta(n)$ time. Hence we have $c+1$ operations which take time $\Omega(n)$. This means that the amortized time per operation is $\Omega(n/(c+1))$. Because c is constant, this means that the amortized time for each operation is $\Omega(n)$, which is not $O(\lg n)$.

Clearly we need a more nuanced view of what it means for a node to be highly unbalanced. To that end, we say that the *imbalance ratio* of a node x is equal to the imbalance of x divided by the size of x. More formally:

$$imb\text{-}ratio(x) = \frac{imbalance(x)}{x.size} = \frac{|x.left.size - x.right.size|}{x.size}$$

(c) Show that any binary search tree in which every node has imbalance ratio at most $1/2$ has depth $O(\lg n)$.

Solution: We want to know the relationship between the size of some node x and the size of its children. WLOG, assume that the left child has more descendants. Because of the definition of subtree sizes, we have $x.size = 1 + x.left.size + x.right.size$. Hence, we can do the following math:

$$\begin{aligned}
imb\text{-}ratio(x) &\leq \frac{1}{2} \\
\frac{|x.left.size - x.right.size|}{x.size} &\leq \frac{1}{2} \\
|x.left.size - x.right.size| &\leq \frac{x.size}{2} \\
|x.left.size - (x.size - x.left.size - 1)| &\leq \tfrac{1}{2} \cdot x.size \\
|2 \cdot x.left.size - x.size + 1| &\leq \tfrac{1}{2} \cdot x.size \\
2 \cdot x.left.size - x.size + 1 &\leq \tfrac{1}{2} \cdot x.size \\
2 \cdot x.left.size &\leq \tfrac{3}{2} \cdot x.size - 1 \\
x.left.size &\leq \tfrac{3}{4} \cdot x.size - \tfrac{1}{2}
\end{aligned}$$

So the larger child cannot have more than $3/4$ of the descendants of the parent. This means that at depth d, the number of descendants is $\leq \left(\frac{3}{4}\right)^d \cdot root.size = (\frac{3}{4})^d \cdot n$. Each node in the tree has at least one descendant (itself), so we have the following equation:

$$\begin{aligned} 1 \leq \text{\# of descendants} &\leq (\tfrac{3}{4})^d \cdot n \\ \log_{\frac{4}{3}} 1 &\leq \log_{\frac{4}{3}} \left((\tfrac{3}{4})^d \cdot n\right) \\ 0 &\leq \log_{\frac{4}{3}}(\tfrac{3}{4})^d + \log_{\frac{4}{3}} n \\ d &\leq \log_{\frac{4}{3}} n \end{aligned}$$

So the maximum depth of a node is $O(\lg n)$.

Suppose that you modify your code for INSERT and DELETE so that you rebalance a node x as soon as $imb\text{-}ratio(x) > 1/2$. In the next two parts of this problem, you will show that under this rebalancing scheme the amortized runtime of INSERT and DELETE is $O(\lg n)$.

(d) Suppose that you have a BST of size n such that all nodes have imbalance at most 1. Use the accounting method to find the amortized runtime of a sequence of k INSERT and DELETE operations on this BST.

Hint: A rebalance of x occurs when $imb\text{-}ratio(x) > 1/2$. We can also write this condition as $imbalance(x) > x.size/2$. How many descendants of x must be inserted or deleted in order to cause the imbalance to become that high? And what is the cost of rebalancing x?

Solution: Whenever we insert or delete a node, we walk the BST to find it and then donate $\$c$ to every ancestor of the inserted or deleted node, where c is a constant to be selected later. How much does this add to the cost of inserting or deleting a node? It depends on the depth of the node modified. The depth of the tree is at most $O(\lg n)$, because the size of the tree is initially n and the number of operations k is polynomial in n. So the total amount of credit added to the data structure in a single insertion or deletion is at most $O(c \lg n) = O(\lg n)$. Since insertions and deletions already cost $O(\lg n)$, that basically just increases the constant factor behind the cost.

Now consider what happens when we choose to rebalance some node x. We will pay for the cost of rebalancing using the credit stored at x. How much credit is there at x when x is rebalanced? After the node x was previously rebalanced (thereby resetting the amount of credit to zero), it had an imbalance of at most 1. Because x is being rebalanced again, we know that $imbalance(x) > x.size/2$. Let's consider how the sequence of operations that has occurred since the previous rebalancing affects both the credit stored at x, and the imbalance of x.

Any insertion or deletion that does not affect any descendant of x will not affect the imbalance of x or the credit stored at x. An insertion into the subtree rooted at x could

cause $imbalance(x)$ to increase by at most 1, and always adds $\$c$ of credit to the node x. A deletion from the subtree rooted at x could cause $imbalance(x)$ to increase by at most 1, and always adds $\$c$ of credit to the node x. So $imbalance(x)$ increases by at most 1 for every operation that affects a descendant of x, and every time that occurs, the credit stored at x increases by $\$c$.

We know that at the time that x is rebalanced, $imbalance(x) > x.size/2$. So the number of insertions and deletions affecting a descendant of x since the last rebalance was at least $x.size/2 - 1$. Therefore, the credit stored up at x must be $c(x.size/2 - 1)$. The actual cost of rebalancing x is proportional to $x.size$, so if we set c sufficiently high then we can pay for the total cost of rebalancing with the credit that we have stored up. This means that the amortized cost of rebalancing is 0, so the amortized cost of insertion and deletion is still $O(\lg n)$ even when we consider rebalancing.

(e) Suppose that you have a BST of size n such that all nodes have imbalance at most 1. Use the potential method to find the amortized runtime of a sequence of k INSERT and DELETE operations on this BST.

Hint: Remember that when using the potential method, the final potential should always be at least as large as the initial potential, no matter what the sequence of operations is. Keeping in mind that an imbalance of 1 is, in some sense, as balanced as possible, is there a way to define your potential function so that the initial potential is 0, but the potential function is always non-negative?

Solution: We define the potential for a particular node x to be

$$\phi(x) = c \cdot \max\{0, imbalance(x) - 1\}.$$

We then define the potential for the entire BST D to be $\Phi(D) = \sum_x \phi(x)$. First, consider how insertion or deletion affects the potential of the data structure, without considering rebalancing. When we insert or delete a node, it can possibly increase the imbalance of all of its ancestors in the tree. In the worst case, it will increase the imbalance of all of its ancestors. Because we start with n nodes and the number of operations k is polynomial in n, the number of ancestors of any node is $O(\lg n)$. So the total potential increase caused by insertion or deletion is at most $O(\lg n)$. This means that the amortized cost of insertion or deletion is still $O(\lg n)$, barring rebalancing.

What about rebalancing the node x? Rebalancing the node x means ripping out the subtree rooted at x and replacing it with a subtree that is as balanced as possible: all imbalances equal to 0 or 1. This means that the potential contributed by the resulting subtree is equal to 0. Any node that is not a descendant of x will not experience any change in potential — the local structural changes do not affect the number of descendants for any node not in the subtree. So the rebalancing will cause the total potential to decrease by an amount equal to the sum of the potentials of x and all of its descendants before the rebalancing occured.

To trigger a rebalancing at node x, the imbalance at x must have been greater than $x.size/2$. So the potential at x before the rebalancing must have been at least $c \cdot (x.size/2 - 1)$. Before the rebalancing, the descendants of x may have had potential as well, but all potentials are non-negative, so the total potential contributed by x and its descendants must have been at least $c \cdot (x.size/2 - 1)$. Therefore, rebalancing causes the total potential to decrease by at least $c \cdot x.size/2 - c$. The actual cost of rebalancing the subtree rooted at x is $a \cdot x.size$, where a is some constant. Hence, the amortized cost of rebalancing is

$$\begin{aligned}\text{amortized cost} &= \text{real cost} + \Delta\Phi \\ &= a \cdot x.size - c \cdot x.size/2 + c \\ &= (a - c/2) \cdot x.size + c\end{aligned}$$

So if we set $c \geq 2a$, we ensure that the amortized cost of rebalancing x is constant. As previously discussed, inserting or deleting only affects the imbalance and size of $O(\lg n)$ nodes, and so it cannot trigger more than $O(\lg n)$ rebalance operations, each of which costs $\leq c$. As a result, the cost incurred by rebalancing after inserting or deleting is $O(c \lg n) = O(\lg n)$. So the total amortized cost of inserting or deleting is $O(\lg n) + O(\lg n) = O(\lg n)$.

Note that unlike regular AVL or red-black trees, this method of rebalancing BSTs does not require any rotations. As a result, this type of BST (which is a variant on what is known as a "weight-balanced tree") can be easier to implement in certain contexts. However, AVL trees and red-black trees achieve *worst-case* $O(\lg n)$ time, while weight-balanced BSTs only achieve *amortized* $O(\lg n)$ time. So there is something of a trade-off between runtime and simplicity.

Problem 6-2. House for Sale

You would like to sell your house, but there are very few buyers due to the housing market crash. Assume that at any given time there is at most one buyer interested in your house. You know with certainty that the range of offer from any buyer will be from 1 to M dollars. After getting an offer from each buyer, you make a decision to either take it or leave it. If you leave it, the buyer will buy some other house, so you will lose the offer forever. Note that the list of buyers is finite, and you do NOT know how many there will be. If you reject all of the offers, you will end up selling your house to the bank for 1 dollar. You want to maximize the return from selling your house. To this end, your knowledge of competitive analysis from 6.046 comes in very handy. Note that a clairvoyant algorithm will always pick the best offer.

(a) Give a deterministic algorithm that is $\frac{1}{\sqrt{M}}$-competitive. In other words, for any input sequence where x is the best offer, your algorithm should always choose an offer of at least $\frac{x}{\sqrt{M}}$ dollars.

Solution: The algorithm we use is as follows: as soon as we get an offer that is at least $\sqrt{M}$, we accept the offer. If no such offer occurs, we sell it to the bank for $\$1$. To analyze the algorithm, we consider two cases:

- **Case 1:** The best offer has value at least $\sqrt{M}$. Then there exists at least one bid with value $\geq \sqrt{M}$, so our algorithm will accept it. How will that bid compare to the best offer? Well, in this case, the best offer could be as high as M. The offer we accept will be at least $\sqrt{M}$, so:

$$\text{offer we accept} \geq \sqrt{M} = \frac{M}{\sqrt{M}} \geq \frac{\text{best offer}}{\sqrt{M}}$$

  Hence, the competitive ratio of our algorithm under these conditions is $\frac{1}{\sqrt{M}}$.
- **Case 2:** The best offer has value strictly less than $\sqrt{M}$. Then we'll never accept any offers, and we'll end up selling to the bank for $\$1$. Hence, we get the following equation:

$$\text{offer we accept} = \$1 = \frac{\sqrt{M}}{\sqrt{M}} > \frac{\text{best offer}}{\sqrt{M}}$$

  So once again, the competitive ratio of our algorithm is $\frac{1}{\sqrt{M}}$.

Hence, we have an algorithm whose competitive ratio is $\frac{1}{\sqrt{M}}$ no matter what the input is.

(b) Devise a randomized algorithm that is expected $\frac{1}{2 \lg M}$-competitive. In other words, for any input sequence where x is the best offer, the expected value of the offer selected by your algorithm should be at least $\frac{x}{2 \lg M}$ dollars.
Hint: You may want to consider powers of 2 from 1 to M.

Solution: Let $\ell = \lfloor \lg M \rfloor$. At the start of the algorithm, we pick a value i uniformly at random from $\{1, \ldots, \ell\}$. We then accept any bid that is greater than $\frac{M}{2^i}$.
We wish to discover how this algorithm does when compared with the clairvoyant algorithm. Suppose that the algorithm receives as input a sequence of bids such that the best bid is x. Let k be the integer satisfying the equation

$$\frac{M}{2^k} < x \leq \frac{M}{2^{k-1}}$$

Because x lies in the range $[1, M]$, k must belong to the set $\{1, \ldots, \ell+1\}$. We consider two cases:

- **Case 1:** $k \in \{1, \ldots, \ell\}$. Then we have probability $1/\ell$ of picking $i = k$. If we pick $i = k$, we know that $x > \frac{M}{2^k}$, so there is at least one bid that is greater than $\frac{M}{2^k}$. Therefore, we know that:

$$(\text{offer we accept if } i = k) > \frac{M}{2^k} = \frac{1}{2} \cdot \frac{M}{2^{k-1}} \geq \frac{1}{2} \cdot x$$

What does this mean for the expected value of our bid? We get the following:

$$\begin{aligned}\mathbb{E}[\text{offer we accept}] &= \sum_{j=1}^{\ell} P(\text{picking } i = j) \cdot (\text{offer we accept if } i = j) \\ &\geq P(\text{picking } i = k) \cdot (\text{offer we accept if } i = k) \\ &\geq \frac{1}{\ell} \cdot \frac{x}{2} = \frac{x}{2\lfloor \lg M \rfloor} \\ &\geq \frac{x}{2 \lg M}\end{aligned}$$

This means that our expected competitive ratio in this case is $\geq \frac{1}{2 \lg M}$.

- **Case 2:** $k = \ell + 1$. Then we have no chance of picking $i = k$. This means that we might have to take the bank's offer of \$1. How large is x compared to \$1? Because $\ell = \lfloor \lg M \rfloor$, we know that $2^\ell \leq M < 2^{\ell+1}$. This means that we have:

$$x \leq \frac{M}{2^{k-1}} = \frac{M}{2^\ell} = 2 \cdot \frac{M}{2^{\ell+1}} < 2 \cdot \frac{M}{M} = 2$$

Hence, we know that the offer we accepted from the bank is at least $\frac{1}{2}$ of the value of the best bid. So clearly in this case, our competitive ratio is $\geq \frac{1}{2} > \frac{1}{2 \lg M}$.

In either case, we get an expected competitive ratio which is $\geq \frac{1}{2 \lg M}$. This means that our algorithm as a whole achieves expected competitive ratio $\frac{1}{2 \lg M}$.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012