A good scapegoat is nearly as welcome as a solution to the problem.
— Anonymous

Let's play.
— El Mariachi [Antonio Banderas], *Desperado* (1992)

6 Dynamic Binary Search Trees (February 8)

6.1 Definitions

I'll assume that everyone is already familiar with the standard terminology for binary search trees—node, search key, edge, root, internal node, leaf, right child, left child, parent, descendant, sibling, ancestor, subtree, preorder, postorder, inorder, etc.—as well as the standard algorithms for searching for a node, inserting a node, or deleting a node. Otherwise, see Chapter 13 of CLR.

For this lecture, we will adopt the convention that every internal node has *exactly* two children, and only the *internal* nodes actually store search keys. Thus, in practice, we can represent leaves as null pointers.

Recall that the *depth* $d(v)$ of a node v is its distance from the root, and its *height* $h(v)$ is the distance to the farthest leaf in its subtree. The height (or depth) of the tree is just the height of the root. The *size* $|v|$ of v is the number of nodes in its subtree. The size of the whole tree is just the total number of nodes, which I'll usually denote by n.

A tree with height h has at most 2^h leaves, so the minimum height of an n-leaf binary tree is $\lceil \lg n \rceil$. In the worst case, the time required for a search, insertion, or deletion to the height of the tree, so in general we would like keep the height as close to $\lg n$ as possible. The best we can possibly do is to have a *perfectly balanced* tree, in which each subtree has as close to half the leaves as possible, and both subtrees are perfectly balanced. The height of a perfectly balanced tree is $\lceil \lg n \rceil$, so the worst-case search time is $O(\log n)$. However, even if we started with a perfectly balanced tree, a malicious sequence of insertions and/or deletions could make the tree arbitrarily unbalanced, driving the search time up to $\Theta(n)$.

To avoid this problem, we need to periodically modify the tree to maintain 'balance'. There are several methods for doing this, and depending on the method we use, the search tree is given a different name. Examples include AVL trees, red-black trees, height-balanced trees, weight-balanced trees, bounded-balance trees, path-balanced trees, B-trees, B*-trees, and (a, b)-trees. Some of these trees support searches, insertions, and deletions, in $O(\log n)$ *worst-case* time, others in $O(\log n)$ *amortized* time.

In this lecture, I'll discuss two binary search tree data structures with good *amortized* performance. The first is the *scapegoat tree*, discovered by Arne Andersson in 1989 and independently by Igal Galperin and Ron Rivest in 1993.[1] The second is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1985.[2]

[1] A. Andersson. General balanced trees. *J. Algorithms* 30:1-28, 1999. I. Galperin and R. L. Rivest. Scapegoat trees. *Proc. SODA 1993*, pp. 165–174, 1993. The claim of independence is Andersson's; the conference version of his paper appeared in 1989. The two papers actually describe very slightly different rebalancing algorithms. The algorithm I'm using here is closer to Andersson's, but my analysis is closer to Galperin and Rivest's.

[2] D. D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM* 32:652–686, 1985.

6.2 Deletions: Global Rebuilding

First let's consider the simple case where we start with a perfectly-balanced tree, and we only want to perform searches and deletions. To get good search and delete times, we will use a technique called *global rebuilding*. When we get a delete request, we locate and mark the node to be deleted, *but we don't actually delete it*. This requires a simple modification to our search algorithm—we still use marked nodes to guide searches, but if we search for a marked node, the search routine says it isn't there. This keeps the tree more or less balanced, but now the search time is no longer a function of the amount of data currently stored in the tree. To remedy this, we also keep track of how many nodes have been marked, and then apply the following rule:

> **Global Rebuilding Rule.** *As soon as half the nodes in the tree have been marked, rebuild a new perfectly balanced tree containing only the unmarked nodes.*[3]

With this rule in place, a search takes $O(\log n)$ time in the worst case, where n is the number of unmarked nodes. Specifically, since the tree has at most n marked nodes, or $2n$ nodes altogether, we need to examine at most $\lg n + 1$ keys. There are several methods for rebuilding the tree in $O(n)$ time, where n is the size of the new tree. (Homework!) So a single deletion can cost $\Theta(n)$ time in the worst case, but only if we have to rebuild; most deletions take only $O(\log n)$ time.

We spend $O(n)$ time rebuilding, but only after $\Omega(n)$ deletions, so the *amortized* cost of rebuilding the tree is $O(1)$ per deletion. (Here I'm using a simple version of the 'taxation method'. For each deletion, we charge a \$1 tax; after n deletions, we've collected \$$n$, which is just enough to pay for rebalancing the tree containing the remaining n nodes.) Since we also have to find and mark the node being 'deleted', the total amortized time for a deletion is $O(\log n)$.

6.3 Insertions: Partial Rebuilding

Now suppose we only want to support searches and insertions. We can't 'not really insert' new nodes into the tree, since that would make them unavailable to the search algorithm.[4] So instead, we'll use another method called *partial rebuilding*. We will insert new nodes normally, but whenever a *subtree* becomes unbalanced enough, we rebuild it. The definition of 'unbalanced enough' depends on an arbitrary constant $\alpha > 1$.

Each node v will now also store its height $h(v)$ and the size of its subtree $|v|$. We now modify our insertion algorithm with the following rule:

> **Partial Rebuilding Rule.** *After we insert a node, walk back up the tree updating the heights and sizes of the nodes on the search path. If we encounter a node v where $h(v) > \alpha \cdot \lg|v|$, rebuild its subtree into a perfectly balanced tree (in $O(|v|)$ time).*

[3] Alternately: When the number of unmarked nodes is one less than an exact power of two, rebuild the tree. This rule ensures that the tree is always *exactly* balanced.

[4] Actually, there is another dynamic data structure technique, first described by Jon Bentley and James Saxe in 1980, that *doesn't* really insert the new node! Instead, their algorithm puts the new node into a brand new data structure all by itself. Then as long as there are two trees of exactly the same size, those two trees are merged into a new tree. So this is exactly like a binary counter—instead of one n-node tree, you have a collection of 2^i-nodes trees of distinct sizes. The amortized cost of inserting a new element is $O(\log n)$. Unfortunately, we have to look through up to $\lg n$ trees to find anything, so the search time goes up to $O(\log^2 n)$. [J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1:301-358, 1980.]

If we always follow this rule, then after an insertion, the height of the tree is at most $\alpha \cdot \lg n$. Thus, since α is a constant, the worst-case search time is $O(\log n)$. In the worst case, insertions require $\Theta(n)$ time—we might have to rebuild the entire tree. However, the *amortized* time for each insertion is again only $O(\log n)$. Not surprisingly, the proof is a little bit more complicated than for deletions.

Define the *imbalance* $I(v)$ of a node v to be one less than the absolute difference between the sizes of its two subtrees, or zero, whichever is larger:

$$I(v) = \max\left\{\big||\text{left}(v)| - |\text{right}(v)|\big| - 1,\ 0\right\}$$

A simple induction proof implies that $I(v) = 0$ for every node v in a perfectly balanced tree. So immediately after we rebuild the subtree of v, we have $I(v) = 0$. On the other hand, each insertion into the subtree of v increments either $|\text{left}(v)|$ or $|\text{right}(v)|$, so $I(v)$ changes by at most 1.

The whole analysis boils down to the following lemma.

Lemma 1. *Just before we rebuild v's subtree, $I(v) = \Omega(|v|)$.*

Before we prove this, let's first look at what it implies. If $I(v) = \Omega(|v|)$, then $\Omega(|v|)$ keys have been inserted in the v's subtree since the last time it was rebuilt from scratch. On the other hand, rebuilding the subtree requires $O(|v|)$ time. Thus, if we amortize the rebuilding cost across all the insertions since the last rebuilding, v is charged *constant* time for each insertion into its subtree. Since each new key is inserted into at most $\alpha \cdot \lg n = O(\log n)$ subtrees, the total amortized cost of an insertion is $O(\log n)$.

Proof: Since we're about to rebuild the subtree at v, we must have $h(v) > \alpha \cdot \lg |v|$. Without loss of generality, suppose that the node we just inserted went into v's left subtree. Either we just rebuilt this subtree or we didn't have to, so we also have $h(\text{left}(v)) \le \alpha \cdot \lg |\text{left}(v)|$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg |v| \ < \ h(v) \ \le \ h(\text{left}(v)) + 1 \ \le \ \alpha \cdot \lg |\text{left}(v)| + 1.$$

After some algebra, this simplifies to $|\text{left}(v)| > |v|/2^{1/\alpha}$. Combining this with the identity $|v| = |\text{left}(v)| + |\text{right}(v)| + 1$ and doing some more algebra gives us the inequality

$$|\text{right}(v)| < \left(1 - 1/2^{1/\alpha}\right)|v| - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$I(v) \ \ge \ |\text{left}(v)| - |\text{right}(v)| - 1 \ > \ \left(2/2^{1/\alpha} - 1\right)|v|$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. □

6.4 Scapegoat Trees

Finally, to handle both insertions and deletions efficiently, *scapegoat trees* use both of the previous techniques. We use partial rebuilding to re-balance the tree after insertions, and global rebuilding to re-balance the tree after deletions. Each search takes $O(\log n)$ time in the worst case, and the