Everything was balanced before the computers went off line. Try and adjust something, and you unbalance something else. Try and adjust that, you unbalance two more and before you know what's happened, the ship is out of control.

— Blake, *Blake's 7*, "Breakdown" (March 6, 1978)

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

— Anonymous

Let's play.

— El Mariachi [Antonio Banderas], *Desperado* (1992)

8 Dynamic Binary Search Trees (February 8)

8.1 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 12 of CLRS.

For this lecture, we will consider only *full* binary trees—where every internal node has *exactly* two children—where only the *internal* nodes actually store search keys. In practice, we can represent the leaves with 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, treaps, randomized binary search trees, skip lists,[1] and jumplists.[2] Some of these trees support searches, insertions, and deletions, in $O(\log n)$ *worst-case* time, others in $O(\log n)$ *amortized* time, still others in $O(\log n)$ *expected* 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

[1] Yeah, yeah. Skip lists aren't really binary search trees. Whatever you say, Mr. Picky.

[2] These are essentially randomized variants of the Phobian binary search trees you saw in the first midterm! [H. Brönnimann, F. Cazals, and M. Durand. Randomized jumplists: A jump-and-walk dictionary data structure. Manuscript, 2002. http://photon.poly.edu/~hbr/publi/jumplist.html.] So now you know who to blame.

Igal Galperin and Ron Rivest in 1993.[3] The second is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1985.[4]

8.2 Lazy 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.*[5]

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 $\boxed{O(\log n)}$.

8.3 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.[6] 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:

[3] 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.

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

[5] 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.

[6] 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.] See also Homework 3.

> **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).*

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:

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

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 $\boxed{O(\log n)}$.

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. □

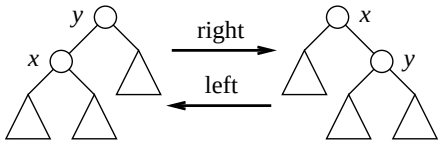
8.4 Scapegoat Trees

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 amortized time for any insertion or deletion is also $O(\log n)$. There are a few small technical details left (which I won't describe), but no new ideas are required.

Once we've done the analysis, we can actually simplify the data structure. It's not hard to prove that at most one subtree (the *scapegoat*) is rebuilt during any insertion. Less obviously, we can even get the same amortized time bounds (except for a small constant factor) if we only maintain the three integers in addition to the actual tree: the size of the entire tree, the height of the entire tree, and the number of marked nodes. Whenever an insertion causes the tree to become unbalanced, we can compute the sizes of all the subtrees on the search path, starting at the new leaf and stopping at the scapegoat, in time proportional to the size of the scapegoat subtree. Since we need that much time to re-balance the scapegoat subtree, this computation increases the running time by only a small constant factor! Thus, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space.

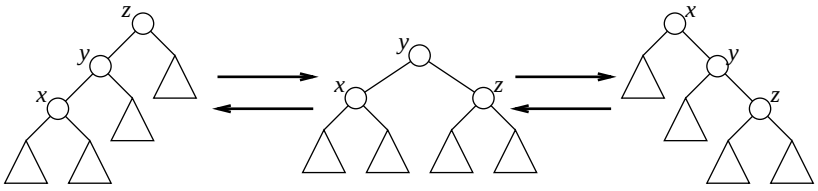
8.5 Rotations, Double Rotations, and Splaying

Another method for maintaining balance in binary search trees is by adjusting the shape of the tree locally, using an operation called a *rotation*. A rotation at a node x decreases its depth by one and increases its parent's depth by one. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

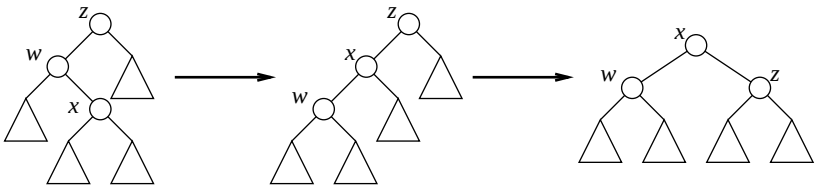


A right rotation at x and a left rotation at y are inverses.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called *roller-coaster* and *zig-zag*. A roller-coaster at a node x consists of a rotation at x's parent followed by a rotation at x, both in the same direction. A zig-zag at x consists of two rotations at x, in opposite directions. Each double rotation decreases the depth of x by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation. Either type of double rotation can be performed in constant time.

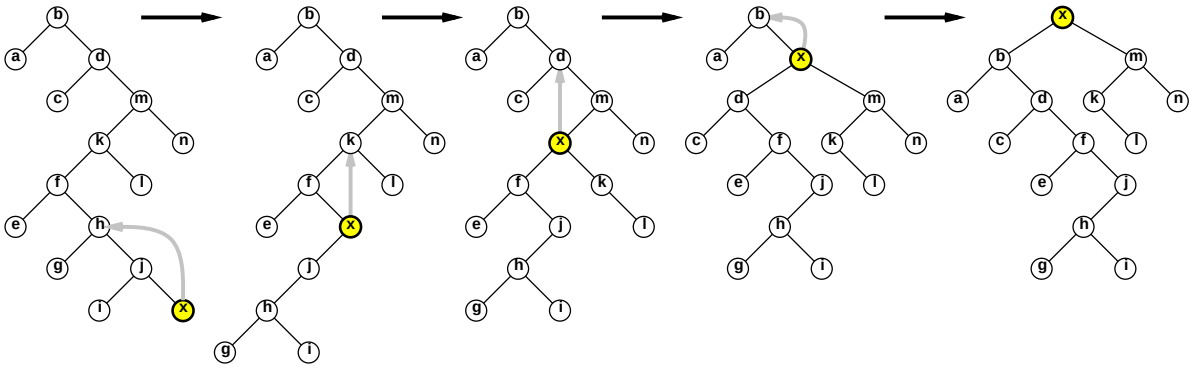


A right roller-coaster at x and a left roller-coaster at z.



A zig-zag at x. The symmetric case is not shown.

Finally, a *splay* operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end. Splaying a node v requires time proportional to $d(v)$. (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

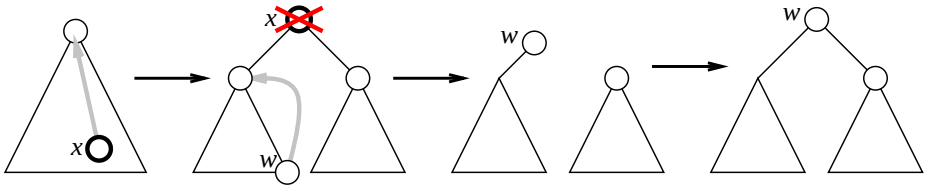


Splaying a node. Irrelevant subtrees are omitted for clarity.

8.6 Splay Trees

A *splay tree* is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation. In more detail:

- **Search:** Find the node containing the key using the usual algorithm, or its predecessor or successor if the key is not present. Splay whichever node was found.
- **Insert:** Insert a new node using the usual algorithm, then splay that node.
- **Delete:** Find the node x to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys less than x, the other with keys bigger than x. Find the node w in the left subtree with the largest key (*i.e.*, the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.



Deleting a node in a splay tree.

Each search, insertion, or deletion consists of a constant number of operations of the form *walk down to a node, and then splay it up to the root.* Since the walk down is clearly cheaper than the splay up, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

Believe it or not, the easiest way to do this uses the potential method. The *rank* of any node v is defined as $r(v) = \lfloor \lg |v| \rfloor$. In particular, if v is the root of the tree, then $r(v) = \lfloor \lg n \rfloor$. We define the *potential* of a splay tree to be the sum of the ranks of all its nodes:

$$\boxed{\Phi = \sum_v r(v) = \sum_v \lfloor \lg |v| \rfloor}$$

The amortized analysis of splay trees boils down to the following lemma. Here, $r(v)$ denotes the rank of v before a (single or double) rotation, and $r'(v)$ denotes its rank afterwards.

Lemma 2. *The amortized cost of a single rotation at v is at most $1 + 3r'(v) - 3r(v)$, and the amortized cost of a double rotation at v is at most $3r'(v) - 3r(v)$.*

Proving this lemma requires an easy but boring case analysis of the different types of rotations, which takes up almost a page in Sleator and Tarjan's original paper. (They call it the 'Access Lemma'.) I won't repeat it here.

By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node v is at most $1 + 3r'(v) - 3r(v)$, where $r'(v)$ is the rank of v after the splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, v is the root, so $r'(v) = \lfloor \lg n \rfloor$, which means that the amortized cost of a splay is at most $3 \lg n - 1 = O(\log n)$. Thus, every insertion, deletion, or search in a splay tree takes $\boxed{O(\log n) \text{ amortized time}}$, which is optimal.

Actually, splay trees are optimal in a much stronger sense. If $p(v)$ denotes the *probability* of searching for a node v, then the amortized search time for v is $O(\log(1/p(v)))$. If every node is equally likely, we have $p(v) = 1/n$ so $O(\log(1/p(v))) = O(\log n)$, as before. Even if the nodes aren't equally likely, though, the *optimal static* binary search tree for this set of *weighted* nodes has a search time of $\Theta(\log(1/p(v)))$. Splay trees give us this optimal amortized weighted search time with *no* change to the data structure or the splaying algorithm.