5 Dynamic Binary Search Trees (February 4)

5.1 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 [CLR, Chapter 13].

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 leaves in its subtree; the weight of the whole tree is just the total number of leaves, 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 as 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]

5.2 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

[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 approach I'm using is closer to Andersson's.

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

says it isn't there. This keeps the tree "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.*

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, we need to examine between $\lg n$ and $2 \lg n$ keys. There are several methods for rebuilding a 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 linear time rebuilding, but only after a linear number of deletions, so the *amortized* cost of rebuilding the tree is $O(1)$ per deletion. Since we also have to find and mark the node begin "deleted", the total amortized time for a deletion is $O(\log n)$.

5.3 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. 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 $h(v) > \alpha \lg|v|$, 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 $\alpha \lg n$. 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.

If the two children of v have different heights, let $\hat{v}$ denote its *tall* child and let $\check{v}$ denote its *short* child. If both children have the same height, arbitrarily call one $\hat{v}$ and the other $\check{v}$. For all nodes v, we have $h(\hat{v}) \geq h(\check{v})$ and $h(v) = h(\hat{v}) + 1$.

Define the *imbalance* $I(v)$ of a node v as follows:

$$\boxed{I(v) = \max\{0, |\hat{v}| - |\check{v}| - 1\}}$$

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 $|\hat{v}|$ or $|\check{v}|$, so either $I(v)$ stays the same or it increases by 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 the lemma, 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 $\alpha \lg n = O(\log n)$ subtrees, the total amortized cost of an insertion is $O(\log n)$.

Proof: We just have to grind through the definitions.

$$
\begin{array}{rll}
 & h(v) > \alpha \lg|v| & \text{We have to rebuild } v. \\
 & h(\hat{v}) < \alpha \lg|v| & \text{We just rebuilt } \hat{v}\text{, or we didn't have to.} \\
 & h(v) = h(\hat{v}) + 1 & \text{This is the recursive definition of } h(v). \\
\Longrightarrow & \alpha \lg|v| < \alpha \lg|\hat{v}| + 1 & \\
\Longrightarrow & |\hat{v}| > |v|/2^{1/\alpha} & \\
 & |v| = |\check{v}| + |\hat{v}| & \text{This is the recursive definition of } |v|. \\
 & I(v) \geq |\hat{v}| - |\check{v}| - 1 & \text{Definition of imbalance.} \\
 & = 2|\hat{v}| - |v| - 1 & \\
 & \geq 2|v|/2^{1/\alpha} - |v| - 1 & \\
 & = \left(2^{1-1/\alpha} - 1\right)|v| - 1 &
\end{array}
$$

Since α is a constant and $\alpha > 1$, the factor in parentheses is a positive constant. □

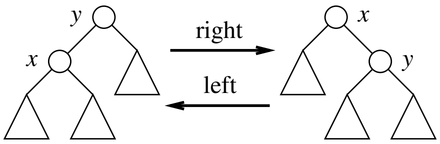
5.4 Scapegoat Trees

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 $O(\log n)$. I won't describe the details, but no new ideas are required.

Now that we've done the analysis, we can make a few simplifications to the algorithm. First of all, it's not hard to prove that at most one subtree (the *scapegoat*) is rebuilt during any insertion. Second, we can 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 tree, the height of the tree, and the number of marked nodes. Unlike most other kinds of balanced trees, scapegoat trees only require $O(1)$ extra space.

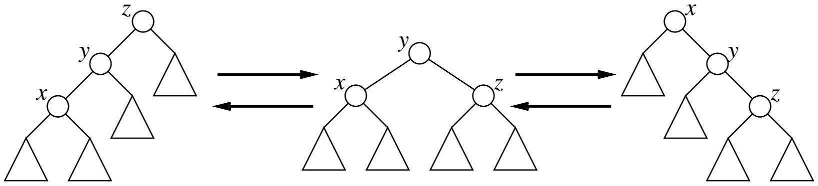
5.5 Rotations, Double Rotations, and Splaying

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.

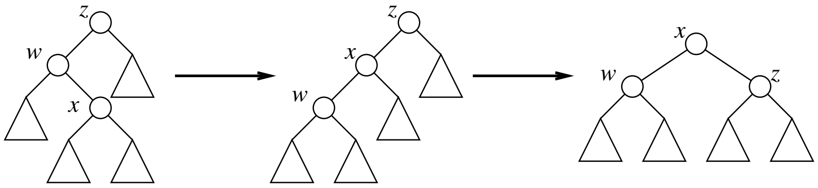


A right rotation at x and a left rotation at y are inverses.

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*. Roller-coaster rotations consist of two rotations in the same direction; zig-zags consist of two rotations in opposite directions. A double rotation at a node x decreases the depth of x by two, increases the depth of its parent by one, 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.

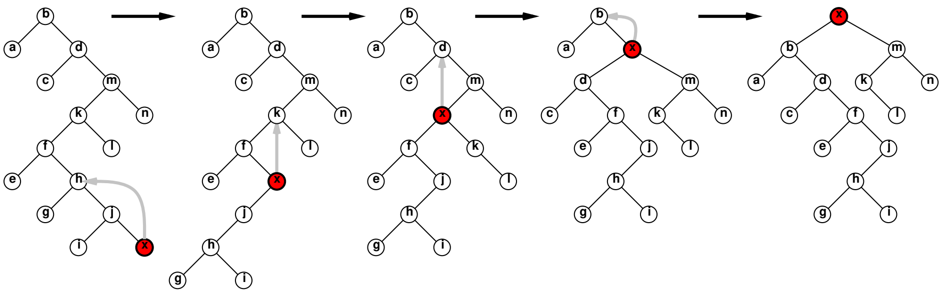


A right roller-coaster at x and a left roller-coaster at z.



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!)



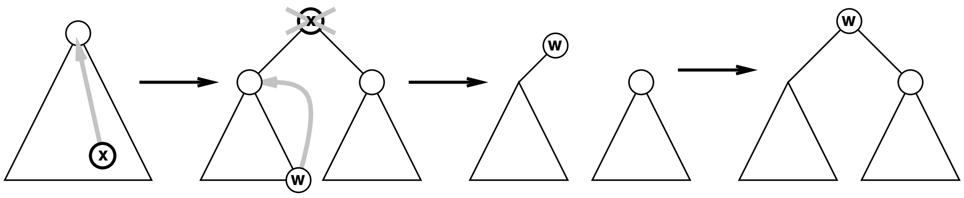
Splaying a node. Irrelevant subtrees are omitted for clarity.

5.6 Splay Trees

A *splay tree* is a binary search tree that is kept more or less balanced by splaying. Intuitively, after a node is accessed, it is splayed to the root. 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 neef to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

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:

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

Our amortized analysis 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 a fairly 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 intermeidate 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 $O(\log n)$ 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.