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)

CAPTAIN: TAKE OFF EVERY 'ZIG'!!
CAPTAIN: YOU KNOW WHAT YOU DOING.
CAPTAIN: MOVE 'ZIG'.
CAPTAIN: FOR GREAT JUSTICE.

— *Zero Wing* (1992)

2 Scapegoat and Splay Trees

2.1 Definitions

★★★ Move intro paragraphs to earlier treap notes, or maybe to new appendix on basic data structures (arrays, stacks, queues, heaps, binary search trees).

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, consult your favorite data structures textbook.

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* of a node is its distance from the root, and its *height* 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* of a node is the number of nodes in its subtree. The size n of the whole tree is just the total number of nodes.

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. Some of these trees support searches, insertions,

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

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 three binary search tree data structures with good *amortized* performance. The first two are variants of *lazy* balanced trees: *lazy weight-balanced trees*, developed by Mark Overmars* in the early 1980s [14], and *scapegoat trees*, first described by Arne Andersson* in 1989 [1, 2] and independently rediscovered[2] by Igal Galperin* and Ron Rivest in 1993 [11]. The third structure is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1981 [19, 16].

2.2 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 can 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 $\boldsymbol{O(\log n)}$.

2.3 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 *height*(v) and *size*(v). We now modify our insertion algorithm with the following rule:

---

[2]The claim of independence is Andersson's [2]. 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.

[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]Well, we could use the Bentley-Saxe* logarithmic method [3], but that would raise the query time to $O(\log^2 n)$.

> **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 $height(v) > \alpha \cdot \lg(size(v))$, rebuild its subtree into a perfectly balanced tree (in $O(size(v))$ 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 the absolute difference between the sizes of its two subtrees:

$$Imbal(v) := |size(left(v)) - size(right(v))|$$

A simple induction proof implies that $Imbal(v) \leq 1$ for every node v in a perfectly balanced tree. In particular, immediately after we rebuild the subtree of v, we have $Imbal(v) \leq 1$. On the other hand, each insertion into the subtree of v increments either $size(left(v))$ or $size(right(v))$, so $Imbal(v)$ changes by at most 1.

The whole analysis boils down to the following lemma.

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

Before we prove this lemma, let's first look at what it implies. If $Imbal(v) = \Omega(size(v))$, then $\Omega(size(v))$ 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 $O(size(v))$ time. Thus, if we amortize the rebuilding cost across all the insertions since the previous rebuild, 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 $\boldsymbol{O(\log n)}$.

Proof: Since we're about to rebuild the subtree at v, we must have $height(v) > \alpha \cdot \lg size(v)$. 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 $height(left(v)) \leq \alpha \cdot \lg size(left(v))$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg size(v) \ < \ height(v) \ \leq \ height(left(v)) + 1 \ \leq \ \alpha \cdot \lg size(left(v)) + 1.$$

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

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

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

$$Imbal(v) \ \geq \ size(left(v)) - size(right(v)) - 1 \ > \ \left(2/2^{1/\alpha} - 1\right) size(v)$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. $\square$

2.4 Scapegoat (Lazy Height-Balanced) 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.

2.5 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.

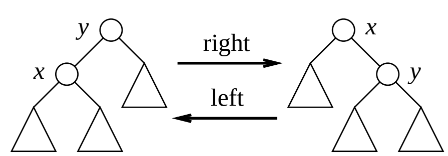

Figure 1. A right rotation at x and a left rotation at y are inverses.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called *zig-zag* and *roller-coaster*. A zig-zag at x consists of two rotations at x, in opposite directions. 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. 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.

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 *depth*(v). (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

2.6 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:

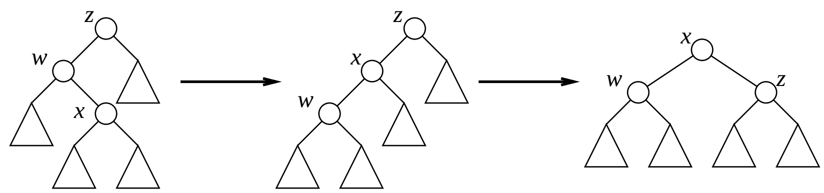

Figure 2. A zig-zag at x. The symmetric case is not shown.

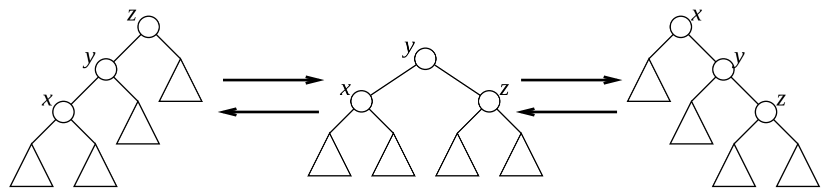

Figure 3. A right roller-coaster at x and a left roller-coaster at z.

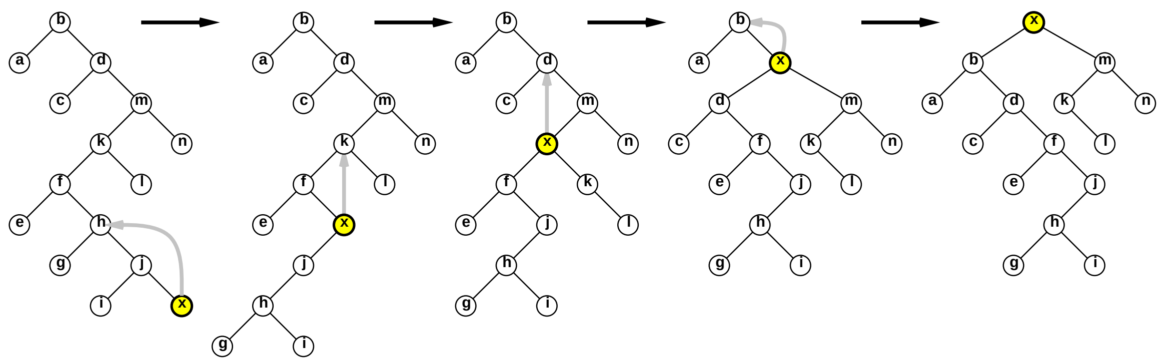

Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

- **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 (the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.

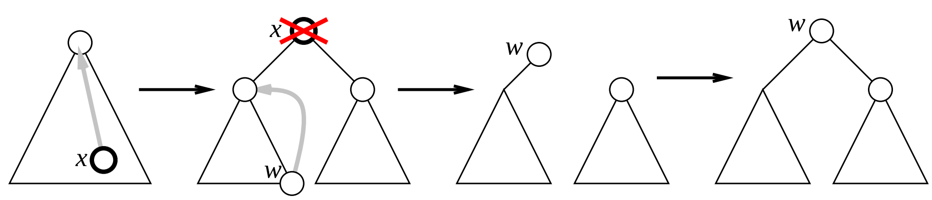

Figure 5. Deleting a node in a splay tree.

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. We define the *rank* of a node v to be $\lfloor \lg size(v) \rfloor$, and the *potential* of a splay tree to be the sum of the ranks of its nodes:

$$\Phi := \sum_v rank(v) = \sum_v \lfloor \lg size(v) \rfloor$$

It's not hard to observe that a perfectly balanced binary tree has potential $\Theta(n)$, and a linear chain of nodes (a perfectly *unbalanced* tree) has potential $\Theta(n \log n)$. The amortized cost of an operation is now *defined* to be the number of rotations plus the drop in potential.

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

The Access Lemma. *The amortized cost of a single rotation at any node v is at most $1 + 3\,rank'(v) - 3\,rank(v)$, and the amortized cost of a double rotation at any node v is at most $3\,rank'(v) - 3\,rank(v)$.*

Proving this lemma is a straightforward but tedious case analysis of the different types of rotations. For the sake of completeness, I'll give a proof (of a generalized version) in the next section.

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 $1 + 3\,rank'(v) - 3\,rank(v)$, where $rank'(v)$ is the rank of v after the entire splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, node v is the root! Thus, $rank'(v) = \lfloor \lg n \rfloor$, which implies that the amortized cost of a splay is at most $3 \lg n - 1 = O(\log n)$.

We conclude that every insertion, deletion, or search in a splay tree takes $O(\log n)$ amortized time.

*2.7 Other Optimality Properties

In fact, splay trees are optimal in several other senses. Some of these optimality properties follow easily from the following generalization of the Access Lemma.

Let's arbitrarily assign each node v a non-negative real *weight* $w(v)$. These weights are not actually stored in the splay tree, nor do they affect the splay algorithm in any way; they are only used to help with the analysis. We then redefine the *size* $s(v)$ of a node v to be the sum of the weights of the descendants of v, including v itself:

$$s(v) := w(v) + s(right(v)) + s(left(v)).$$

If $w(v) = 1$ for every node v, then the size of a node is just the number of nodes in its subtree, as in the previous section. As before, we define the *rank* of any node v to be $r(v) = \lg s(v)$, and the *potential* of a splay tree to be the sum of the ranks of all its nodes:

$$\Phi = \sum_v r(v) = \sum_v \lg s(v)$$

In the following lemma, $r(v)$ denotes the rank of v before a (single or double) rotation, and $r'(v)$ denotes its rank afterwards.

The Generalized Access Lemma. *For **any** assignment of non-negative weights to the nodes, the amortized cost of a single rotation at any node x is at most $1+3r'(x)-3r(x)$, and the amortized cost of a double rotation at any node v is at most $3r'(x)-3r(x)$.*

Proof: First consider a single rotation, as shown in Figure 1.

$$\begin{aligned}
1+\Phi'-\Phi &= 1+r'(x)+r'(y)-r(x)-r(y) && [\text{only } x \text{ and } y \text{ change rank}]\\
&\le 1+r'(x)-r(x) && [r'(y)\le r(y)]\\
&\le 1+3r'(x)-3r(x) && [r'(x)\ge r(x)]
\end{aligned}$$

Now consider a zig-zag, as shown in Figure 2. Only w, x, and z change rank.

$$\begin{aligned}
&2+\Phi'-\Phi\\
&\quad= 2+r'(w)+r'(x)+r'(z)-r(w)-r(x)-r(z) && [\text{only } w,x,z \text{ change rank}]\\
&\quad= 2+r'(w)+r'(z)-r(w)-r(x) && [r'(x)=r(z)]\\
&\quad\le 2+r'(w)+r'(z)-2r(x) && [r(x)\le r(w)]\\
&\quad= 2+(r'(w)-r'(x))+(r'(z)-r'(x))+2(r'(x)-r(x))\\
&\quad= 2+\lg\frac{s'(w)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+2(r'(x)-r(x))\\
&\quad= 2+\lg\frac{s'(w)+s'(z)}{s'(x)}+2(r'(x)-r(x)) && [\lg \text{ is concave}]\\
&\quad\le 2+2\lg\frac{s'(x)/2}{s'(x)}+2(r'(x)-r(x)) && [s'(w)+s'(z)\le s'(x)]\\
&\quad= 2(r'(x)-r(x))\\
&\quad\le 3(r'(x)-r(x)) && [r'(x)\ge r(x)]
\end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x, y, and z change rank.

$$\begin{aligned}
&2+\Phi'-\Phi\\
&\quad= 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) && [\text{only } x,y,z \text{ change rank}]\\
&\quad\le 2+r'(x)+r'(z)-r(x)-r(y) && [r'(y)\le r(z)]\\
&\quad\le 2+r'(x)+r'(z)-2r(x) && [r(x)\ge r(y)]\\
&\quad= 2+(r(x)-r'(x))+(r'(z)-r'(x))+3(r'(x)-r(x))\\
&\quad= 2+\lg\frac{s(x)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+3(r'(x)-r(x))\\
&\quad\le 2+2\lg\frac{s(x)+s'(z)}{s'(x)}+3(r'(x)-r(x)) && [\lg \text{ is concave}]\\
&\quad\le 2+2\lg\frac{s'(x)/2}{s'(x)}+3(r'(x)-r(x)) && [s(x)+s'(z)\le s'(x)]\\
&\quad= 3(r'(x)-r(x))
\end{aligned}$$

This completes the proof.[5] $\square$

[5] This proof is essentially taken verbatim from the Sleator and Tarjan's original paper [16]. Another proof technique, which may be more accessible or intuitive for some students, involves maintaining $\lfloor \lg \mathit{size}(v) \rfloor$ tokens on each node v and arguing about the changes in token distribution caused by each single or double rotation. But I haven't yet internalized this approach enough to include it here; it isn't more accessible or intuitive to *me*!

Observe that this argument works for *arbitrary* non-negative vertex weights. By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node x is at most $1 + 3r(root) - 3r(x)$. (The intermediate ranks cancel out in a nice telescoping sum.)

This analysis has several immediate corollaries. The first corollary is that the amortized search time in a splay tree is within a constant factor of the search time in the best possible *static* binary search tree. Thus, if some nodes are accessed more often than others, the standard splay algorithm *automatically* keeps those more frequent nodes closer to the root, at least most of the time.

Static Optimality Theorem. *Suppose each node x is accessed at least $t(x)$ times, and let $T = \sum_x t(x)$. The amortized cost of accessing x is $O(\log T - \log t(x))$.*

Proof: Set $w(x) = t(x)$ for each node x. □

For any nodes x and z, let $dist(x, z)$ denote the *rank distance* between x and y, that is, the number of nodes y such that $key(x) \le key(y) \le key(z)$ or $key(x) \ge key(y) \ge key(z)$. In particular, $dist(x, x) = 1$ for all x.

Static Finger Theorem. *For any fixed node f ('the finger'), the amortized cost of accessing x is $O(\lg dist(f, x))$.*

Proof: Set $w(x) = 1/dist(x, f)^2$ for each node x. Then $s(root) \le \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \ge \lg w(x) = -2 \lg dist(f, x)$. □

Here are a few more interesting properties of splay trees, which I'll state without proof.[6] The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

Working Set Theorem [16]. *The amortized cost of accessing node x is $O(\log D)$, where D is the number of distinct items accessed since the last time x was accessed. (For the first access to x, we set $D = n$.)*

Scanning Theorem [18]. *Splaying all nodes in a splay tree in order, starting from any initial tree, requires $O(n)$ total rotations.*

Dynamic Finger Theorem [7, 6]. *Immediately after accessing node y, the amortized cost of accessing node x is $O(\lg dist(x, y))$.*

*2.8 Splay Tree Conjectures

Splay trees are conjectured to have many interesting properties in addition to the optimality properties that have been proved; I'll describe just a few of the more important ones.

The *Deque Conjecture* [18] considers the cost of dynamically maintaining two fingers l and r, starting on the left and right ends of the tree. Suppose at each step, we can move one of these two fingers either one step left or one step right; in other words, we are using the splay tree as a doubly-ended queue. Sundar* proved that the total cost of m deque operations on an n-node splay tree is $O((m + n)\alpha(m + n))$ [17]. More recently, Pettie later improved this bound to $O(m\alpha^*(n))$ [15]. The Deque Conjecture states that the total cost is actually $O(m + n)$.

[6] This list and the following section are taken almost directly from Erik Demaine's lecture notes [5].

The *Traversal Conjecture* [16] states that accessing the nodes in a splay tree, in the order specified by a *preorder* traversal of any other binary tree with the same keys, takes $O(n)$ time. This is generalization of the Scanning Theorem.

The *Unified Conjecture* [13] states that the time to access node x is $O(\lg \min_y(D(y)+d(x,y)))$, where $D(y)$ is the number of *distinct* nodes accessed since the last time y was accessed. This would immediately imply both the Dynamic Finger Theorem, which is about spatial locality, and the Working Set Theorem, which is about temporal locality. Two other structures are known that satisfy the unified bound [4, 13].

Finally, the most important conjecture about splay trees, and one of the most important open problems about data structures, is that they are *dynamically optimal* [16]. Specifically, the cost of any sequence of accesses to a splay tree is conjectured to be at most a constant factor more than the cost of the best possible dynamic binary search tree *that knows the entire access sequence in advance*. To make the rules concrete, we consider binary search trees that can undergo *arbitrary* rotations after a search; the cost of a search is the number of key comparisons plus the number of rotations. We do not require that the rotations be on or even near the search path. This is an extremely strong conjecture!

No dynamically optimal binary search tree is known, even in the offline setting. However, three very similar $O(\log\log n)$-competitive binary search trees were discovered in a short time period: *Tango trees* [9], *multisplay trees* [20], and *chain-splay trees* [12]. There is also a nice geometric formulation of dynamic binary search trees [8, 10] that offers significant hope for future progress.

References

[1] Arne Andersson*. Improving partial rebuilding by using simple balance criteria. *Proc. Workshop on Algorithms and Data Structures*, 393–402, 1989. Lecture Notes Comput. Sci. 382, Springer-Verlag.

[2] Arne Andersson. General balanced trees. *J. Algorithms* 30:1–28, 1999.

[3] Jon L. Bentley and James B. Saxe*. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980.

[4] Mihai Bădiou* and Erik D. Demaine. A simplified and dynamic unified structure. *Proc. 6th Latin American Sympos. Theoretical Informatics*, 466–473, 2004. Lecture Notes Comput. Sci. 2976, Springer-Verlag.

[5] Jeff Cohen* and Erik Demaine. 6.897: Advanced data structures (Spring 2005), Lecture 3, February 8 2005. ⟨http://theory.csail.mit.edu/classes/6.897/spring05/lec.html⟩.

[6] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM J. Comput.* 30(1):44–85, 2000.

[7] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM J. Comput.* 30(1):1–43, 2000.

[8] Erik D. Demaine, Dion Harmon*, John Iacono, Daniel Kane*, and Mihai Pătraşcu. The geometry of binary search trees. *Proc. 20th Ann. ACM-SIAM Symp. Discrete Algorithms*, 496–505, 2009.

[9] Erik D. Demaine, Dion Harmon*, John Iacono, and Mihai Pătraşcu**. Dynamic optimality—almost. *Proc. 45th Annu. IEEE Sympos. Foundations Comput. Sci.*, 484–490, 2004.

[10] Jonathan Derryberry*, Daniel Dominic Sleator, and Chengwen Chris Wang*. A lower bound framework for binary search trees with rotations. Tech. Rep. CMU-CS-05-187, Carnegie Mellon Univ., Nov. 2005. ⟨http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-187.pdf⟩.

[11] Igal Galperin* and Ronald R. Rivest. Scapegoat trees. *Proc. 4th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 165–174, 1993.

[12] George F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove $\log\log N$-competitiveness by splaying. *Inform. Proc. Lett.* 106(1):37–43, 2008.

[13] John Iacono*. Alternatives to splay trees with $O(\log n)$ worst-case access times. *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 516–522, 2001.

[14] Mark H. Overmars*. *The Design of Dynamic Data Structures*. Lecture Notes Comput. Sci. 156. Springer-Verlag, 1983.

[15] Seth Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms*, 1115–1124, 2008.

[16] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.

[17] Rajamani Sundar*. On the Deque conjecture for the splay algorithm. *Combinatorica* 12(1):95–124, 1992.

[18] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica* 5(5):367–378, 1985.

[19] Robert Endre Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983.

[20] Chengwen Chris Wang*, Jonathan Derryberry*, and Daniel Dominic Sleator. $O(\log\log n)$-competitive dynamic binary search trees. *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 374–383, 2006.

*Starred authors were graduate students at the time that the cited work was published. **Double-starred authors were undergraduates.

See my Algorithms lecture notes for exercises!