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)

10 Scapegoat and Splay Trees

10.1 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,

© Copyright 2024 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

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, 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*, discovered by Arne Andersson* in 1989 [1, 2] and independently[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].

10.2 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.*

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

10.3 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.[3] 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 $1 < \beta < 2$.

---

[1] Yes, skip lists *are too* binary search trees!

[2] The claim of independence is Andersson's [2]. The two papers actually describe very slightly different rebalancing conditions and algorithms. The algorithm I'm using here is closer to Andersson's, but my analysis is closer to Galperin and Rivest's.

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

In addition to its search key and pointers to its children, each node v also stores its height *v.height* and the size of its subtree *v.size*. 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 $v.height > \log_\beta(v.size)$, rebuild its subtree into a perfectly balanced tree (in $O(v.size)$ time).*

If we always follow this rule, then after an insertion, the height of the tree is at most $\log_\beta n = O(\log n)$, because β is a constant. Thus, the worst-case search time is $O(\log n)$. Insertions require $\Theta(n)$ time in the worst case, because 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 more complicated than for deletions.

Define the *imbalance* $Imbal(v)$ of a node v to be the absolute difference between the sizes of its two subtrees:

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

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 $v.left.size$ or $v.right.size$, 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, $Imbal(v) = \Omega(v.size)$.*

Before we prove this lemma, let's first look at what it implies. If $Imbal(v) = \Omega(v.size)$, then $\Omega(v.size)$ 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 $O(v.size)$ 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 $\log_\beta n = O(\log n)$ subtrees, the total amortized cost of an insertion is $\boldsymbol{O(\log n)}$.

Proof: Because we're about to rebuild the subtree at v, we must have $v.height > \log_\beta(v.size)$. 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; in either case, we have $v.left.height \leq \log_\beta(v.left.size)$. Combining these two inequalities with the recursive definition of height, we get

$$\log_\beta(v.size) \;<\; v.height \;\leq\; v.left.height + 1 \;\leq\; \log_\beta(v.left.size) + 1.$$

These inequalities imply $v.left.size > v.size/\beta$ and therefore

$$v.right.size < \big(1 - 1/\beta\big)\, v.size.$$

(In particular, we have $v.left.size > v.right.size$ because $\beta < 2$.) Finally, we combine these two inequalities using the definition of imbalance.

$$Imbal(v) \;=\; v.left.size - v.right.size \;>\; \big(2/\beta - 1\big)\, v.size$$

Because $\beta < 2$ is a constant, the factor in parentheses is a positive constant. □

10.4 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.

We can even simplify the data structure slightly. Suppose inserting a new node x makes the tree unbalanced (because the depth of x is greater than $\log_\beta n$). Let v be the *deepest* ancestor of x such that $v.height > \log_\beta(v.size)$; we call v the ***scapegoat*** for this insertion.[4] To keep the height of the tree under $\log_\beta(n)$, it suffices to rebalance only the subtree rooted at the scapegoat.

In fact, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space in addition to the tree itself; specifically, we only need to maintain the size of the entire tree and the number of marked nodes, along with a few local variables during each insertion, deletion, or query. During insertion, we can compute the depth of the newly inserted node x on the fly. If the depth of x is greater than $\log_\beta n$, we can find the scapegoat as follows.

```
FindScapegoat(x):
  v ← x
  height ← 0
  size ← 1
  while height ≤ log_β(size)
      if v = v.parent.left
          size ← 1 + size + Size(v.parent.right)
      else
          size ← 1 + Size(v.parent.left) + size
      v ← v.parent
      height ← height + 1
  return v
```

```
Size(v):
  if v = Null
      return 0
  else
      return 1 + Size(v.left) + Size(v.right)
```

The subroutine Size computes the size of the subtree rooted at v in constant time per vertex. Thus, the overall running time of FindScapegoat is proportional to the size of the scapegoat subtree. Since we need that much time to rebalance the scapegoat's subtree, this computation increases the running time by only a small constant factor!

10.5 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.

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

[4]Leviticus describes a Jewish purification ritual, practiced during Yom Kippur, in which one goat is sacrificed and another goat is sent off into the wilderness, carrying with it the sins of the community. William Tyndale coined the word "scapegoat" for the second goat in 1530, as he was writing the first English translation of Hebrew scripture, as a mistranslation of the Hebrew word "Azazel", which refers to either a demon believed to live in the desert or a particular set of cliffs from which the scapegoat was thrown. Following earlier Greek and Latin translations, Tyndale apparently misread "Azazel" as "ez ozel", meaning "goat who departs".

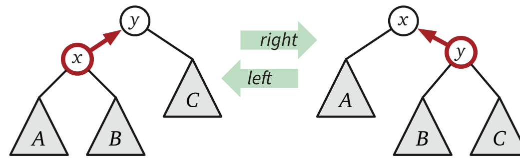


Figure 1. A right rotation at x and a left rotation at y are inverses.

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.

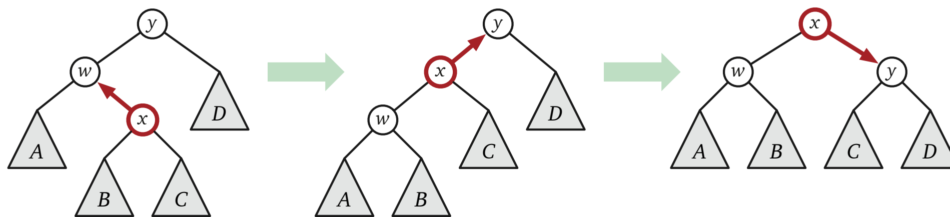


Figure 2. A zig-zag at x. The symmetric case is not shown.

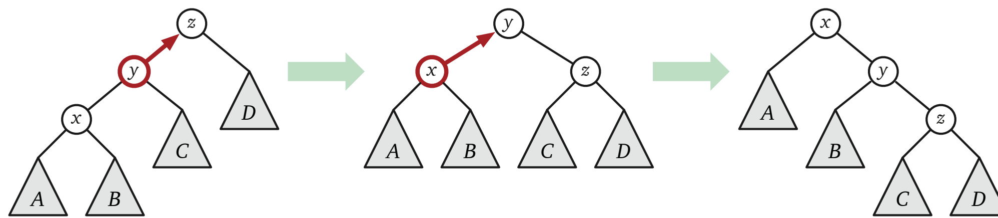


Figure 3. A right roller-coaster at x.

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!)

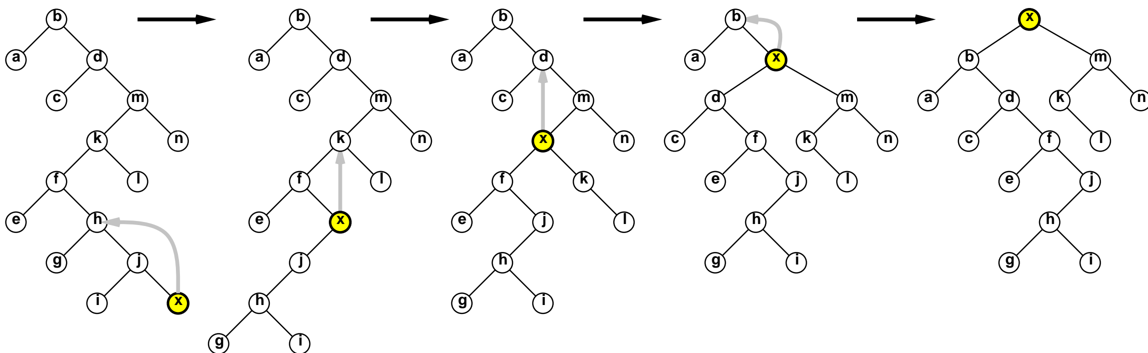


Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity. **REDRAW**

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

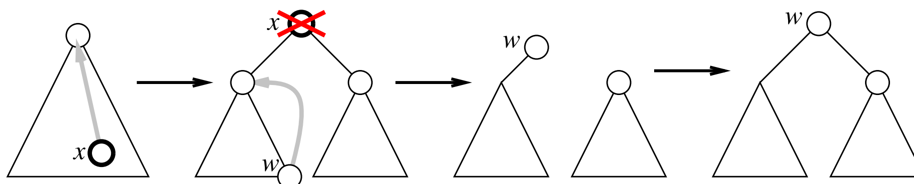


Figure 5. Deleting a node in a splay tree. **REDRAW**

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

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

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 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. Recall that the amortized cost is defined to be the number of rotations plus the drop in potential.

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, 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.

*10.7 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.

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

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

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

This completes the proof. [5] □

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.

[5] This proof is essentially taken verbatim from the original Sleator and Tarjan paper. Another proof technique, which may be more accessible, involves maintaining $\lfloor \lg s(v) \rfloor$ 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.

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

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 $O(\lg dist(x, y))$.*

*10.8 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 $O(m+n)$.

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 have been discovered in the last few years: *Tango trees* [9], *multisplay trees* [20], and *chain-splay trees* [12]. A recently-published geometric formulation of dynamic binary search trees [8, 10] also 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.

Exercises

0. An n-node binary tree is *perfectly balanced* if either $n \le 1$, or its two subtrees are perfectly balanced binary trees, each with at most $\lfloor n/2 \rfloor$ nodes. Prove that $I(v) \le 1$ for every node v of any perfectly balanced tree.

1. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean

nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

(a) Describe and analyze an algorithm to purge an arbitrary n-node dirty binary search tree in $O(n)$ time. (Such an algorithm is necessary for scapegoat trees to achieve $O(\log n)$ amortized insertion cost.)

*(b) Modify your algorithm so that is uses only $O(\log n)$ space, in addition to the tree itself. Don't forget to include the recursion stack in your space bound.

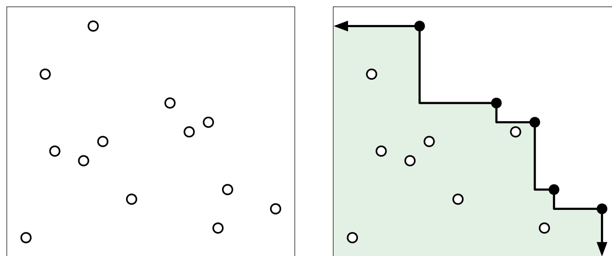
★(c) Modify your algorithm so that is uses only $O(1)$ additional space. In particular, your algorithm cannot call itself recursively at all.

2. Consider the following simpler alternative to splaying:

```
MoveToRoot(v):
  while v.parent ≠ Null
    rotate at v
```

Prove that the amortized cost of MoveToRoot in an n-node binary tree can be $\Omega(n)$. That is, prove that for any integer k, there is a sequence of k MoveToRoot operations that require $\Omega(kn)$ time to execute.

3. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.

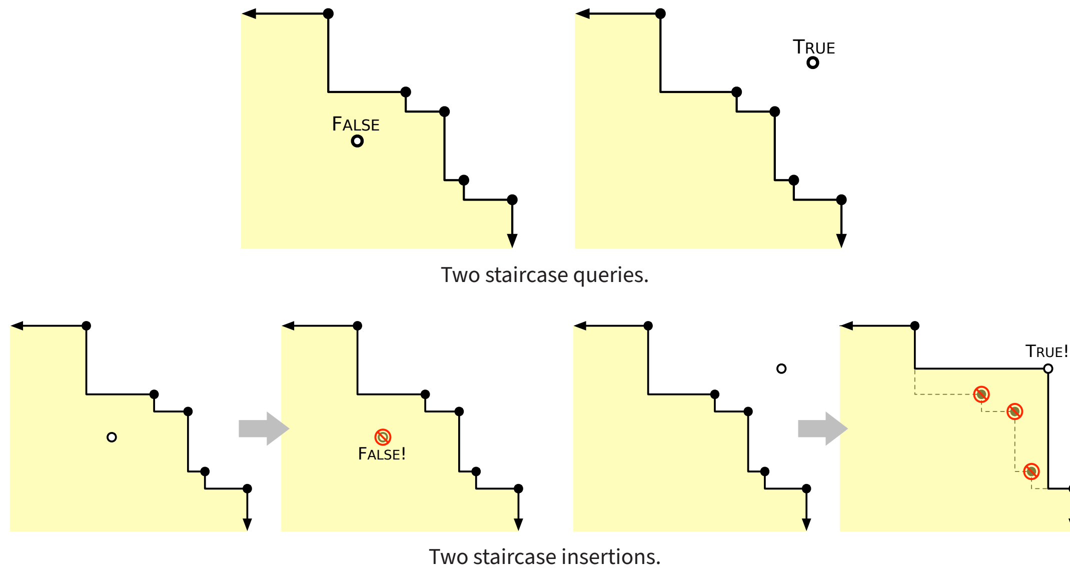


A set of points in the plane and its staircase (shaded).

(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm Above?(x, y) that returns True if the point (x, y) is above the staircase, or False otherwise. Your data structure should use $O(n)$ space, and your Above? algorithm should run in $O(\log n)$ time.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function Insert(x, y) that adds the point (x, y) to the underlying point set and returns True or False to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your Insert algorithm should run in $O(\log n)$ amortized time.



Two staircase queries.

Two staircase insertions.

4. Suppose we want to maintain a set of numbers, subject to the following operations:

   - INSERT(x): Add x to the set (if it isn't already there).
   - PRINT&DELETEBETWEEN(a, b): Print every element x in the range $a \leq x \leq b$, in increasing order, and delete those elements from the set.

   For example, if the current set is $\{1, 5, 3, 4, 8\}$, then

   - PRINT&DELETEBETWEEN($4, 6$) prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$;
   - PRINT&DELETEBETWEEN($6, 7$) prints nothing and does not change the set;
   - PRINT&DELETEBETWEEN($0, 10$) prints the sequence $1, 3, 4, 5, 8$ and deletes everything.

   (a) Suppose we store the set in our favorite balanced binary search tree, using the standard INSERT algorithm and the following algorithm for PRINT&DELETEBETWEEN:

   ```
   PRINT&DELETEBETWEEN(a, b):
     x ← SUCCESSOR(a)
     while x ≤ b
         print x
         DELETE(x)
         x ← SUCCESSOR(a)
   ```

   Here, SUCCESSOR(a) returns the smallest element greater than or equal to a (or ∞ if there is no such element), and DELETE is the standard deletion algorithm. Prove that the amortized time for INSERT and PRINT&DELETEBETWEEN is $O(\log N)$, where N is the *maximum* number of items that are ever stored in the tree.

   (b) Describe and analyze INSERT and PRINT&DELETEBETWEEN algorithms that run in $O(\log n)$ amortized time, where n is the *current* number of elements in the set.

   (c) What is the running time of your INSERT algorithm in the worst case?

(d) What is the running time of your PRINT&DELETEBETWEEN algorithm in the worst case?

5. Say that a binary search tree is *augmented* if every node v has a field $v.size$ storing the number of nodes in the subtree rooted at v.

   (a) Show that a rotation in an augmented binary tree can be performed in constant time.

   (b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in these notes are already augmented.)

   (c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

   (d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

6. Many applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Let T be an arbitrary binary tree. The secondary data structure at any node v stores exactly the same set of items as the subtree of T rooted at v. This secondary structure has size $O(size(v))$ and can be built in $O(size(v))$ time, where $size(v)$ denotes the number of descendants of v.

   The primary and secondary data structures are typically defined by different attributes of the data being stored. For example, to store a set of points in the plane, we could define the primary tree T in terms of the x-coordinates of the points, and define the secondary data structures in terms of their y-coordinate.

   Maintaining these secondary structures complicates algorithms for keeping the top-level search tree balanced. Specifically, performing a rotation at any node v in the primary tree now requires $O(size(v))$ time, because we have to rebuild one of the secondary structures (at the new child of v). When we insert a new item into T, we must also insert into one or more secondary data structures.

   (a) Overall, how much space does this data structure use *in the worst case*?

   (b) How much space does this structure use if the primary search tree is perfectly balanced?

   (c) Suppose the primary tree is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

   (d) Now suppose the primary tree T is a scapegoat tree. How long does it take to rebuild the subtree of T rooted at some node v, as a function of $size(v)$?

   (e) Suppose the primary tree and all secondary trees are scapegoat trees. What is the amortized cost of a single insertion?

   *(f) Finally, suppose the primary tree and every secondary tree is a treap. What is the worst-case *expected* time for a single insertion?

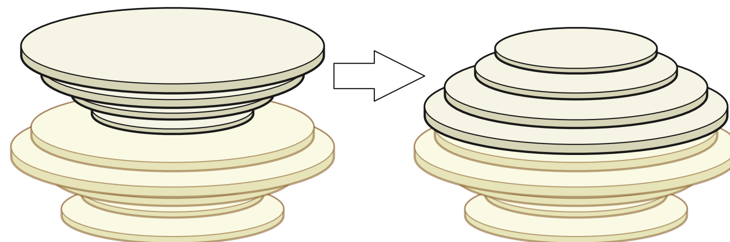
7. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
   - Concat(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
   - Split(S, k) removes the strings S (given by a pointer) from the data structure, adds the first k characters of S and the rest of S as two new strings in the data structure, and returns pointers to the two new strings.
   - Reverse(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
   - Lookup(S, k) returns the kth character in string S (given by a pointer), or Null if the length of the S is less than k.

   Describe and analyze a simple data structure that supports NewString and Reverse in $O(1)$ *worst-case* time, supports every other operation in $O(\log n)$ time (either worst-case, expected, or amortized), and uses $O(n)$ space, where n is the sum of the *current* string lengths. *[Hint: Why is this problem here?]*

8. After the Great Academic Meltdown of 2020, you get a job as a cook's assistant at Jumpin' Jack's Flapjack Stack Shack, which sells arbitrarily-large stacks of pancakes for just four bits (50 cents) each. Jumpin' Jack insists that any stack of pancakes given to one of his customers must be sorted, with smaller pancakes on top of larger pancakes. Also, whenever a pancake goes to a customer, at least the top side must not be burned.

   The cook provides you with a unsorted stack of n perfectly round pancakes, of n different sizes, possibly burned on one or both sides. Your task is to throw out the pancakes that are burned on both sides (and *only* those) and sort the remaining pancakes so that their burned sides (if any) face down. Your only tool is a spatula. You can insert the spatula under any pancake and then either *flip* or *discard* the stack of pancakes above the spatula.

   Flipping the top four pancakes. Again.

   More concretely, we can represent a stack of pancakes by a sequence of distinct integers between 1 and n, representing the sizes of the pancakes, with each number marked to indicate the burned side(s) of the corresponding pancake. For example, $\underline{1}\,\overline{4}\,3\,\overline{\underline{2}}$ represents a stack of four pancakes: a one-inch pancake burned on the bottom; a four-inch pancake burned on the top; an unburned three-inch pancake, and a two-inch pancake burned

on both sides. We store this sequence in a data structure that supports the following operations:

- Position(x): Return the position of integer x in the current sequence, or 0 if x is not in the sequence.
- Value(k): Return the kth integer in the current sequence, or 0 if the sequence has no kth element. Value is essentially the inverse of Position.
- TopBurned(k): Return True if and only if the top side of the kth pancake in the current sequence is burned.
- Flip(k): Reverse the order and the burn marks of the first k elements of the sequence.
- Discard(k): Discard the first k elements of the sequence.

(a) Describe an algorithm to filter and sort any stack of n burned pancakes using $O(n)$ of the operations listed above. Try to make the big-Oh constant small.

$$\underline{1}\,\overline{4}\,3\,\overline{\underline{2}} \xrightarrow{\text{Flip}(4)} \overline{\underline{2}}\,3\,\underline{4}\,\overline{1} \xrightarrow{\text{Discard}(1)} 3\,\underline{4}\,\overline{1} \xrightarrow{\text{Flip}(2)} \overline{4}\,3\,\overline{1} \xrightarrow{\text{Flip}(3)} \underline{1}\,3\,\underline{4}$$

(b) Describe a data structure that supports each of the operations listed above in $O(\log n)$ amortized time. Together with part (a), such a data structure gives us an algorithm to filter and sort any stack of n burned pancakes in $O(n \log n)$ time.

9. Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ be a sequence of m integers, each from the set $\{1, 2, \ldots, n\}$. We can visualize this sequence as a set of integer points in the plane, by interpreting each element x_i as the point (x_i, i). The resulting point set, which we can also call X, has exactly one point on each row of the $n \times m$ integer grid.

   (a) Let Y be an arbitrary set of integer points in the plane. Two points (x_1, y_1) and (x_2, y_2) in Y are *isolated* if (1) $x_1 \neq x_2$ and $y_1 \neq y_2$, and (2) there is no other point $(x, y) \in Y$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. If the set Y contains no isolated pairs of points, we call Y a *commune*.[7]

   Let X be an arbitrary set of points on the $n \times n$ integer grid with exactly one point per row. Show that there is a commune Y that contains X and consists of $O(n \log n)$ points.

   (b) Consider the following model of self-adjusting binary search trees. We interpret X as a sequence of accesses in a binary search tree. Let T_0 denote the initial tree. In the ith round, we traverse the path from the root to node x_i, and then *arbitrarily reconfigure* some subtree S_i of the current search tree T_{i-1} to obtain the next search tree T_i. The only restriction is that the subtree S_i must contain both x_i and the root of T_{i-1}. (For example, in a splay tree, S_i is the search path to x_i.) The *cost* of the ith access is the number of nodes in the subtree S_i.

   Prove that the minimum cost of executing an access sequence X in this model is at least the size of the smallest commune containing the corresponding point set X. *[Hint: Lowest common ancestor.]*

[7]Demaine *et al.* [8] refer to communes as *arborally satisfied sets*.

*(c) Suppose X is a *random* permutation of the integers $1, 2, \ldots, n$. Use the lower bound in part (b) to prove that the expected minimum cost of executing X is $\Omega(n \log n)$.

★(d) Describe a polynomial-time algorithm to compute (or even approximate up to constant factors) the smallest commune containing a given set X of integer points, with at most one point per row. Alternately, prove that the problem is NP-hard.

© Copyright 2024 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.