

# Splaying Preorders and Postorders\*

Caleb C. Levy<sup>†</sup>      Robert E. Tarjan<sup>‡</sup>

July 16, 2019

## Abstract

Let  $T$  be a binary search tree of  $n$  nodes with root  $r$ , left subtree  $L = \text{left}(r)$ , and right subtree  $R = \text{right}(r)$ . The *preorder* and *postorder* of  $T$  are defined as follows: the preorder and postorder of the empty tree is the empty sequence, and

$$\begin{aligned}\text{preorder}(T) &= (r) \oplus \text{preorder}(L) \oplus \text{preorder}(R) \\ \text{postorder}(T) &= \text{postorder}(L) \oplus \text{postorder}(R) \oplus (r),\end{aligned}$$

where  $\oplus$  denotes sequence concatenation.<sup>1</sup> We prove the following results about the behavior of splaying [21] preorders and postorders:

1. Inserting the nodes of  $\text{preorder}(T)$  into an empty tree via splaying costs  $O(n)$ . (Theorem 2.)
2. Inserting the nodes of  $\text{postorder}(T)$  into an empty tree via splaying costs  $O(n)$ . (Theorem 3.)
3. If  $T'$  has the same keys as  $T$  and  $T$  is *weight-balanced* [18] then splaying either  $\text{preorder}(T)$  or  $\text{postorder}(T)$  starting from  $T'$  costs  $O(n)$ . (Theorem 4.)

For 1 and 2, we use the fact that preorders and postorders are *pattern-avoiding*: i.e. they contain no subsequences that are order-isomorphic to  $(2, 3, 1)$  and  $(3, 1, 2)$ , respectively. Pattern-avoidance implies certain constraints on the manner in which items are inserted. We exploit this structure with a simple potential function that counts inserted nodes lying on access paths to uninserted nodes. Our methods can likely be extended to permutations that avoid more general patterns. The proof of 3 uses the fact that preorders and postorders of balanced search trees do not contain many large “jumps” in symmetric order, and exploits this fact using the dynamic finger theorem [6, 5].

Items 2 and 3 are both novel. Item 1 was originally proved by Chaudhuri and Höft [4]; our proof simplifies theirs. These results provide further evidence in favor of the elusive *dynamic optimality conjecture* [21].

---

\*Research at Princeton University partially supported by an innovation research grant from Princeton and a gift from Microsoft.

<sup>†</sup>Baskin School of Engineering, UC Santa Cruz; clevy@ucsc.edu.

<sup>‡</sup>Department of Computer Science, Princeton University, and Intertrust Technologies; ret@cs.princeton.edu.

<sup>1</sup>We will refer to any such sequence as a *preorder* or a *postorder*.

**Outline.** Section 1 discusses the mathematical preliminaries, historical background, and context for this investigation, and Section 2 samples some related work. Familiar readers may skip directly to the main results and their proofs, in Sections 3 and 4. Section 3 proves that inserting both preorders and postorders via splaying takes linear time. Section 4 establishes that splaying preorders and postorders of *balanced* search trees [18] takes linear time, regardless of starting tree. Section 5 provides our thoughts on how to analyze insertion splaying permutations that avoid more general patterns, particularly the class of “ $k$ -increasing” sequences [3].

## 1 Preliminaries

### Binary Search Trees

A *binary tree*  $T$  contains of a finite set of *nodes*, with one node designated to be the *root*. All nodes have a *left* and a *right child* pointer, each leading to a different node. Either or both children may be *missing*, and we denote a missing child by `null`. Every node in  $T$ , save for the root, has a single *parent* node of which it is a child. (The root has no parent.) The *size* of  $T$  is the number of nodes it contains, and is denoted  $|T|$ .

There is a unique path from  $\text{root}(T)$  to every other node  $x$  in  $T$ , called the *access path for  $x$  in  $T$* . If  $x$  is on the access path for  $y$  then we say  $x$  is an *ancestor* of  $y$ , and  $y$  is a *descendent* of  $x$ . We refer to the subtree comprising  $x$  and all of its descendants as the subtree rooted at  $x$ . Nodes thus have *left* and *right subtrees* rooted respectively at their left and right children. (Subtrees are *empty* for `null` children.) The *depth* of the node  $x$ , denoted  $d_T(x)$ , is the number of edges on its access path. Its *right-depth* is the number of right pointers followed, and its *left-depth* is the number of left pointers followed.

In a *binary search tree*, every node has a unique *key*, and the tree satisfies the *symmetric order* condition: every node’s key is greater than those in its left subtree and smaller than those in its right subtree. The binary search tree derives its name from how its structure enables finding keys. To find a key  $k$  initialize the current node to be the root. While the current node is not `null` and does not contain the given key, replace the current node by its left or right child depending on whether  $k$  is smaller or larger than the key in the current node, respectively. The search returns the last current node, which contains  $k$  if  $k$  is in the tree and otherwise `null`.

The *lowest common ancestor* of  $x$  and  $y$  in  $T$ , denoted  $\text{lca}_T(x, y)$ , is the deepest node shared by the access paths of both  $x$  and  $y$ . Since the root is a common ancestor of any pair of nodes in  $T$  and  $T$  is finite,  $\text{lca}_T(x, y)$  exists and is well defined. Furthermore  $\min\{x, y\} \leq \text{lca}_T(x, y) \leq \max\{x, y\}$ .

To *insert* a new key  $k$  into a binary search tree  $T$ , we first do a search for  $k$  in  $T$ . When the search reaches a missing node, we replace this node with a node containing the key  $k$ . (Inserting into an empty tree makes  $k$  the root key.)

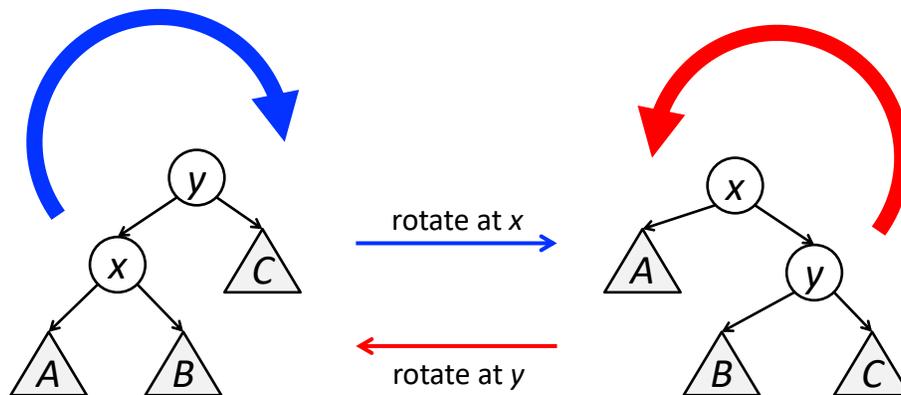


Figure 1: Rotation at node  $x$  with parent  $y$ , and reversing the effect by rotating at  $y$ .

### Rotation

Binary search trees are the canonical data structure for maintaining an ordered set of elements, and are building blocks in countless algorithms. Perhaps the most attractive feature of binary search trees is that the number of comparisons required to find an item in an  $n$ -node binary search tree is  $O(\log n)$ , provided that the tree is properly arranged, which is good in theory and practice. However, without exercising care when inserting nodes, a binary search tree can easily become unbalanced (for example when inserting  $1, 2, \dots, n$  in order), leading to search costs as high as  $\Omega(n)$ . Thus, binary search trees require some form of maintenance and restructuring for good performance.

We will employ a restructuring primitive called *rotation*. A rotation at left child  $x$  with parent  $y$  makes  $y$  the right child of  $x$  while preserving symmetric order. A rotation at a right child is symmetric, and rotation at the root is undefined. (See Figure 1). A rotation changes three child pointers in the tree.

Rotations were first employed in “balanced” search trees, which include AVL trees [1], Red-Black trees [10], weight-balanced trees [18], and more recently weak AVL trees [11]. These trees augment nodes with bits that provide rough information about how “balanced” each node’s subtree is. Whenever an item is inserted or deleted, rotations are performed to restore invariants on the balance bits that ensure all search paths have  $O(\log n)$  nodes. While balanced searched trees are not the focus of this work, they were progenitors for the main algorithm of interest.

### Splay

The Splay algorithm [21] eschews keeping track of balance information, replacing it with an intriguing notion: instead of adjusting the search tree only after insertion and deletion, Splay modifies the tree after *every* search.

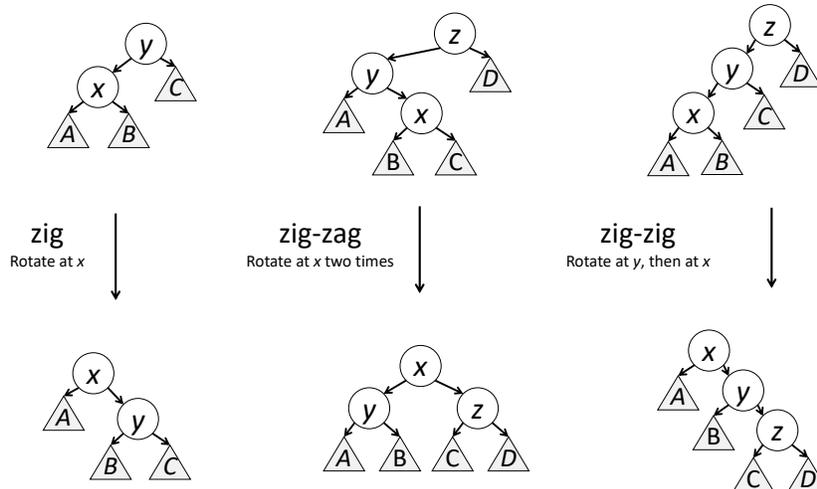


Figure 2: A splaying step at node  $x$ . Symmetric variants not shown. Triangles denote subtrees.

The algorithm begins with a binary search for a key in the tree. Let  $x$  be the node returned by this search. If  $x$  is not `null` then the algorithm repeatedly applies a “splay step” until  $x$  becomes the root. A splay step applies a certain series of rotations based on the relationship between  $x$ , its parent, and its grandparent, as follows. If  $x$  has no grandparent (i.e.  $x$ ’s parent is the root), then rotate at  $x$  (this case is always terminal). Otherwise, if  $x$  is a *left* child and its parent is a *right* child, or vice-versa, rotate at  $x$  twice. Otherwise, rotate at  $x$ ’s parent, and then rotate at  $x$ . Sleator and Tarjan [21] assigned the respective names *zig*, *zig-zag* and *zig-zig* to these three cases. The series of splay steps that bring  $x$  to the root are collectively called to as *splaying at x*, or simply *splaying x*. The three cases are depicted in Figure 2.

The *cost* of splaying a single item  $x$  in  $T$  is defined to be  $d_T(x) + 1$ .<sup>2</sup> If  $X = (x_1, \dots, x_m)$  is a sequence of requested keys in  $T$  then the cost of splaying  $X$  starting from  $T$  is defined as  $m + \sum_{i=1}^m d_{T_{i-1}}(x_i)$ , where  $T_0 = T$ , and for  $1 \leq i \leq m$ , we form  $T_i$  by splaying  $x_i$  in  $T_{i-1}$ . To perform *insertion splaying*, insert a key into the tree and then splay the newly created node. The cost of an insertion splay is the cost splaying the new node.

While splaying individual items can cost  $\Omega(n)$ , the *total* cost of splaying  $m$  requested items in a tree of size  $n > 0$  is  $O((m + n) \log n)$ . Hence, the worst case cost of a splay operation, *amortized* over all the requests, is the same as any balanced binary search tree. This is perhaps surprising for an algorithm that keeps no record of balance information.

What makes Splay truly remarkable is how it takes advantage of “latent

<sup>2</sup>We absorb the search cost into the rotations.

structure” in the request sequence, and provides more than simple “worst-case” guarantees. As just one example, if  $t_X(i)$  is the number of different items accessed before access  $i$  since the last access to item  $x_i$  (or since the beginning of the sequence if  $i$  is the first access to  $x_i$ ), then the cost to splay  $X$  starting from  $T$  is  $O(n \log n + \sum_{j=1}^m \log(t_X(j) + 1))$  [21].<sup>3</sup> (This is called the “working set” property.) Thus, Splay exploits “temporal locality” in the access pattern.

Splay simultaneously exploits “spatial” locality, as shown by the following theorem (originally conjectured in [21]) that we will use later on:

**Theorem 1** (Dynamic Finger [6, 5]). *Let the rank of  $x$  in  $T$ , denoted  $r_T(x)$ , be the number of nodes in  $T$  whose keys are less than or equal the key in  $x$ . The cost of splaying  $X = (x_1, \dots, x_m)$  starting from  $T$  is  $O(|T| + m + \text{DF}_T(X))$ , where  $\text{DF}_T(X) \equiv \sum_{i=2}^m \log_2(|r_T(x_i) - r_T(x_{i-1})| + 1)$ .*

In fact, the properties of Splay inspired the authors of [21] to speculate on a much stronger possibility: that Splay’s cost is always within a constant factor of the “optimal” way of executing the requests. Formally, an *execution*  $E$  for  $(X, T)$  comprises the following. Let  $T_0 = T$ , and for  $1 \leq i \leq m$ , we perform some number  $e_i \geq 0$  of rotations starting from  $T_{i-1}$  to form  $T_i$ , followed by a search for  $x_i$ . The *cost* of this execution is  $\sum_{i=1}^m (1 + e_i + d_{T_{i-1}}(x_i))$ . The *optimal cost*  $\text{OPT}(X, T) \equiv \min\{\text{cost}(E) \mid E \text{ executes } (X, T)\}$ . The following conjecture has spawned a great deal of related research (see §2):

**Conjecture 1** (Dynamic Optimality [21]).  $\text{cost}_{\text{splay}}(X, T) = O(\text{OPT}(X, T))$ .

The conjecture remains open. In fact, there is no sub-exponential time algorithm *whatsoever* that is known to compute, even to within a constant factor, the cost of an optimum binary search tree execution for an instance. There are several known lower bounds [7, 25], none known to be tight (though some conjectured to be).

## Pattern-Avoidance

For simplicity, we restrict subsequent discussion to *permutation* request sequences (i.e. no key is requested twice). By [3], any algorithm that achieves optimal cost on all permutations can be extended to an algorithm that is optimal for all request sequences.

An auxiliary question to determining if Splay (or any other algorithm) is dynamically optimal is: “what class(es) of permutations have optimum executions with ‘low’ cost?” This issue is not a mere curiosity, as almost every permutation of length  $n$  has optimal execution cost  $\Theta(n \log n)$  [14], a bound achieved by any balanced search tree. Thus, in the absence of insertions or deletions, adjusting the tree after every access only gives an advantage on a small subset of “structured” request sequences. In addition, these structured request sequences provide candidate counter-examples to dynamic optimality. In this work, we

<sup>3</sup>Note that  $O(\log n)$  amortized cost per splay is a corollary of this.

focus on certain *pattern-avoiding* permutations: those that do not contain any subsequences of a specified type. More formally:<sup>4</sup>

Two permutations  $\alpha = (a_1, \dots, a_n)$  and  $\beta = (b_1, \dots, b_n)$  of the same length are *order-isomorphic* if their entries have the same relative order, i.e.  $a_i < a_j \iff b_i < b_j$ . For example,  $(5, 8, 1)$  is order-isomorphic to  $(2, 3, 1)$ . A sequence  $\pi$  *avoids* a sequence  $\alpha$  (or is called  $\alpha$ -*avoiding*) if it has no subsequence that is order-isomorphic with  $\alpha$ . If  $\pi$  is  $\alpha$ -avoiding then all subsequences of  $\pi$  are  $\alpha$ -avoiding. We use  $\pi \setminus \alpha$  as shorthand for “an (arbitrary) permutation  $\pi$  that avoids  $\alpha$ .” Both preorders and postorders may be characterized as pattern-avoiding permutations:

**Lemma 1** (Lemma 1.4 from [13]). *For any permutation  $\pi$ :*

(a)  $\pi = \text{preorder}(T)$  for some binary search tree  $T \iff \pi$  avoids  $(2, 3, 1)$ .

(b)  $\pi = \text{postorder}(T)$  for some binary search tree  $T \iff \pi$  avoids  $(3, 1, 2)$ .

*Sketch.* For preorders, Kozma builds a bijection between binary search trees and  $(2, 3, 1)$ -avoiding sequences, and uses a simple argument by contradiction to show preorders avoid  $(2, 3, 1)$  [13]. The proof for postorders is a nearly symmetric variation of this argument.  $\square$

## 2 Related Work

The first result about Splay’s behavior on pattern-avoiding request sequences was the *sequential access* theorem [24]: the cost of splaying the nodes of  $T$  in order is  $O(|T|)$ . This is a special case of a corollary<sup>5</sup> of dynamic optimality:

**Conjecture 2** (Traversal [21]). *There exists  $c > 0$  for which the cost of splaying  $\text{preorder}(T)$  starting from  $T'$  is at most  $c|T|$  for all pairs of binary search trees  $T, T'$  with the same keys.*

Theorem 2 and [4] is another special case, when  $T = T'$ . In §3 we prove a new special case: when  $T$  is  $\alpha$ -weight balanced.

Interest in the behavior of binary search tree algorithms on “structured” request sequences was revived by Seth Pettie’s analysis of the performance of Splay-based deque data structures using Davenport-Schinzel sequences [20], and his later reproof of the sequential access theorem via the theory of forbidden submatrices [19].

This analysis was later adapted to and greatly extended for another binary search tree algorithm, colloquially known as “Greedy,” that was first proposed as an *off*-line algorithm independently by Lucas [16] and Munro [17]. Greedy is widely conjectured to be dynamically optimal, and is known to have many of

<sup>4</sup>The following definitions and theorems are taken from [13, Chapter 1.3], almost verbatim.

<sup>5</sup>A priori, the traversal conjecture follows from dynamic optimality *conditioned* on Splay being optimal with low “additive overhead.” The authors recently proved that this corollary is actually *unconditional* [15].

the same properties of Splay, including the working set [8] and dynamic finger [12] bounds.

Greedy was later recast as an on-line algorithm in a “geometric” view of binary search trees [7]. This geometric view of Greedy is especially amenable to forbidden submatrix analysis. In [3], Chalermsook et. al. show that Greedy has nearly-optimal run-time on a broad class of pattern-avoiding permutations. Moreover, they demonstrate that if Greedy is optimal on a certain class of “non-decomposable” permutations then it is dynamically optimal. Chalermsook et al.’s analysis was later simplified in [9].

### 3 Insertion Splaying Preorders and Postorders

If  $\pi = (p_1, \dots, p_n)$  is a permutation then the *insertion tree* for  $\pi$ , denoted  $\text{BST}(\pi)$ , is the binary search tree obtained by starting from an empty tree and inserting keys in order of their first appearance in  $\pi$ .

**Lemma 2.** *If  $x$  is a proper ancestor of  $y$  in  $\text{BST}(\pi)$  then  $x$  precedes  $y$  in  $\pi$ .*

*Proof.* Let  $\pi_{\prec y}$  denote the prefix of  $\pi$  containing the elements preceding  $y$ . By construction,  $y$  is inserted as a child of some node  $z$  in  $\text{BST}(\pi_{\prec y})$ . Every proper ancestor of  $y$  is an ancestor of  $z$ , thus  $x \in \text{BST}(\pi_{\prec y})$ . Hence,  $x$  precedes  $y$ .  $\square$

Insertion splaying  $\pi$  has the same cost as splaying  $\pi$  starting from  $\text{BST}(\pi)$ .<sup>6</sup> For the purposes of analysis we will assume that, initially, every node in  $\text{BST}(\pi)$  is marked as *untouched*. An insertion splay marks the node as *touched*, and then splays the node. The touched nodes form a connected subtree containing the root, called the *touched subtree*. The untouched nodes form subtrees each of which contains no touched node. Call an untouched node with a touched parent a *sub-root*. The subtrees rooted at sub-roots have identical structure in both the splayed tree and  $\text{BST}(\pi)$ . By Lemma 2, the next node to be touched is always a sub-root.

For  $1 \leq i \leq n$ , form  $T_i$  by touching and then splaying  $p_i$  in  $T_{i-1}$ , where  $T_0 = \text{BST}(\pi)$  starts with all nodes untouched. At any time we define the *potential* to be the twice the number of touched nodes that are ancestors of sub-roots, and we define  $\Phi_i$  to be the potential of  $T_i$ . The *amortized* cost of splaying  $p_i$  in  $T_{i-1}$  is defined as  $c_i = t_i + \Phi_i - \Phi_{i-1}$ , where  $t_i$  denotes the actual cost. By a standard telescoping sum argument, the cost of insertion splaying  $\pi$  is  $\sum_{i=1}^n t_i = \sum_{i=1}^n c_i + \Phi_0 - \Phi_n$  [23]. Since  $\Phi_0 = \Phi_n = 0$ , an upper bound on amortized cost provides an upper bound on the actual cost.

Pattern-avoidance provides certain information about both  $\text{BST}(\pi)$  and about which sub-root can be touched next. We exploit this information in the next two sections.

---

<sup>6</sup>This is because the manner in which Splay restructures the access path is independent of nodes outside the path.

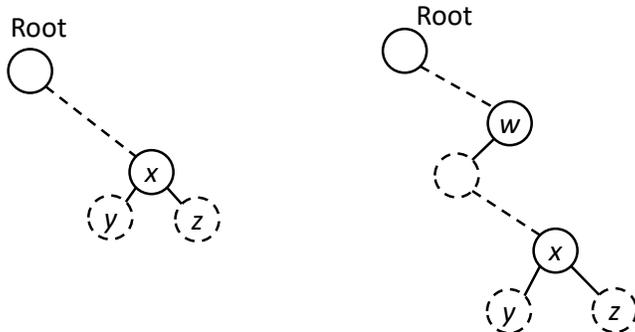


Figure 3: Possible locations for the next sub-root  $x$  to be insertion splayed in  $\pi \setminus (2, 3, 1)$ . The case on the left occurs when the next splayed node has left-depth 0, and the case on the right occurs when it has left-depth 1. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.

### Preorders

There are no restrictions on the possible structure of preorder insertion trees as  $\text{BST}(\text{preorder}(T)) = T$ .<sup>7</sup> However, the manner in which sub-roots are chosen is particularly simple.

**Lemma 3.** *If  $\pi \setminus (2, 3, 1) = (p_1, \dots, p_n)$  is a preorder then, for  $1 \leq i \leq n$ ,  $p_i$  is the smallest sub-root of  $T_{i-1}$ , where all nodes begin untouched in  $T_0 = \text{BST}(\pi)$  and  $T_i$  is formed by touching and splaying  $p_i$  in  $T_{i-1}$ .*

*Proof.* The statement is vacuously true for  $i = 1$ . We prove for  $i > 1$  by contradiction, as follows. Suppose  $T_{i-1}$  has some sub-root  $q$  that is smaller than  $p_i$ . Since  $q$  and  $p_i$  are both sub-roots in  $T_{i-1}$ , they are both children of respective (though not necessarily distinct) nodes  $a$  and  $b$  in  $T_{i-1}$ . Let  $r = \text{lca}_{T_{i-1}}(a, b)$ . Since  $q \neq a$  and  $p_i \neq b$ , all of  $p_i$ ,  $q$  and  $r$  are distinct nodes in  $T_{i-1}$ , and furthermore  $q < r < p_i$ . By Lemma 2,  $r$  precedes both  $q$  and  $p_i$  in  $\pi$ , and by construction  $p_i$  precedes  $q$ . We thus have  $(r, p_i, q)$  is a subsequence of  $\pi$ . But  $(r, p_i, q)$  is order-isomorphic with  $(2, 3, 1)$ , contradicting  $\pi \setminus (2, 3, 1)$ .  $\square$

**Theorem 2.** *Insertion splaying  $\text{preorder}(T)$  keeps each sub-root at left-depth at most 1 and takes  $O(1)$  amortized time per splay operation.*

*Proof.* The theorem is trivial for the first insertion splay. The inductive hypothesis is that every sub-root has left depth 0 or 1. Let  $x$  be the next sub-root to

<sup>7</sup>In fact, this property is shared by any permutation  $\pi$  for which every node in  $T$  appears in  $\pi$  before those in its left and right subtrees.

be splayed, and let  $y$  and  $z$  (either or both of which can be missing) be its left and right children. Touching  $x$  makes  $y$  and  $z$  into sub-roots.

Suppose  $x$  has left depth 0 before it is touched. Converting  $x$  from untouched to touched (without splaying it) increases the potential by at most 2 and gives the new sub-roots  $y$  and  $z$  left depths of 1 and 0, respectively. (In this case they are the only two sub-roots.) Each splay step, except possibly the last, is a zig-zig in which  $x$  starts as a left child with parent  $p$  and grandparent  $g$ . After completing the zig-zig,  $g$  is no longer an ancestor of any untouched node, which decreases the potential by 2. The zig-zig also preserves the left depths of  $y$  and  $z$ . ( $y$  becomes the right child of  $p$ .) No other sub-roots can increase left-depth, as  $x$  is the smallest sub-root. If the last splay step is a zig, the potential does not change (although the length of the path to  $y$  increases by 1).

More complicated is the case in which  $x$  has left depth 1. Converting  $x$  from untouched to touched (without splaying it) makes  $y$  a sub-root of left depth 2 and  $z$  a sub-root of left depth 1. Let  $w$  be the parent of the ancestor of  $x$  that is a left child. All other sub-roots are in the right subtree of  $w$ , which is unaffected by splaying  $x$ . The splay of  $x$  consists of 0 or more left zig-zigs, followed by a zig-zag (which can either left-right or right-left), followed by zero or more left zig-zigs, followed possibly by a zig. Each zig-zig reduces the potential by 2 and preserves the left depths of all sub-roots. The zig-zag does not increase the potential, reduces the left depth of  $y$  from 2 to 1, and that of  $x$  from 1 to 0, and preserves the left depth of  $z$ . Now  $x$  has left depth 0, and the argument above applies to the remaining splay steps.

By Lemma 3, the next node to be splayed will be  $y$  if present, otherwise  $z$  if present, otherwise  $w$  if present. All three of these items have left-depth 0 or 1, hence an identical form to Figure 3. Thus the hypothesis holds.

To obtain the constant factor, we observe that converting  $x$  from untouched to touched increases the potential by 2. Each zig-zig step pays for itself: it requires 2 rotations, paid for by the potential decreasing by at least 2. The zig-zag requires 2 rotations, and the zig requires 1 rotation. If the cost of a splay is the number of nodes on the splay path, equal to the number of rotations plus 1, we have an amortized cost of 6 per splay.  $\square$

## Postorders

Postorder insertion trees are more restricted. A binary search tree  $C$  is a (*left-toothed*) *comb* if the access path for  $x \in C$  always comprises some number  $j \geq 0$  of right children followed by some number  $k \geq 0$  of left children. The nodes of  $C$  are partitioned into *teeth*, where every node in the  $i^{\text{th}}$  tooth has right-depth  $i - 1$ . The shallowest node in a tooth is called the *head*. The insertion trees of postorders are combs:

**Lemma 4.** *If  $\pi$  is a postorder then no left child in  $\text{BST}(\pi)$  has a right child.*

*Proof.* By contradiction. Let  $y$  be a left child in  $\text{BST}(\pi)$  with right child  $z$ , and let  $x = \text{parent}(y)$ . As  $z$  is  $y$ 's right child,  $y < z$ . Similarly, as both  $y$  and  $z$  are in  $x$ 's left subtree,  $y < z < x$ . By Lemma 2,  $y$  can be an ancestor of  $z$

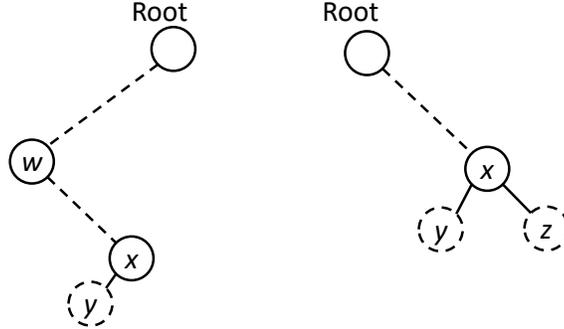


Figure 4: Possible locations for the next sub-root  $x$  to be insertion splayed in  $\pi \setminus (3, 1, 2)$ . The case on the left occurs when the next splayed node is less than the root, and the case on the right occurs when the next sub-root is the new tree maximum. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.

only if  $y$  precedes  $z$  in  $\pi$ , and similarly  $x$  must precede  $y$ . Thus,  $(x, y, z)$  is a subsequence of  $\pi$  that is order-isomorphic to  $(3, 1, 2)$ . By Lemma 1(b),  $\pi$  is not a postorder.  $\square$

While postorder insertion trees are less varied than for preorders, there may be many postorders with a given insertion tree. This affords some amount of freedom for choosing different sub-roots.

**Lemma 5.** *Let  $\pi \setminus (3, 1, 2) = (p_1, \dots, p_n)$  be a postorder with insertion tree sequence  $T_0, T_1, \dots, T_n$ . For  $1 \leq i \leq n$ ,  $p_i$  is either:*

- (a) *The single sub-root greater than  $\max\{T_{i-1}\}$  (if present), or*
- (b) *The largest sub-root smaller than  $\max\{T_{i-1}\}$  (if present).*

*Proof.* The result is vacuous for  $i = 1, 2$ . If  $p_i$  is case (a), we merely note that if  $p_i$  is a new maximum then it must be the right child of the largest node in  $\max\{T_{i-1}\}$ . There can be at most one sub-root in this position. Hence,  $p_i$  is unique.

For the sake of contradiction, suppose  $p_i$  is not of the form in case (a) or (b), and let  $q$  be the largest sub-root smaller than  $\max\{T_{i-1}\}$ . By Lemma 2, the items of each tooth are added in decreasing order. As  $q$  is not the head of its tooth, its successor  $r$  must be in  $T_{i-1}$ , and furthermore  $r$  precedes both  $p_i$  and  $q$  in  $\pi$ . By construction,  $(r, p_i, q)$  is a subsequence of  $\pi$ . Yet this subsequence is isomorphic to  $(3, 1, 2)$  since  $p_i < q < r$ , contradicting Lemma 1(b).  $\square$

**Theorem 3.** *Insertion splaying postorders maintains the following invariants:*

1. After each insertion splay, the path to every sub-root comprises  $j \geq 0$  left pointers followed by  $k \geq 0$  right pointers. (Furthermore, after the first insertion,  $k \geq 1$ .)
2. The left-depth of every sub-root decreases from smallest to largest.<sup>8</sup>
3. The splay operation takes constant amortized time.

*Proof.* The base case is trivial. Lemma 5 dictates that the next splayed sub-root is either greater than all marked items, or is the largest sub-root smaller than the tree root. Let  $x$  be the next node to be insertion splayed,  $y$  its left child, and  $z$  its right child (either or both children may be missing).

Suppose  $x$  is greater than the current tree root. Marking  $x$  increases the potential by 2 and makes  $y$  and  $z$  new sub-roots. The splay operation brings  $x$  to the root by a sequence of left zig-zigs followed possibly by a left zig (depending on whether the length of the access path is odd or even). After each one of these zigs or zig-zigs,  $y$ 's left-depth remains 1, and  $z$ 's left depth remains 0. Let  $v$  be the root prior to the splay operation. If the last splay step is a zig then the last splay operation increases the left depth of  $v$  and everything in its left subtree by either 1 or 2. Since the left-depth of  $x$  was 0 and  $x$  was the largest sub-root, the inductive hypothesis ensures that all sub-roots had left-depth at least 1 before the splay operation, and therefore at least 2 afterward. Thus, when  $x$  becomes the root, the left-depths of each sub-root decrease from left to right.

Otherwise,  $x$  is the largest sub-root less than the root. Marking  $x$  again increases the potential by at most 2. By Lemma 4,  $x$  has no right child (see Figure 4), so we only need to worry about its left child  $y$ . Let  $w$  be the last ancestor of  $x$  that is a left child. Each left zig-zig prior to the splay step involving  $w$  maintains the left-depth of  $y$  to be one greater than the left-depth of  $x$ . The splay step involving  $w$  will either be a left zig-zig or a left-right zig-zag, depending on the length of the original path connecting  $w$  to  $x$ . Regardless, immediately after the splay step involving  $w$ , the ancestor of  $y$  that is the left child of  $x$  is either the left child of  $w$  or the left child of  $w$ 's parent. Since all the sub-roots less than  $y$  are in the left subtree of  $w$ , and thus have left-depth greater than the left-depth of  $y$ , the invariant is restored, and remains true after each right zig-zig or zig that brings  $x$  to the root.

All that remains is showing constant amortized time. As noted before, marking  $x$  costs 2. If  $x$  is greater than the root then each left zig-zig, except possibly the last, pays for itself, giving amortized cost of 4. In the other case, all splay steps except for the one involving  $w$  and the one making  $x$  the root pay for themselves, giving amortized cost at most 6.  $\square$

## 4 Balanced Trees

Let  $|x|$  denote the size of the subtree rooted at  $x$ . Following [18], we say  $T$  is  $\alpha$  weight balanced for  $\alpha \in (0, 1/2]$  if  $\min\{|\text{left}(x)|, |\text{right}(x)|\} + 1 \geq \alpha \cdot (|x| + 1)$

<sup>8</sup>The first two invariants dictate that the ancestors of sub-roots form a *right-toothed comb*.

for all  $x \in T$ , and write  $T \in \text{BB}[\alpha]$ .

**Theorem 4.** *For any (fixed)  $0 < \alpha \leq 1/2$ , if  $S \in \text{BB}[\alpha]$  and  $T$  has the same keys as  $S$ , then the cost of splaying  $\text{preorder}(S)$  or  $\text{postorder}(S)$  starting from  $T$  is  $O(|T|)$ .*

*Proof.* By Theorem 1, it suffices to show that  $\text{DF}_T(\text{preorder}(S)) = O(|T|)$ . Let

$$A_\alpha(n) \equiv \max\{\text{DF}_T(\text{preorder}(S)) \mid S \in \text{BB}[\alpha] \text{ and } |T| = n\}.$$

Recall that  $\text{preorder}(S) = (\text{root}(S)) \oplus \text{preorder}(L) \oplus \text{preorder}(R)$ , where  $L$  and  $R$  are the left and right subtrees of the root of  $S$ , respectively. Notice that the rank differences between  $\text{root}(S)$  and the first item in  $\text{preorder}(L)$ , and between the last item in  $\text{preorder}(L)$  and the first item in  $\text{preorder}(R)$ , are at most  $|T|$  by definition. Hence,

$$\text{DF}_T(\text{preorder}(S)) \leq \text{DF}_T(\text{preorder}(L)) + \text{DF}_T(\text{preorder}(R)) + 2 \log_2(|T| + 1).$$

Observe that  $(|L| + 1)/(|S| + 1) \in [\alpha, 1 - \alpha]$  since  $S \in \text{BB}[\alpha]$ , and by definition  $|R| < |S| - |L|$ . Hence,

$$A_\alpha(n) = \max_{\alpha \leq \beta \leq 1/2} \{A_\alpha(\beta \cdot n) + A_\alpha((1 - \beta) \cdot n)\} + O(\log n).^9$$

Akra-Bazzi's result [2] suffices to show  $A_\alpha(n) = O(n)$  for fixed  $\alpha$ . The proof for postorders is identical.  $\square$

**Remark 1.** In actuality,  $A_\alpha(n) = O(f(\alpha) \cdot n)$  for some function  $f$  of  $\alpha$ . Unfortunately, the computation appears to be messy. We have declined to do the necessary footwork, as we strongly suspect that, regardless,  $A_\alpha(n)$  does not tightly bound the cost of splaying these sequences.

**Remark 2.** This result extends to any binary search tree algorithm that satisfies the dynamic finger bound. Iacono and Langerman proved Greedy also has the dynamic finger property [12]; their analysis does not consider initial trees, however.

## 5 Remarks

Patterns that avoid  $(2, 1, 3)$  are “symmetric” to those that avoid  $(2, 3, 1)$ : if  $\pi \setminus (2, 1, 3)$  then  $\pi$  is the preorder of the mirror image of  $\text{BST}(\pi)$ . Similarly, patterns that avoid  $(1, 3, 2)$  are symmetric to patterns that avoid  $(3, 1, 2)$ . Thus, insertion splaying  $\pi \setminus (2, 1, 3)$  and  $\pi \setminus (1, 3, 2)$  takes linear time.

The only other patterns of length three are  $(3, 2, 1)$  and its symmetric counterpart  $(1, 2, 3)$ . The pattern  $(3, 2, 1)$  was explored in [3], where it was shown

<sup>9</sup>Technically, since  $|L|/|S| < (|L| + 1)/(|S| + 1)$ , we need to pick  $S$  sufficiently large for a given alpha, and offset the recurrence term by a corresponding constant. This does not asymptotically affect the result.

that Greedy executes  $(3, 2, 1)$ -avoiding permutations in linear time starting from an *arbitrary* tree. In fact, they showed that executing  $\pi \setminus (k, \dots, 2, 1)$  takes time proportional to  $n \cdot 2^{O(k^2)}$ ; this is linear in  $n$  for fixed  $k$ . These permutations are called *k-increasing* because they can be partitioned into  $k - 1$  disjoint monotonically increasing subsequences [3]. They form the natural generalization of *sequential access*, which is the (unique) permutation of the tree nodes that avoids  $(2, 1)$ .

More general invariants can be derived about insertion tree structure and sub-root insertion order based on pattern-avoidance. As one particularly interesting example:

**Theorem 5.** *If  $\pi \setminus (k, \dots, 2, 1)$  then no node in  $\text{BST}(\pi)$  has left-depth more than  $k - 2$ , and the next sub-root inserted (without splaying) is always the smallest sub-root with its given left-depth.*

The proof is similar to Lemmas 4 and 5. In particular, the insertion trees of  $(3, 2, 1)$ -avoiding permutations look like the combs of postorder insertion trees, except the teeth are rightward, instead of leftward paths.

For  $k$ -increasing sequences, the potential used for Theorems 2 and 3 needs modifications. The main issue is that in both of these cases, the zig-zigs paid for themselves because the nodes knocked off the access path did not have sub-root descendants. This structure no longer holds for  $(3, 2, 1)$ -avoiding sequences, since we must splay the nodes of the teeth in increasing order. The proof seems to require a generalization of the sequential access theorem. It is possible that the notion of *kernel trees* used by Sundar in [22] for a potential-based proof of the sequential access theorem could be useful.

## References

- [1] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.
- [2] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.
- [3] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak. Pattern-avoiding access in binary search trees. In *FOCS*, pages 410–423, 2015.
- [4] R. Chaudhuri and H. Häußt. Splaying a search tree in preorder takes linear time. *ACM SIGACT News*, 24(2):88–93, 1993.
- [5] R. Cole. On the dynamic finger conjecture for splay trees. part ii: the proof. *SICOMP*, 30(1):44–85, 2000.
- [6] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting  $\log n$ -Block Sequences. *SICOMP*, 30(1):1–43, 2000.

- [7] E. Demaine, D. Harmon, J. Iacono, D. Kane, and M. Patrascu. The geometry of binary search trees. In *SODA*, pages 496–505, 2009.
- [8] K. Fox. Upper bounds for maximally greedy binary search trees. In *WADS*, pages 411–422, 2011.
- [9] N. Goyal and M. Gupta. Better analysis of binary search tree on decomposable sequences. *Theoretical Computer Science*, 2019.
- [10] L. Guibas and R. Sedgwick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.
- [11] B. Haeupler, S. Sen, and R. Tarjan. Rank-balanced trees. *TALG*, 11(4), 2015.
- [12] J. Iacono and S. Langerman. Weighted dynamic finger in binary search trees. In *SODA*, pages 672–691, 2016.
- [13] L. Kozma. *Binary Search Trees, Rectangles and Patterns*. PhD thesis, Saarland University, 2016.
- [14] J. Kujala and T. Elomaa. The cost of offline binary search tree algorithms and the complexity of the request sequence. *TCS*, 393:231–239, 2008.
- [15] C. Levy and R. Tarjan. A new path from splay to dynamic optimality. In *SODA*, pages 1311–1330, 2019.
- [16] J. Lucas. Canonical Forms for Competitive Binary Search Tree Algorithms. Technical report, Rutgers University, 1988.
- [17] I. Munro. On the competitiveness of linear search. In *ESA*, pages 338–345, 2000.
- [18] J. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SICOMP*, 2(1):33–43, 1973.
- [19] S. Pettie. Applications of forbidden 0-1 matrices to search tree and path compression-based data structures. In *SODA*, pages 1457–1467, 2010.
- [20] S. Pettie. Splay trees, davenport-schinzel sequences, and the deque conjecture. In *SODA*, pages 1115–1124, 2008.
- [21] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
- [22] R. Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12(1):95–124, 1992.
- [23] R. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.
- [24] R. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.
- [25] R. Wilber. Lower bounds for accessing binary search trees with rotations. *SICOMP*, 18(1):56–67, 1989.