Reduced Restructuring in Splay Trees

Evan Huus (Carleton University)
eapache@gmail.com

April 15, 2014

Abstract

In 1985, Daniel Sleator and Robert Tarjan published what has become a seminal paper in computer science, introducing the world to "splay trees" [11]. While splay trees have a number of excellent properties, several of which were proved in the original paper, they have tended to be less useful in practice due to the large amount of restructuring they must do on each operation. In this paper we investigate the various directions that have been studied over the last 29 years in order to reduce the amount of restructuring done by splay trees.

1 Introduction

In 1985, Daniel Sleator and Robert Tarjan invented the "splay tree", a clever scheme for restructuring binary search trees on the fly to maintain a number of powerful properties. In that paper alone they proved that splay trees are as good as the statically optimal tree for any sufficiently long sequence of accesses, that splay trees have the working-set property, and that splay trees have the static finger property for any fixed item f. Later papers by other authors have only added to the near-magical powers of splay trees; for example, Cole proved in 2001 that splay trees also have the dynamic finger property (see [4] and its companion paper [5]).

There are also a number of open conjectures on the splay tree which, if proven, would further strengthen its theoretical performance. In their original paper, Sleator and Tarjan conjectured that splay trees were not just as good as any statically optimal tree, but were in fact as good as any *dynamically* optimal tree for any sufficiently-long access sequence [11]. In 2001, John Iacono added the unified conjecture, combining and extending the working set and dynamic finger properties [7]. In addition to all of these impressive features, splay trees have the same bounds per operation in an amortized sense as any other balanced binary search tree algorithm; search, insert and removal can all be done in $O(\log n)$ time per operation.

Despite this, splay trees have turned out to be surprisingly impractical in real-world applications. Each operation is required to perform a non-trivial amount of work restructuring the tree in order to maintain all of the above properties, and while this restructuring does not change the order of the running-time, it does add a substantial constant factor. As such, splay trees have typically been outperformed by more traditional balanced tree algorithms even when the access sequence exhibits patterns for which splay trees have beneficial properties.

Because of this problem, a great deal of work has gone into improving splay trees, looking for tweaks and variants that reduce the restructuring penalty while trying to maintain all of the excellent properties of the original design. In this paper we survey such results, which exhibit a surprising number of problems. Many of the proposed variants sacrifice all of the properties that make splay trees so attractive, and few of them achieve competitive running-times despite this. There also appears to have been a fair amount of duplicated effort, with several authors inventing and testing the same designs. Confusingly, empirical results for identical designs are wildly inconsistent across multiple papers.

The remainder of this paper is structured as follows. In section 2 we introduce and summarize the core concepts of the original splay tree. This serves as the base data structure on which all the other variants are built. In section 3 we explore the numerous deterministic variants of the base splay tree, presenting these approaches by category, rather than chronologically or by paper. While the resulting order jumps around a bit, we believe that categorizing the approaches in this way reveals some essential patterns.

In section 4 we explore the randomized splay tree variants, which attempt to reduce restructuring at the cost of giving only expected running-time bounds. In section 5 we consider promising approaches from the previous sections, and some of the problems that they faced. We use this analysis to synthesize several interesting new splay tree variants. We conclude in section 6.

2 The Splay Tree

On the surface, the behaviour of the original splay tree as introduced by Sleator and Tarjan is extremely simple, which no doubt contributes in part to its charm. A splay tree is maintained as a typical binary search tree requiring no extra information at each node; left and right child pointers (and optionally parent pointers, though they can be avoided with a little work) are the only data needed. Whenever a node is inserted, removed, or accessed by search, that node is then *splayed*, itself a relatively straight-forward process.

The splay operation on some node, x, performs a series of tree rotations that bring x to be the root of the tree. Following Sleator and Tarjan, these rotations are typically presented in three cases named `zig`, `zig-zag`, and `zig-zig` (leaving off the symmetric cases of `zag-zig` and `zag-zag`). Intuitively however, splaying is only a subtle variant of the naive move-to-root approach. In the naive approach, we rotate the parent node of x in the direction such that x moves closer to the root, rotating right if x is a left child, and left if x is a right child.

With that context, the three cases of the actual splay algorithm are as follows:

zig This case only occurs when x ends up as the immediate child of the root; if d (the original depth of node x) is even then this case never occurs since the other two cases both operate two levels at a time. In the `zig` case, we perform a single rotation to make x the root, exactly as in the naive version.

zig-zag This case occurs when x is a right child and x's parent is a left child, or symmetrically (in the `zag-zig` case) when x is a left child and x's parent is a right child. The naming should be intuitive: to traverse from x's grandparent to x we would have to first "zig" one way and then "zag" the other. This case is also exactly the same as the naive version; we perform a rotation at x's parent, then at x's new parent (previously x's grandparent) to move x up two levels in the tree.

zig-zig This case covers the remaining possibilities; it occurs when x is a left child and x's parent is also a left child, or symmetrically (the `zag-zag` case) when x is a right child and x's parent is also a right child. This case is the only one which is not the same as the naive case, although the difference is subtle. Instead of performing a rotation at x's parent, then at x's new parent (its previous grandparent), we instead reverse that order. The splay operation first performs the rotation at x's grandparent, moving both x and its parent up one level. It then performs the rotation at x's parent, moving x up the second level.

While the difference between splaying and the naive move-to-root implementation appears minor, it actually makes a significant difference. The naive implementation has none of the excellent properties of the splay tree, and in particular can cost up to $O(n)$ per access, even in the amortized sense.

3 Deterministic Approaches

In this section we explore the numerous different deterministic approaches that have been tried to reduce the work done by splay trees. We present these attempts in three distinct categories. In 3.1 we deal with attempts to find alternative, more efficient implementations of the original splay tree structure that are more practically efficient (fewer pointer assignments per rotation, etc.) without changing the algorithmic properties of the structure.

In 3.2 we look at the techniques which make use of traditional splay operations, but only splay nodes conditionally. That is to say, these variants do not splay the node on every single insertion, deletion or access. Instead, some algorithm or extra data is used to decide whether to splay the node in question, or leave it in place. By reducing the number of splay operations these variants naturally reduce the restructuring penalty. However, they frequently sacrifice the splay tree's good algorithmic properties in the process.

In 3.3 we look at a more ambitious set of techniques. These variants change the splay operation in a fundamental way, sometimes by changing the order of rotations or by only splaying certain nodes along the path. The results for these variants are hit-and-miss. While some of them achieve good empirical improvements and claim to maintain the good algorithmic properties that make splay trees so attractive, others appear to be targeting different use cases entirely.

3.1 Splaying Implementations

Intuitively, the simplest method for reducing the work done by a splay tree is to improve the implementation. If, for example, some technique can be found for performing tree rotations with one fewer pointer change, then splaying can be done more efficiently without any effect on its algorithmic properties. As many of these special properties require complex proofs, it is highly advantageous to avoid the need to reprove them.

Top-Down Splaying The most prominent such improved implementation is called "top-down" splaying, and was introduced by Sleator and Tarjan in their original 1985 paper [11]. The idea behind top-down splaying is relatively simple, although the implementation is somewhat less intuitive. While traditional splaying works in a "bottom-up" manner, rotating the target node upwards in the tree until it is at the root, you will note that we have already traversed the path followed in order to find it in the first place. Top-down splaying effectively combines these steps: the tree is splayed as it is traversed downwards, reducing the total amount of work performed.

The implementation is less obvious however as, contrary to bottom-up splaying, top-down splaying cannot be done properly with just rotations. Suppose we are inserting, removing or accessing some element x. During traversal, the tree is broken into three distinct parts. The left part contains all the elements that are known to be less than x. The right tree, symmetrically, contains all the elements that are known to be greater than x. The final, "middle" part consists of the remaining elements which have as an ancestor the current node on the access path. The algorithm starts at the root, meaning the left and right parts are empty, and the middle part contains the entire tree.

During top-down splaying, we traverse the tree downwards, considering nodes in pairs just as in bottom-up splaying. Subtrees that we do not traverse (for example the right subtree of a node at which we turn left) are detached from their parent and efficiently added to either the left or right part of the tree. When the target node x is finally reached, the left and right parts are made its left and right children, and its original children are added to those new subtrees. As with bottom-up splaying, it is the `zig-zig` step that is slightly different in top-down splaying. When travelling down such a path, a single rotation is performed at that node before the normal splitting is done.

Sleator and Tarjan claim that their primary lemma (frequently known as the "access lemma") holds for top-down splaying with an unchanged constant factor compared to bottom-up splaying, however that top-down splaying is typically somewhat more efficient to implement.

Simple Splaying Sleator and Tarjan also consider the use of a method they call "simple splaying", which has an equivalent top-down variant. In simple splaying, the second rotation (or, in the top-down variant, the second "link") of the `zig-zag` case is skipped. They claim that the same analysis holds as was used for normal splaying, but with a slightly higher constant factor. As the actual splay operations are marginally faster however, they suggest that either method could prove empirically more efficient.

Empirical Comparisons In 1987, Erkki Mäkinen studied the properties of top-down splaying and simple top-down splaying, performing several theoretical and empirical measurements of their efficiency [10]. He also showed that while bottom-up and top-down splaying have exactly the same theoretical properties, they do result in slightly different trees under certain conditions. Based on a precise analysis of the implementation cost in pointer-assignments of the two primitives (rotation and link), Mäkinen suggested that top-down splaying has an effective constant factor of only 2.79 compared to a value of 3 for bottom-up splaying. Empirically, he compared top-down splaying and simple top-down splaying, finding the normal (non-simple) variant to be somewhat faster under most loads. Unfortunately, Mäkinen did not include a comparison with any kind of bottom-up splaying. The only control group was something he termed "Stephenson's variant" which was simply a top-down version of the naive approach outlined in section 2.

In 1993, Bell and Gupta did a much more thorough empirical study of various tree types. Although they did not implement the "simple" splay versions, they did compare top-down and bottom-up splaying with AVL trees, random trees, and other unrelated variants [2]. Their results indicated that, in general, splay trees simply weren't competitive. For balanced workloads, top-down splaying took approximately twice the time of an AVL or random tree to perform the same operations, and bottom-up splaying longer than that. Even for heavily skewed workloads, where the splay tree's extra properties should have made it shine, top-down splaying took just as long as the AVL and random trees, and bottom-up splaying was still worse.

However, these results did not go unchallenged. In 2001, Williams et al. performed an empirical evaluation of splay trees, using red-black trees and hash tables as benchmarks when working with large text collections [12].[1] They found that while neither top-down nor bottom-up splaying were competitive with a red-black tree or hash table, the difference was only around 25% (as compared with the factor of 2 or more found by Bell and Gupta). Adding to the mystery, they found that top-down splaying was, in fact, 10% *slower* than bottom-up splaying, not faster. Williams et al. suggested that this difference was due to

[1]Williams et al. also proposed and evaluated several new splay tree variants, but we will present those later.

the small size of the test data used by Bell and Gupta, permitting it to fit entirely in the CPU cache.

In 2005 Lee found, contrary to Mäkinen, that simple top-down splaying was in fact faster than normal top-down splaying (cited in [9]; I was unable to find the original paper). In 2009 Brinkmann et al. found, matching Bell and Gupta but contrary to Williams et al., that top-down splaying was faster than bottom-up splaying, but that the difference was small, less than 30% in all cases [3]. The only conclusion I can draw is that more thorough experiments are needed to precisely control for the numerous variables that might affect the results.

3.2 Conditional Splaying

While different implementations of the core splaying concept can go some way to reducing the amount of work performed by a splay tree, that approach is fundamentally limited in that it must move every accessed node all the way to the root, and must perform certain operations all the way along that path in order to achieve the necessary guarantees. As such, a substantial amount of work has also gone into methods to reduce the number of splay operations performed.

In these conditional splaying variants, certain accesses are splayed and others are not, depending on some heuristic or additional data. This has obvious and substantial benefits – splaying half as much obviously spends only half as much time on restructuring – but weakens or voids many of the nice properties that have been shown to hold for proper splay trees.

Long Splaying As with the implementation-based improvements, Sleator and Tarjan again lead the charge, suggesting two different heuristic conditions in their 1985 paper [11]. The first of these, called "long splaying", proposes splaying a node only if its path is too long. This has an obvious intuitive appeal; splaying a node shortens its path, so by applying this heuristic the resulting tree should remain relatively balanced.

The question then becomes one of what constitutes "too long". In their paper, Sleater and Tarjan suggest that the path to a node x be considered too long if it is longer than $c' \log(W/w(x)) + c'/c$, where c is the constant factor from their Lemma 1 (the "access lemma"), c' is some constant greater than c, $w(x)$ is the fixed node weight of x, and W is the sum of all node weights in the tree.

In theorem 7 of their paper, Sleator and Tarjan prove that under this definition, the total splaying cost is independent of the number of accesses and proportional to the amortized cost of accessing each element once. These properties suggest that our intuition was correct; the tree is splayed until it is relatively balanced, and further splaying does not occur.

Unfortunately, we are not aware of anyone who has bothered to implement this method and compare it to other balanced trees such as red-black trees. This variant also obviously lacks most of the splay tree's other good properties since it does not adapt to changes in usage patterns.

Snapshot Splaying Also from Sleator and Tarjan's original paper is a method which they refer to as "snapshot splaying" [11]. In this variant, splaying is performed normally for the first j operations. For all subsequent operations, no splaying occurs; the tree is treated as a static binary search tree. The intuition here is that j splay operations is enough to put the tree in a relatively good shape. This is also supported by the fact that, since splay tree bounds are proved under amortization, we can somehow expect the average state of the tree to be good.

While this variant, like long splaying, clearly lacks many of the nice properties of a full splay tree, Sleator and Tarjan still prove an interesting result on it in theorem 8 of their paper (the "snapshot theorem"). Consider a sequence of m accesses on any initial tree, where the snapshot value of j is chosen uniformly from $1 \ldots m$. Then the resulting search tree has an expected access time related to the entropy of the distribution: $O((n \log n)/m + \sum_{i=1}^{n} p_i \log(1/p_i))$.

Unlike long splaying, this variant has been tested empirically at least once, as it was one of the types included in the 2001 study by Williams et al. [12]. Called the "stopsplay" heuristic in that paper, they found the results to be inconsistent depending on j, but that it did not seem to offer any practical improvement over normal splaying.

Periodic Splaying Another relatively straight-forward variant is one which we will name "periodic splaying". Unlike the others so far, periodic splaying has been well-studied, appearing in several distinct empirical evaluations. In this variant, some period k is chosen. Every kth operation is then splayed, while any other operations are not. With $k = 2$ this then results in $1/2$ the splay operations of a traditional splay tree. The intuition is that the somewhat-reduced adaptive balance is more than compensated by less time spent splaying.

This method has a peculiar history. It seems to have been first suggested in passing by Bell and Gupta [2]. As it was not the focus of their paper however, they did not name or analyze this method. In 2001, Williams et al. propose this method and call it "periodic rotation" [12]. While they cite Bell and Gupta elsewhere in their paper, they do not credit them with this heuristic. Instead, they mention a similar randomized algorithm proposed by Fürer [6].

In 2007, Lee and Martel proposed this heuristic again and called it k-splaying. While they do cite Williams et al. elsewhere, they instead credit their version as a deterministic variant of the randomized algorithm invented by Albers and Karpinski [1]. It is unclear which of these many proposals were valid new inventions and which were not; regardless, original credit must go to Bell and Gupta as their paper precedes the others by a significant margin.

In their empirical evaluation, Williams et al. found periodic rotation to be relatively successful, providing an improvement of approximately 27 percent and making splay trees effectively competitive with other balanced tree types. They found a period of 11 to be roughly optimal for their workloads, though they note that this could vary substantially. Lee and Martel found periodic rotation to out-perform a random BST by 18%, and that the optimal value of k shrank as the trees grew, down to the point where $k = 4$ was optimal for trees with more than one million nodes.

As with many of the other variants, this one sacrifices most of the nice properties of splay trees. There are certain obvious access patterns where periodic splaying results in $O(n)$ amortized cost per operation, making it no better than any unbalanced binary search tree. However, it does seem intuitive that periodic splaying results in an expected behaviour close to that of splay trees, at least for unpatterned access sequences.

Count Splaying Among the more complex conditional splaying variants is one proposed by Williams et al. in 2001, which they call "count splaying" [12]. In this scheme, an access count is stored with each node, and incremented on every access. When the count for a particular node reaches a certain threshold, that node is splayed. They found this method to work particularly poorly as it adapted too slowly to changes in access patterns.

Sliding Window Splaying In 2007, Lee and Martel proposed an interesting variant based on a sliding window [9]. The depth of each of the last w accesses are saved with the tree; if fewer than t of them have depth less than d, then the next access is splayed. The intuition behind this method is that if most of the accesses are near the root, then splaying simply produces unnecessary shuffling. If, however, many accesses are deep in the tree, then splaying will be able to pay for itself by improving the shape of the tree.

For their empirical study, Lee and Martel chose values of $d = \log_2 n$, $w = 32$ and $t = 16$. They found that while sliding-window splaying was a substantial improvement over normal splaying, it performed only marginally better than the much simpler periodic splaying.

3.3 Splaying Differently

While we have explored different implementations of splaying in 3.1 and various conditional variants in 3.2, all of those attempts make use of the same fundamental splaying operation described in Sleator and Tarjan's original paper. However, there are a number of other variants which do not touch the implementation or frequency of splaying. Instead, the splaying operation itself is changed to require less work.

In some of these variants, only part of the path is splayed, depending on certain conditions or additional data. In these cases, the splayed node moves up, but doesn't necessarily make it all the way to the root. In other variants, the order or number of rotations made is changed in certain cases, with a variety of interesting side-effects. While the different splaying implementations trivially maintained all of the nice properties of splay trees, and the conditional variants almost universally sacrificed those properties, the methods found here are a mixed bag. Some of them have been shown to have all the same properties of splay trees with a different constant factor, while others have no theoretical results at all.

Semi-Splaying Once again, the first variant for this section comes from Sleator and Tarjan's original 1985 paper [11]. Called semi-splaying, this method is identical to normal splaying in the `zig` and `zig-zag` case. However, the `zig-zig` case involves one fewer rotation; the first rotation at the node's grandparent occurs as normal, but the second rotation at the node's parent does not. Instead, the semi-splaying algorithm simply continues from the parent. The net result is that semi-splaying performs fewer rotations than regular splaying, but does not necessarily move the splayed node all the way to the root of the tree. Instead, it ends up with depth at most half of where it was previously.

Sleator and Tarjan present interesting semi-splaying results in their paper. They claim that their access lemma holds for semi-splaying with a constant factor of 2 (compared to a constant factor of 3 for normal splaying). This is sufficient to imply that semi-splaying has many if not all of the nice properties of normal splay trees. They also give a top-down equivalent of semi-splaying to go with the top-down implementation of normal splaying.

In the 2001 study by Williams et al. they do not present any specific results related to semi-splaying, however they do state that "In all cases, we have found that semi-splaying performs worse than all other variants described in this paper, including RSTs" [12]. This seems counter-intuitive, though Brinkmann et al. point out that, based on the admittedly vague description, the variant tested by Williams et al. might not have been the same as the variant proposed by Sleator and Tarjan [3].

Independent and Simple Semi-Splaying Brinkmann et al. also propose their own pair of minor variants, called "simple semi-splaying" and "independent semi-splaying" respectively. Simple semi-splaying appears to be a combination of the simple splaying implementation from section 3.1 and the semi-splaying algorithm described above.

They note that this combination can be understood entirely without rotations. Instead, the current node, its parent and grandparent are restructured into a complete binary tree of three nodes, then the algorithm repeats at the root of this new tree. Independent semi-splaying is presented in a similar fashion, except that it is performed top-down and nodes are never reused. The path is split into strictly independent node triples, each of which is restructured to be a complete binary tree. They note that while semi-splaying guarantees that a node will end up at least half as deep as before, independent semi-splaying guarantees only that it will end up two thirds as deep as before.

Empirically, Brinkmann et al. compared semi-splaying with both its simple and independent variants, using normal top-down splaying as a comparison point. They found that semi-splaying performs around 10% better than regular top-down splaying, but that simple semi-splaying performs only about on par, depending on how heavily weighted the access distribution was. Independent semi-splaying, on the other hand, performed nearly 15% better than the normal top-down version. Unfortunately, Brinkmann et al. do not benchmark any other tree types for comparison.

Partial Splaying In 1992, Klostermeyer published a paper which proposed several interesting variants in this category and did a simple empirical simulation to test their performance [8]. One of these, the "partial splay tree", shares an approach with the conditional "count splaying" scheme proposed by Williams et al. [12] and covered in section 3.2 of this paper.

In both versions, each node keeps a counter of the number of times it has been the target of an access. In the version of Williams et al., a node is splayed fully when the count exceeds a certain threshold. However in the version presented by Klostermeyer each node is always splayed, but only until its parent has a greater access count. This results in frequently-accessed nodes appearing near the root, but not getting displaced when an infrequently accessed node is splayed.

Empirically, Klostermeyer compared partial splaying to traditional splaying and to an AVL tree as a benchmark. They found that partial splaying did not provide any advantage except in specific cases of static but non-uniform access frequencies since, like the count splaying scheme of Williams et al., it adapted poorly to changes in access frequency.

Partway Splaying Another variant presented by Klostermeyer was called the "partway splay tree". In this tree, a node is splayed partway to the root, as in the partial splay tree, but the distance splayed is determined as a specific fraction of the node's existing depth. For example, an obvious choice would be to have the tree always splays a node to one half its previous depth.

Empirically, Klostermeyer found that partway splaying performed almost exactly on par with traditional splaying, offering no particular advantage.

Height-Balanced Splaying In the height-balanced splaying scheme also presented by Klostermeyer, a splay tree is in some sense combined with a classic AVL tree such that splaying occurs, but only within the limits of keeping the tree balanced. An AVL tree never permits a node's left and right subtrees to be out of balance by more than one, but Klostermeyer found this to be overly restrictive. Instead, this was relaxed to some arbitrary "balance limit".

In height-balanced splaying, the path from node to root is walked normally. However, instead of splaying for every triple of nodes, the `zig` and `zig-zag` splaying operations are performed only when they preserve the balance limit of the tree. This provides some of the benefits of splaying, but ensures that the tree remains at least somewhat balanced even in the worst case. Height-balanced splaying was found to behave marginally better than traditional splaying, but was still not competitive with AVL trees except under extremely biased workloads.

Reference-Balanced Trees The reference-balanced tree is sufficiently different that it is not, strictly speaking, a variant of the splay tree at all. It seems to me better described as a variant of an AVL tree instead. However, it was presented by Klostermeyer together with height-balanced splaying so we will touch on it here.

In the reference-balanced tree, each node maintains balance between its left and right subtrees just as in an AVL tree. However, instead of balancing based on height as an in AVL tree, a reference-balanced tree balances (as the name implies) on the number of accesses to each subtree, counted as in Klostermeyer's partial splaying scheme, or Williams et al.'s count splaying scheme.

The reference-balanced tree explicitly sacrifices many of the dynamic properties of splay trees in order to balance accesses across the tree. Klostermeyer found it to perform best in practice of all the variants tested, though it still wasn't competitive with the AVL tree on evenly-distributed workloads.

4 Randomized Approaches

In the previous section we explored a substantial number of deterministic variants on the original splay tree of Sleator and Tarjan, however a number of randomized variants have also been proposed. These variants use probabilistic methods to achieve better expected running times, typically at the cost of poorer worst-case performance.

Full Random Splaying The full random splaying scheme appears to have been proposed independently by Albers and Karpinski (cited in [1] but I could not find the original) and Fürer [6]. In this simple scheme, similar to the deterministic periodic splaying scheme from section 3.2, each access is splayed only with some probability.

Albers and Karpinski performed a simple empirical evaluation of this scheme, finding that it was moderately effective for certain query distributions, but behaved poorly for ones that exhibited high locality of reference. They also note that the simple deterministic version (periodic splaying) behaves just as well but without the cost of generating the random numbers. It is worth noting that the variant proposed by Fürer is technically full random *semi*-splaying.

Random Parent Splaying In order to maintain the performance of full random splaying but without sacrificing the guaranteed amortized performance provided by normal splaying, Fürer proposes two further algorithms [6]. The first, which he referred to simply as "algorithm B", I will refer to as random parent splaying. In this variant, with probability one half, the parent of the target node is (semi-)splayed instead of the node itself.

I do not particularly understand the intuition behind this scheme – Fürer's treatment of it is quite brief – though he does claim that it maintains the amortized worst-case behaviour of normal splay trees while somehow providing 25% better expected performance.

Partial Random Splaying The third algorithm proposed by Fürer (which he calls "algorithm C") is in some sense a combination of full random splaying and random parent splaying. In this algorithm, which I will call "partial random splaying", some constant k is chosen. When splaying an element, one edge pair is chosen uniformly at random from the first k edge pairs along the path. Semi-splaying is then performed at this edge pair and every kth edge pair along the path from it; all other edges are unchanged.

Again, Fürer's treatment is very brief, but he claims that this algorithm maintains the amortized worst-case behaviour of normal splay trees while providing the same expected running-time as full random splaying, above. He also notes that this method is amenable to conversion into a top-down method. Unfortunately I have been unable to find any further evaluation (empirical or theoretical) of this method.

5 New Work

So far we have sampled a substantial number of splay tree variants based on several different fundamental approaches. In this section we consider some of the advantages and shortcomings of the variants we have seen, and propose several new variants that may perform better.

Improved Partway Splaying Klostermeyer proposed a variant called partway splaying where each node was only splayed a fixed fraction of the way to the root [8]. He found that in practice this method performed no better than regular splaying. Consider, however, the behaviour of such a tree; given a node at depth d, it would take $\log d$ consecutive accesses to that node to bring it to the root. It seems possible that a similar variant might perform better if the distance splayed was not a function of d but instead a function of n, or a function of the maximum depth of any node.

For example, consider a partway splay tree that splayed each node at most $\log n$ steps. If the access pattern (and thus the tree) is relatively balanced then this behaves almost exactly like a normal splay tree. But if the access pattern is unbalanced, then infrequently accessed nodes are not brought all the way to the root; frequently accessed nodes still dominate there. Of course this method has its drawbacks; it may take $n/\log n$ consecutive accesses to bring a distant node all the way to the root.

Alternatively, consider a partway splay tree that splays each node at most half the depth of the tree. This is sufficient to bring any node to the root in no more than three consecutive accesses, but it has some of the apparently nice properties of the previous suggestion.

Counted Age Splaying The counted splaying method proposed by Williams et al. [12] and the partial splaying scheme suggested by Klostermeyer [8] were both found to adapt poorly in changes in access patterns, as the counter for each node tracked only frequency, not how recently those accesses had occurred. What is needed here is a way for the access counts to gradually age out over time, thus tracking both frequency and recency. Fortunately, the sliding window variant from Lee and Martel does just that [9].

However, while Lee and Martel used one sliding window for the entire tree, it seems quite feasible to maintain a window per-node instead. If each node stored the "time" of the most recent accesses to that node (using a single tree-wide access counter as a simple "timer"), then a node could be splayed on one of several interesting conditions. For example, a node could be splayed only until the sum of its previous few access times becomes less than its parent's, similar to the partial splaying scheme by Klostermeyer. Or, a node could be splayed only when the average of its last few access times is close (to within some constant value) to the current "time", similar to the counted splaying method of Williams et al.

This scheme could even work if only the age of the single most recent access were stored with each node, thereby taking no more space than the simple counted splaying scheme from Williams et al. All of these variants would have some of the same nice properties as the ones which they mimic, but would adapt much more quickly to changes in access patterns.

Freeze Splaying The "snapshot splaying" method proposed by Sleator and Tarjan in their original paper has certain interesting properties, but over the long run it lacks any semblance of adapting to access pattern changes; the tree is entirely static once frozen [11]. The sliding window scheme proposed by Lee and Martel is in some sense an answer to this. We expect that in practice the tree will "freeze" for a while (no splaying) when the window contains many short accesses then "thaw" (splaying) when the window contains many long accesses [9].

The version by Lee and Martel was found in practice to be complex to implement, and to perform only marginally better in practice than the much simpler periodic splaying. However, we can achieve this freeze/thaw effect without the complications of a sliding window. The simplest way, of course, is to maintain a counter: freeze for k accesses, then splay for k accesses, and repeat. We do not expect this naive version to perform any better than periodic splaying in practice.

There are other methods of deciding when to freeze and thaw, however. Rather than maintaining a sliding window, we could maintain a single counter. On each access, the depth of the accessed node is added to the counter, and the entire value is reduced by 10% (or reduced by a fixed $\log n$, or any number of other techniques possible to "age" it). Splaying is performed if that value is over some threshold (which itself may be a function of n).

It may also be effective to use different triggers for freezing and thawing. For example, a thaw could be triggered by any access to a node over a certain depth d, while a freeze could be triggered by any access to a node under a certain depth d' (where $d' < d$). This would have the advantage of greatly reducing churn near the top of the tree.

Combinations As many of the previously explored variants are in some sense disjoint, there are also a large number of available combinations that have not yet been explored. Most of the proposed deterministic variants could probably be used to generate a randomized equivalent, though if the experience of Albers and Karpinski is any guide these will not have any immediate performance advantage over their deterministic cousins [1]. They would, however, have the advantage of being resistant to attacks via crafted access sequences.

Another area for generating combinations lies in both splaying conditionally (section 3.2) and splaying differently (section 3.3). Brinkmann et al. suggest combining their semi-splaying variants with some of what they call "partial techniques" as an interesting direction for future research [3]. For example, periodic semi-splaying seems like it might show promise given the benefits shown individually by its two parents.

6 Conclusion

As we have seen, there are a truly astonishing number of splay tree variants proposed in the literature, and new ones that show some intuitive promise are relatively easy to invent. This multiplicity is a testament to the power and intriguing properties of the original splay tree, and to its poor practical performance.

Unfortunately, the vast majority of the variants we found sacrificed everything that makes the splay tree so nice in order to win this practical performance. The exceptions to this rule are sparse: Sleator and Tarjan's semi-splaying seems to maintain most of the splay trees properties, as does independent semi-splaying by Brinkmann et al. The different implementations from section 3.1 do of course, but the room for further improvement in that area seems limited.

Another point of particular concern is the lack of consistent empirical results reported by various authors. Despite efforts to control for common variables, many of the results we found were not just different but in fact flatly contradicted each other. Clearly further study is needed, in particular to nail down the effects of various parameters on behaviour.

Several such problematic parameters include key type and tree size. While some studies were done with integers (which are fast to compare), others were done with strings (which are not). If one variant increases the number of required comparisons while decreasing other work done, it might perform better than usual on integer keys, but worse than usual on string keys. Also relevant is the size of the tree compared to the size of the cache (a variable addressed only by Lee and Martel [9]). If the entire tree fits into the processor's cache, certain operations may have different effective costs than if the algorithm must frequently go to main memory.

Any survey of this type is bound to be incomplete, especially given the breadth of literature on the topic. There are several papers cited here for which we were unable to find or access an original copy; we have had to make do with what is reported in abstracts and quoted in papers we do have access to. However, we hope this provides an effective and useful jumping-in point for future work in this area.

References

[1] Susanne Albers and Marek Karpinski. Randomized splay trees: Theoretical and experimental results. *Information Processing Letters*, 81(4):213 – 221, 2002.

[2] Jim Bell and Gopal Gupta. An evaluation of self-adjusting binary search tree techniques. *Software Practice and Experience*, 23:369–382, 1993.

[3] Gunnar Brinkmann, Jan Degraer, and Karel De Loof. Rehabilitation of an unloved child: semi-splaying. *Software: Practice and Experience*, 39(1):33–45, 2009.

[4] R. Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[5] 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. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[6] Martin Fürer. Randomized splay trees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 903–904, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[7] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 516–522, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[8] William F. Klostermeyer. Optimizing searching with self-adjusting trees. *Journal of Information and Optimization Sciences*, 13(1):85–95, 1992.

[9] Eric K. Lee and Charles U. Martel. When to use splay trees. *Software: Practice and Experience*, 37(15):1559–1575, 2007.

[10] Erkki Mäkinen. On top-down splaying. *BIT Numerical Mathematics*, 27(3):330–339, 1987.

[11] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[12] Hugh E. Williams, Justin Zobel, and Steffen Heinz. Self-adjusting trees in practice for large text collections. *Software: Practice and Experience*, 31(10):925–939, 2001.