Splay Tree Variants: Theory and Experiment

Zhi Wei Gan
zgan@mit.edu

Edward Jin
ehjin@mit.edu

Pranav Krishna
pkrishna@mit.edu

***Abstract*—Splay trees are a special type of binary search tree that perform well without storing additional data and are conjectured to be optimal in all cases. We investigate both deterministic and randomized variants of standard splay trees and analyze them both theoretically and experimentally.**

***Index Terms*—splay trees, randomization, data structures, implementation**

I. Introduction

We seek to solve the problem of somehow improving on the performance of splay trees while still keeping some semblance of its simplicity. In this paper, we present a variety of schemes that have been discussed in other papers, and supplanted them with novel schemes. While no breakthrough was made in terms of improvements, we shed light on why this might be the case.

II. Background on Splay Trees

A. Implementation of Operations

The splay tree [1] is a revolutionary self-adjusting binary search tree that was introduced by Sleator and Tarjan, supporting the operations Insert, Delete, and Access in amortized $O(\log n)$ per operation, without storing any additional information. The central operation that allows for this runtime is the Splay, which repeatedly performs double rotations on the accessed node until it is at the root of the tree.

The rotations in the Splay operation depend on the relative positions of the node x, its parent y, and its grandparent z:

- If the parent of x is the root, then apply a standard single rotation to bring it to root. This is known as a *zig* step if x is a left child, and a *zag* step otherwise.
- If x and y are both left or right children of their respective parents, then apply a double rotation. First, rotate the edge between y and z. Then, rotate the edge between x and y. This has the net effect of bringing x to where g originally was, and is known as a *zig-zig* step when x and y are both left children, or a *zag-zag* step in the other configuration.
- If x and y are different-sided children of their parents, then first rotate the edge between x and y, and then the edge between y and z. This is known as a *zig-zag* step when x is a right child and y is a left child, and as a *zag-zig* step in the other configuration.

The rotation operations are displayed below.

With the Splay operation, the Insert, Delete, and Access operations can be readily implemented. We implement these operations in the same way as in a standard binary search tree, with the critical difference that we Splay after each operation. For Insert, we insert as usual, then splay the newly inserted element. For Delete, we first swap the node we want to delete with a leaf node, then delete it, as is standard. Then, we splay the parent of the deleted node. Finally, for Access, we do a standard tree traversal and then splay the element requested. If the element is not found, we will instead splay the last element visited in the traversal.

B. Runtime Analysis

We will first assign weights w_x to each node x. We will define the *size* of a node $s(x)$ to be the sum of the weights of all its descendants. We will also define the *rank* of the node $r(x)$ to be equal to $\lg s(x)$. Finally, we will use an amortized analysis with the potential function $\Phi = \sum r(x)$, where the summation is over all nodes x.

Lemma 1. *(Access Lemma) The amortized cost for a splay that moves node x to the top of a splay tree with root t is at most $3(r(t) - r(x)) + 1$, where each rotation costs 1.*

Proof. We show that each double rotation on node x costs at most $3(r(z) - r(x))$, where z is the grandparent of node x as before. Then, the total cost of the Splay operation telescopes to $\sum 3(r(z) - r(x)) \leq 3(r(t) - r(x))$. We add a $+1$ at the end to account for the potential single rotation from the *zig* step at the end.

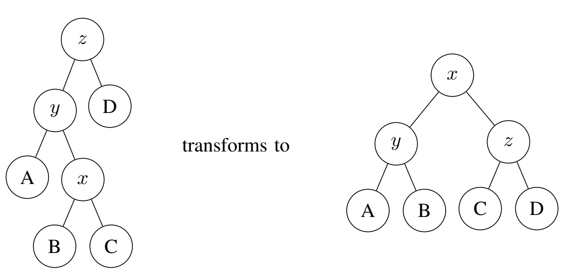

Fig. 1: A zig-zag step.

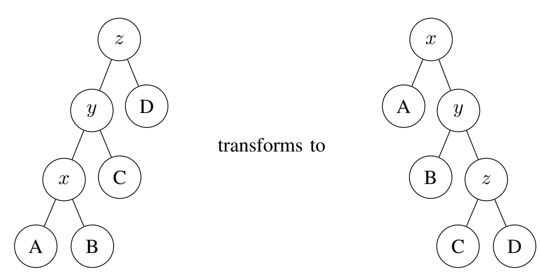

Fig. 2: A zig-zig step.

Note that in each double rotation, aside from the node x, parent y, and grandparent z, the rank of all other nodes stays constant. The amortized cost of one double rotation is $2+\Delta\Phi$, since we need to perform two rotations. Let $r(x), r(y)$, and $r(z)$ be the original ranks of each node, and let the ranks of the nodes after the splay operation be $r'(x), r'(y)$, and $r'(z)$. For all cases, we have $r(z) = r'(x)$, and so the amortized cost of double rotation is $r'(z) + r'(y) - r(x) - r(y) + 2$.

For the zig-zig case, we have $r(y) \geq r(x)$ and $r'(y) \leq r'(x)$. This means that the amortized cost is bounded by $2 + r'(z)+r'(x)-r(x)-r(x) = (r'(x)-r(x))+(2+r'(z)-r(x))$. Then, proving the latter term is bounded by $2(r(z)-r(x))$ will show the lemma, since the first term is equal to $r(z) - r(x)$.

Rearranging the inequality, we need to show that

$$(r(x) - r(z)) + (r'(z) - r(z)) \leq -2$$

If we consider the subtrees according to Figure 1, where $|A|$ denotes the size of subtree A, then we have that

$$r(x) - r(z) \leq \lg \frac{|A| + |B|}{|A| + |B| + |C| + |D|}$$

and also that

$$r'(z) - r(z) \leq \lg \frac{|C| + |D|}{|A| + |B| + |C| + |D|}$$

If we let $q = \frac{|C|+|D|}{|A|+|B|}$, then the sum of these two terms becomes $\lg \frac{1}{1+q} + \lg \frac{q}{1+q}$. Mathematically, this quantity achieves a maximum -2 at $q = 1$, which shows the required claim for zig-zig rotations.

For the zig-zag case, we once again have that $r(y) \geq r(x)$. Then, the relevant inequality for showing the claimed bound is that

$$2 + r'(z) + r'(y) - r(x) - r(x) \leq 3(r'(x) - r(x))$$

or after rearrangement, that

$$\begin{aligned} & r'(z) + r'(y) + r(x) - 3r'(x) \\ &= \lg \frac{r'(z)}{r'(x)} + \lg \frac{r'(y)}{r'(x)} + \lg \frac{r(x)}{r'(x)} \leq -2 \end{aligned}$$

Considering Figure 2, we can find the values of the rank functions in terms of the sizes of the subtrees. Substitution in the above inequality shows that the first two terms turn out to have the same values as in the zig-zig case, and thus we know that the sum of the first terms is already upper bounded by -2. The third term is clearly negative since $r(x) < r'(x)$, and so the original inequality bounding the cost of the zig-zag case is verified.

Finally, the change in potential in a zig step is equal to $r'(y) - r(y) + r'(x) - r(x)$. Since $r'(x) = r(y)$ then this is bounded by $r'(y) - r(x) \leq r'(x) - r(x)$, which telescopes the sum and adds at most 1 to the cost from the single rotation. □

Corollary 1. *The amortized runtime of any splay tree operation is $O(\log n)$.*

Proof. Set the weights of every node to be 1. Then, the amortized cost of a splay becomes $3(r(t) - r(x)) + 1 \leq 3(\log(n) - 0) + 1 = O(\log(n))$.

Since accessing an element requires less work than the actual splay operation, this means that an ACCESS operation takes amortized time $O(\log n)$. INSERT and DELETE operations involve an access and $O(1)$ pointer operations, in addition to the splay at the end, and so they also take amortized $O(\log n)$ time. □

C. Optimality of Splay Trees

Splay trees seem to match the performance of other binary search trees, with $O(\log n)$ runtime for all operations. However, they are also optimal in different ways:

Theorem 1. *(Static Optimality Theorem) Suppose we perform operations such that item x is accessed with probability p_x. Then, the cost of all operations on a splay trees will be at most a constant times the optimal static binary search tree, without knowing the individual p_x.*

Proof. To optimize the time of performing the operations, the best thing to do is to put the most-accessed items in the levels near the root. Each level k in the tree has 2^k spots for elements, meaning that all elements with $p_x \geq 2^{-k}$ can be put in level k. The expected search cost for an element in the optimal static tree is then $\sum -p_x \log_2 p_x$.

For a splay tree, the cost of any access is bounded by a constant times the cost of a splay, as above. Let the weights $w_x = p_x$, and define W to be the size of the root. By the access lemma, the cost of each splay is $O\left(\log_2\left(\frac{W}{w_x}\right)\right) = O\left(\log_2\left(\frac{1}{p_x}\right)\right)$. This means that the expected search cost is $O(p_x \log_2(1/p_x))$, showing that our runtime is within a constant factor of the static optimum. □

A similar analysis can show that the second deterministic scheme and all the randomized schemes presented in this paper, for fixed p, also satisfy Static Optimality.

It was conjectured by Tarjan that splay trees are in fact *dynamically optimal*, meaning that they do within a constant factor of any binary search tree, even when the other has access to all requests in advance. This conjecture is generally thought to be true as no counterexamples have been found so far in about 40 years.

III. DETERMINISTIC SCHEMES

In this section, we explore different deterministic schemes that seek to improve the expected performance of splay trees. As far as we are aware, these schemes are novel, though a variant of even splaying, known as 'semisplaying,' was proposed in [1].

A. Even Splaying

We first propose a variant of splaying called "even-splaying." After each operation, we will splay the node itself if it is an even number of nodes away from the root, or splay the parent otherwise. Since the splay operation itself is unchanged, the Access Lemma and its corollary still applies. Thus, we can still perform all of our tree operations in $O(\log n)$ time. The main benefit of this approach is that it removes the necessity of the zig step at the end of splaying, thus saving one rotation.

One of the main benefits of splay trees is that commonly accessed elements will be moved to the root, such that repeated accesses are cheap. We claim that even splaying also maintains this property in expectation, such that only $O(1)$ repeated accesses are needed in expectation for an element to move close to the root. We further claim that the worst case behavior is needing $O(\log h)$ repeated accesses, where h is the depth of the tree. To show these claims, consider the following cases when we access an element that is an odd number of nodes away, with the notation as in the above figures of rotations:

- If we need to perform a zig-zig step, the splayed element x gets moved up by 2, subtree A moves up by 2, and subtree B moves up by 1.
- If we need to perform a zig-zag step, then the splayed element x move up by 2, and both of the subtrees move up by 1.

Since the element we want to access still remains in a subtree of the splayed node, every double rotation leads to the accessed element moving up by at least 1. The parent moves to the root, and so we move the element halfway up the tree in the odd case, and fully in the even case. This means that we only need 2 accesses in expectation for the element to reach the root or be close to it. If we are unlucky and our node is always an odd distance away, then we will still move up halfway towards the root, ultimately only needing $O(\log h)$ accesses before achieving constant access times. Since $h = O(\log n)$ in expectation, in practice, this means that even splaying will maintain this characteristic property of standard splay trees.

B. *k*-Rotation Splaying

Another variant we propose is "k-rotation splaying." We consider performing only single rotations to be a 1-rotation splay and we consider the original splay tree to employ 2-rotation splays. Since a significant performance bump was found when using double rotations, we hoped that further generalizing would result in a better constant for splay tree operations. The generalization to k-rotations is to perform maximal l-zigs and l-zags that move the accessed node higher, until we've done k rotations overall. We define l-zigs as repeatedly performing zigs from the top down, for a total of l rotations. We similarly define l-zags to be a series of l zag rotations from the top down. An example is shown in Figure 3 on the next page.

Note that the 1-zig is just a single rotation, while the 2-zig is the zig-zig case of the classic splay tree; the zig-zag case of the splay tree can be formed through a 1-zig, then a 1-zag operation.

We now provide an analysis for arbitrary k, proceeding similarly to the analysis of the original splay tree. We assign weights w_x to each node x, define the size function $s(x)$ and rank function $r(x)$ in the same way as before, and also use the same potential function $\Phi = \sum r(x)$ as before. Define x_i to be the ith ancestor of node x, and define $x_0 = x$. Now, we prove a key lemma:

Lemma 2. *For a l-ZIG operation, the amortized cost is at most $\gamma_l(r(x_l) - r(x_0))$, for all $l \geq 2$, for some constant γ_l that depends on l.*

Proof. We must show that

$$l + \sum_{i=0}^{l}[r'(x_i) - r(x_i)] \leq \gamma_l(r(x_l) - r(x_0))$$

By definition, we have that $r'(x_0) = r(x_l)$, $r(x_0) \leq r(x_i)$ and $r'(x_0) \geq r'(x_i)$ for all i. This means that the sum is upper bounded by the quantity

$$(l-1)r'(x_0) + r'(x_l) - lr(x_0)$$

and so the amortized cost is bounded by

$$l + (l-1)r'(x_0) + r'(x_l) - lr(x_0).$$

We now claim that this quantity is bounded by $\gamma_l(r'(x_0) - r(x_0))$ for a specific γ_l. If we substitute this quantity as an upper bound for the amortized cost and manipulate, we see that we need

$$\begin{aligned}&(l-1-\gamma_l)r'(x_0) + r'(x_l) + (\gamma_l - l)r(x_0)\\ &= (\gamma_l - l)(r(x_0) - r'(x_0)) + (r'(x_l) - r'(x_0)) \leq -l\end{aligned}$$

We have that $s(x_0) + s'(x_l) \leq s'(x_0)$, since the subtrees on the left hand side are disjoint, while the quantity $s'(x_0)$ includes all subtrees. To simplify notation, we now define $\frac{s(x_0)}{s'(x_0)} = q$ and $d = \gamma_l - l$. Applying the above inequality on subtrees after dividing by $s'(x_0)$ gives that $\frac{s'(x_l)}{s'(x_0)} \leq 1 - q$.

This means that the left hand side of the inequality is bounded above by $d \lg q + \lg(1-q)$. This function achieves a maximum when $q = \frac{d}{d+1}$, in which case the value is $d \lg \frac{d}{d+1} + \lg \frac{1}{d+1} = \lg \frac{d^d}{(d+1)^{d+1}}$. If this value is less than $-l$, then we must have $f(d) = \frac{(d+1)^{d+1}}{d^d} \geq 2^l$. This means that $\gamma_l = l + f^{-1}(2^l)$ is sufficient to show the claim.

□

With this lemma in hand, we can bound the cost of a k-rotation scheme:

Theorem 2. *(Generalized Access Lemma) The total amortized cost of the k-rotation splays that move node x_0 to the top of a splay tree with root t is at most $\gamma_k(r(t) - r(x)) + 1$, where $k \geq 2$, each rotation costs 1, and where $\gamma_k < k + \frac{1}{e}(2^k - 1)$.*

Proof. Above, we showed that each l-ZIG for $l \geq 2$ would have cost bounded by γ_l, and the same bounds hold for l-ZAG as well by symmetry. Now we deal with single rotations. If we have two single rotations together undertaken in succession, we get a zig-zag operation, which can be analyzed with the original Access Lemma to have a cost bound of $3(r'(x) - r(x))$ for the entire operation. Thus, the cost of any 2-rotation is bounded by $\max\{\gamma_2, 3\}(r'(x) - r(x)) = \gamma_2(r'(x) - r(x))$.

Otherwise, we analyze them as if they were combined with the following l-ZIG operation, for some l. After the

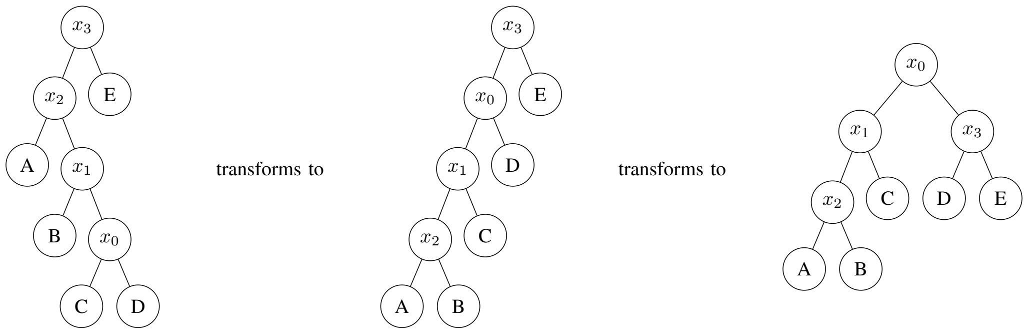

Fig. 3: An example of a 3-rotation on x_0. The transformation consists of a 2-zag, and then a 1-zig.

single rotation, the l-ZIG operation is performed with nodes $x_0, x_2 \ldots x_{l+1}$. The relation that, $r'(x_0) \geq r'(x_1)$ and $r(x_0) \leq r(x_1)$ still holds, and by a similar analysis as above, the constant in our amortized cost becomes at most γ_{l+1} for the $l+1$ rotations. Then, any zig followed by a l-ZIG, has total cost bounded by $\gamma_{l+1}(r'(x) - r(x))$.

Note that $f(x)$ is a strictly increasing function, and thus $f^{-1}(x)$ is also strictly increasing. This implies that $\gamma_k \geq \gamma_l$ for all $l \leq k$, meaning that the cost of every l-ZIG operation and their combinations with single rotations cost at most $\gamma_k(r(x_l) - r(x_0))$. If we add up the contributions from all rotations in a k-splay, the sum telescopes, and hence our overall cost is at most $\gamma_k(r(x_k) - r(x_0))$. The extra $+1$ in the lemma comes from a possible single rotation at the end of the k-splays, which cannot be combined with another l-ZIG.

Finally, note that $f(d)$ is bounded below by $ed + 1$. This can be graphically verified for small d, and since $f(d)$ itself rapidly converges to $e(d+1)$, this lower bound is also true for large d. The inverse function $f^{-1}(d)$ must then be strictly bounded above by the inverse of $ed + 1$, or $\frac{1}{e}(d-1)$. This means that $f^{-1}(2^k) < \frac{1}{e}(2^k - 1)$, thus, $\gamma_k < k + \frac{1}{e}(2^k - 1)$ as desired.

□

Unfortunately, γ_k is exponential in k. However, one can also notice that the worst case scenario is exponentially more rare as k grows, which means that in practice the performance of k-splaying for $k > 2$ may still be acceptable. We present results for $k = 2, 3, 4$.

For $k = 1$, we do not achieve the $O(\log n)$ bound that we do for the other k-rotation schemes, since there is no zig or zag operation that we can associate the individual 1-rotations with, and the proof above breaks down. In fact, we can prove the following for a 1-rotation scheme:

Lemma 3. *Given a binary search tree with n nodes with keys $1 - n$, where n is the root node and $i - 1$ is the left child of node i for all $2 \leq i \leq n$, the adversarial input that involves querying the numbers 1 through n in increasing order repeatedly results in an amortized bound of $\Omega(n)$.*

Proof. We claim through induction on k that the structure after splaying the nodes $1 - k$ in order consists of k as the root of the splay tree, n as the right child of k, and $i-1$ being the left child of i for all $2 \leq i \leq n$, $i \neq k+1$, where $1 \leq k \leq n-1$. The base case $n = 1$ is satisfied, as it is easy to see that each rotation performed is a right rotation, which means that the current right child of node 1 after any number of rotations will become the new left child of the next right child of 1. This ends up making 1 the root node, with the nodes n through 2 still forming a chain.

Now for the inductive step. Suppose the inductive hypothesis holds for some k. We now demonstrate that the hypothesis holds for $k+1$. Consider the tree after splaying the node $k+1$ - first, similar to above, we can see that the first $n - k - 1$ rotations transform the tree into that where k is the root, $k+1$ is the right child of 1, which itself has a right child of n, and all other i are the left child of $i + 1$ (for $i \neq k, k+1, n$). Then, we note that there is a final left rotation from $k + 1$ to k, which just makes k the left child of $k + 1$ and $k + 1$ the new root node (this is because at this point, $k + 1$ has no left children). Thus, we have a resulting tree where $k + 1$ is the root node, n is the right child of $k + 1$, and $i - 1$ is the left child of i for all $2 \leq i \leq n$, $i \neq k+2$, satisfying the inductive hypothesis.

Finally, to complete the proof of our original lemma, we can see that the total cost of splaying node k for each k is $n-k+1$, for $2 \leq k \leq n-1$ (it only costs $n-1$ for $k = 1$), as the node k must rotate with all $n-k$ nodes whose keys are greater than it, as well as the node $k - 1$. Moreover, we can see that for splaying node n to the top costs 1, as it involves only a single rotation; and the final structure after this splay is precisely the structure that we had started with. This means that the average cost of such a sequence would be $\frac{1}{n}\sum_{i=1}^{n}(n-k+1)-1 = \Omega(n)$, as desired. □

Nevertheless, we include this scheme in our experimental tests as a baseline.

IV. Randomized Schemes

In this section, we explore possible improvements to splay trees through various randomization patterns, and generalize them to k-rotations. Schemes I and II have been previously reported for 2-rotations, while, to our knowledge, Scheme III is new.

A. Scheme I

We first analyze the classic scheme where after accessing a node, we perform a k-rotation splay with probability p and do not splay with probability $1-p$. The scheme was first proposed in [2]. The intuition behind this scheme is that splaying is expensive, and so we want to avoid it if we are accessing a rare element. On the contrary, for an element that we access many times, we will splay the element in expectation, thus maintaining the property that commonly-accessed elements are at the top of splay trees.

We prove a variant of the Access Lemma by showing that the amortized cost of an access operation of node x at a tree with root t is bounded by $\gamma_k(r(t)-r(x))$, for the new potential function $\Phi = \frac{1}{p}\sum r(x)$ across all nodes x in the tree, where $\gamma_k < k+\frac{1}{e}(2^k-1)$ is the constant associated with k-rotations. If we let r_i be the real cost of operation i, and $p\Delta\Phi$ be the expected change in potential of the tree, we note that the amortized cost of the operation is bounded by:

$$c_i + p\Delta\Phi \leq \gamma_k(r(t) - r(x)) + 1$$

Note that the change in potential function means that the amortized cost bound changes inversely with the probability of splaying, with no $O(\log n)$ bound holding for $p = 0$, as expected. For a more thorough analysis, we refer the readers to [3].

B. Scheme II

We now analyze the scheme where we perform a k-rotation splay with probability p and splay its parent with probability $1 - p$. This scheme was first detailed in [4], however, only a surface-level analysis was provided. The intuition behind this is that we can further organize and balance the data structure by sometimes splaying a node's parent, while at the same time also raising the height of the accessed element by at least half of its depth, similar to even splaying. The paper claimed that this modification kept the same amortized bound as splay trees, but in practice performed better. We present a more thorough analysis in this section, using the same potential function as in the original analysis $\Phi = \sum r(x)$ across all the nodes of the tree. Note that using this potential function, we apply the Access Lemma to show that the expected cost of splaying a node x with parent p_x in a tree with root t is bounded by:

$$p \cdot (\gamma_k(r(t) - r(x)) + 1) + (1 - p) \cdot (\gamma_k(r(t) - r(p_x)) + 1) \\ \leq \gamma_k(r(t) - r(x)) + 1$$

as we note that $r(p_x) \geq r(x)$. This is exactly the same as the access lemma for the original splay tree, which means that the expected amortized cost of these operations is $O(\log n)$, as desired.

C. Scheme III

We now analyze a scheme where we first set the current node to be x. Then, at each step, randomly choose to perform a k-rotation with probability p or not to splay. If the current node is not splayed, splay the k^{th} ancestor instead. The idea behind this scheme is that we can save on a number of rotation operations by skipping some nodes on the way to the root. This still maintains the effect of decreasing the distance from an element to the root by a constant factor, while reducing the number of operations overall.

For the analysis of this scheme, we change the potential function to that of Scheme I, where we have $\Phi = \frac{1}{p}\sum r(x)$, and complete a similar analysis. Again, we let c_i denote the real cost of the Access operation and splays, let $\mathbb{E}[\Delta\Phi]$ denote the expected change in the potential function, and let $\gamma_k < k + \frac{1}{e}(2^k - 1)$ be the constant associated with k-rotations. We note that for a particular k-rotation, the sum of p times the change in potential and the real cost of a splay operation is bounded by $\gamma_k(r(z) - r(x))$. Thus, because each particular k-rotation in the sequence has an independent probability of p of rotating, we note that the expected change in potential is bounded by $\gamma_k(r(t) - r(x)) - c_i$, through computing the resultant telescoping series. The amortized bound still remains $\gamma_k(r(t) - r(x)) = O(\log n)$ expected.

V. Experimental Results

A. Methodology

We designed 5 different test suites in C++ for each of our splay tree variants above, which can be found on GitHub[1]. These test suites include 3 randomized test suites:

1) Inserts/accesses following a discrete uniform distribution over the integers from 0 to $n-1$. Here, we first insert all the elements in a random order, access all elements in a random order, and then delete all elements in a random order.
2) Inserts/accesses following the same procedure as above, with a distribution governed by Zipf's law. Zipf's law had originated in quantitative linguistics, stating that the frequency of a word in a linguistic corpus is inversely proportional to its rank in its frequency table. However, it has been shown that the law also applies to many other things in the real world, like city population [5], income rankings, music [6] and so on. We use Zipf's law by assigning the i^{th} item we insert a probability of

$$f(i) = \frac{1}{i^s H_{N,s}}$$

of being be accessed where $H_{N,s}$ is the N^{th} generalized harmonic number of order s, and where s is an exponent that has to be tuned for each distribution depending on the context. We choose $s = 1.07$, which has been shown to model city population well in [5].
3) Tree starts off as a randomly generated tree with n nodes and an initial depth of n. We access the maximum depth node on each access. We decided to include this test as a more general worst case than those included in our deterministic test suites, with the set of trees to choose from equivalent to those with the maximum potential.

[1] https://github.com/zhiweigan/randomized-splay-trees

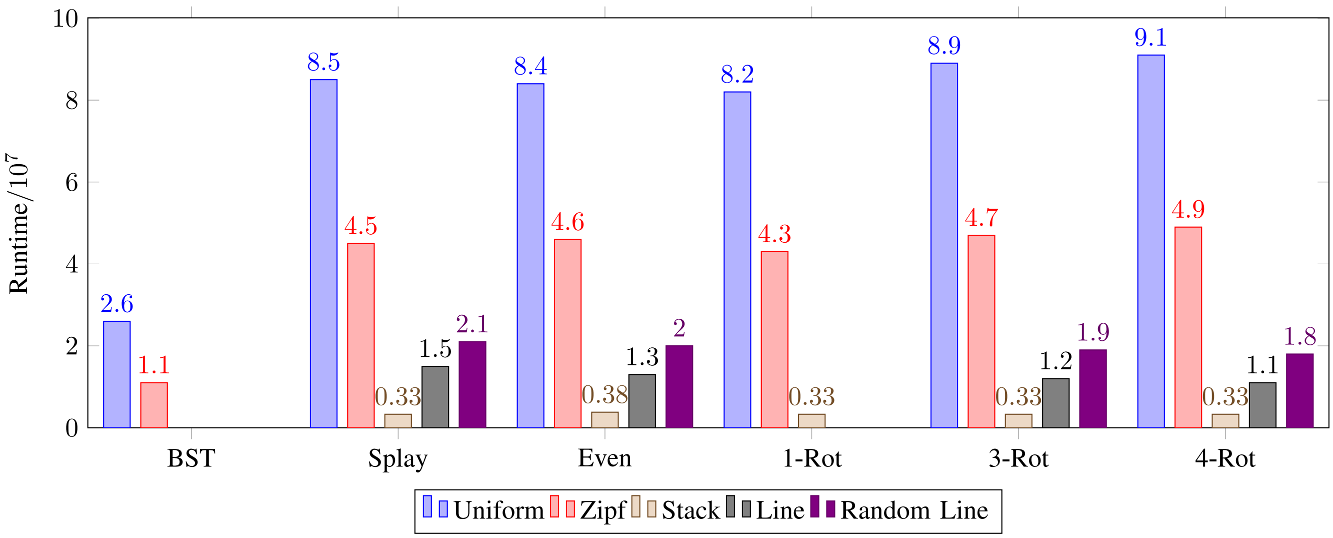

Fig. 4: Runtime of Deterministic Schemes. Note that the performance of the BST and 1-rotation splay trees on some adversarial inputs were omitted, as they exceeded the time limit.

and 2 deterministic test suites:

1) Stack: We insert key values in order from 0 to $n-1$ and access key values from $n-1$ to 0. We call this Stack because the normal splay tree will treat this like a Stack, with the last elements inserted being the first elements accessed, and the sequence of actions this data structure performs can be easily proven to have an $O(1)$ average cost. This sort of acts as an "upper bound" to the performance of the splay tree.
2) Line: The tree starts off like a line with only right children, and we access the node with key $n-i$ on the ith access. We chose this test because this serves as a worst-case scenario for 1-rotations, while also allowing us to compare how varying the value of k in terms of k-rotations deal with this worst-case scenario.

For each test, we tabulated the number of single rotations performed as well as the number of followed pointers. For our implementation, the cost for performing single rotations was experimentally found to be 2.3× the cost of following pointers, and so multiplying the number of single rotations by 2.3 and adding to the number of followed pointers resulted in a combined cost that we used to report our results.[2]

B. Deterministic Schemes

Here is a tabulation of our results for $N = 10^6$ accesses for all of the deterministic schemes. We include the results for a basic binary search tree (with no balancing) and a regular splay tree with no modifications as baseline comparisons.

We excluded results of BST and 1-rotation splay trees from the Line, Random Line, and Stack (for the former) cases because they exceeded 2^{31} instructions.

Tree Type	BST	Splay	Even	1-Rot	3-Rot	4-Rot
Uniform	2.58e7	8.54e7	8.40e7	8.19e7	8.86e7	9.10e7
Zipf	1.14e7	4.51e7	4.62e7	4.31e7	4.70e7	4.86e7
Stack	**TLE**	3.30e6	3.80e6	3.30e6	3.30e6	3.30e6
Line	**TLE**	1.46e7	1.35e7	**TLE**	1.21e7	1.11e7
Rand. Line	**TLE**	2.07e7	2.03e7	**TLE**	1.86e7	1.83e7

Note: **TLE** stands for Time Limit Exceeded

We note that for the deterministic trees with random input distributions, the basic binary search tree actually outperforms all splay trees. The reason for this is that the overhead cost for splaying on an already random input outweighs the benefits we get from splaying, as a random input on a binary search tree is already somewhat balanced. For the adversarial input distributions, increasing the number of rotations we do each splay step improves performance. This makes sense, as we need bigger restructuring changes to bring the adversarial

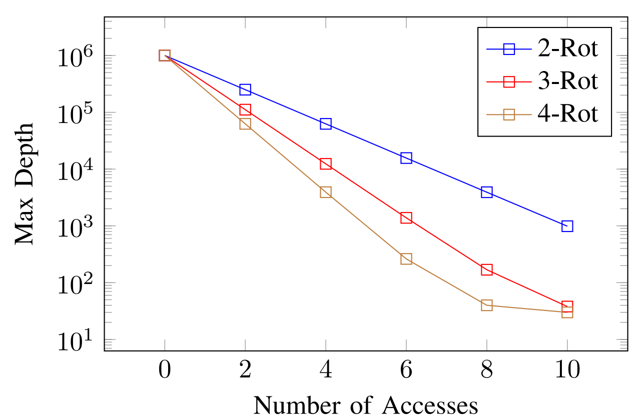

Fig. 5: Comparison of Maximum Tree Depth Every 2 Accesses for 2-Rot, 3-Rot, 4-Rot on the Line Test

[2]The exact numbers and data are available at: https://github.com/zhiweigan/randomized-splay-trees

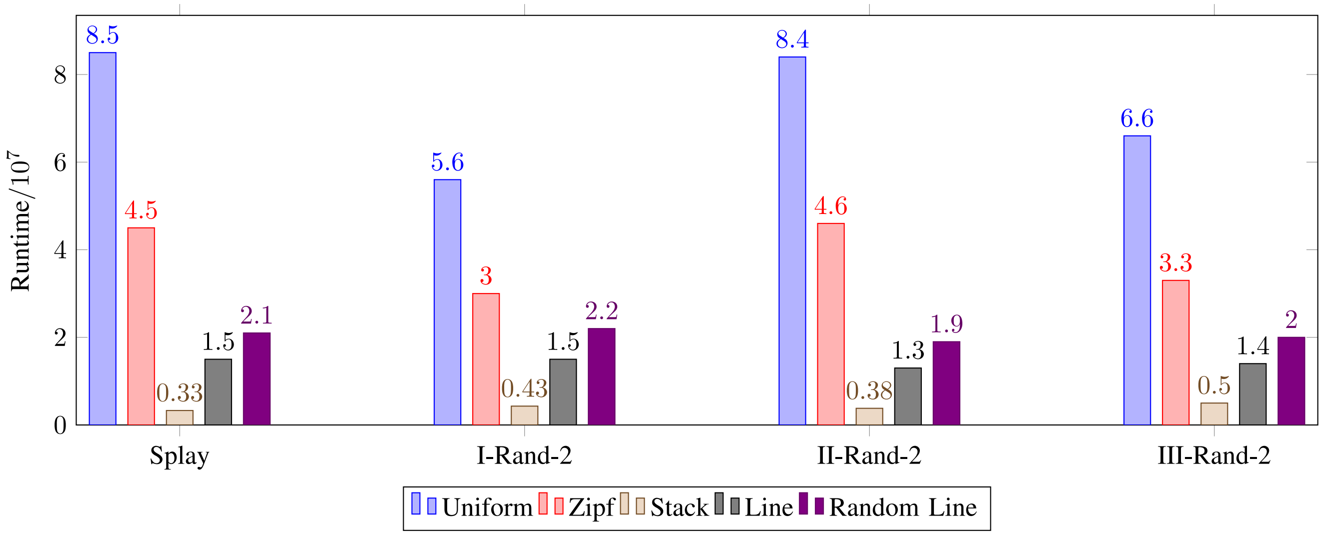

Fig. 6: Runtime of Deterministic 2-Rotation and X-Rand-2 Schemes on Deterministic Inputs

tree structure to a balanced one. In Figure 5, we present a comparison of the maximum tree depth on the Line test for our k-Rotation schemes. We limit our graph to the first few accesses since the depth of the tree rapidly grows afterwards on the Line test, since we make another branch with $\Omega(k)$ height after k accesses. The chart shows that the slope becomes more negative as we increase k, confirming our hypothesis.

In addition, we get a marginal speedup from Even Splaying on the Uniform and (Random) Line tests, but worse results on the Zipf and Stack tests. While we initially investigated this scheme hoping that we would get a speedup from removing the single zig rotation at the end, it seems that the actual speedup comes from another cause. This is apparent when comparing our results here to those of the randomized II-Rand-2 scheme, which was shown to get nearly identical results while splaying either itself or the parent randomly.

We hypothesize that this change in even splaying is simply due to a different element being moved to the root. It is unlikely that the same element would be accessed again in the uniform test, and impossible in the deterministic tests. On the contrary, elements have different access probabilities in the Zipf test, thus splaying the parent of a highly-accessed element would be detrimental. For the Stack test, we perform 50% more pointer traversals but have the same number of rotations as the regular splay tree. This makes sense as well, since the normal splay tree would perform single rotations and always be able to access the next element in 1 traversal, while even rotations would require 1 rotation and 1 more pointer traversal 50% of the time.

C. Randomized Schemes

Here is a tabulation of our results for $N = 10^6$ accesses, and $p = 2^{-1}$ with the same seed for all randomized schemes. The X-Rand-k columns show *Scheme X* applied to the various k-rotation deterministic splay trees.

Results from Scheme I

Tree Type	I-Rand-1	I-Rand-2	I-Rand-3	I-Rand-4
Uniform	5.34e7	5.57e7	5.77e7	5.92e7
Zipf	2.85e7	2.98e7	3.10e7	3.21e7
Stack	4.30e6	4.30e6	4.30e6	4.30e6
Line	**TLE**	1.50e7	1.21e7	1.10e7
Random Line	**TLE**	2.17e7	1.94e7	1.89e7

Results from Scheme II

Tree Type	II-Rand-1	II-Rand-2	II-Rand-3	II-Rand-4
Uniform	8.05e7	8.39e7	8.71e7	8.93e7
Zipf	4.41e7	4.61e7	4.81e7	4.97e7
Stack	3.80e6	3.80e6	3.80e6	3.80e6
Line	**TLE**	1.33e7	1.15e7	1.09e7
Random Line	**TLE**	1.93e7	1.80e7	1.80e7

Results from Scheme III

Tree Type	III-Rand-1	III-Rand-2	III-Rand-3	III-Rand-4
Uniform	6.59e7	6.16e7	6.26e7	6.41e7
Zipf	3.59e7	3.33e7	3.40e7	3.51e7
Stack	5.30e6	4.97e6	4.75e6	4.62e6
Line	**TLE**	1.40e7	1.29e7	1.25e7
Random Line	**TLE**	2.02e7	1.94e7	1.95e7

We can see that Scheme II has the best results for the adversarial test cases, and Scheme I has the best results for the randomized test cases when $p = 2^{-1}$. Comparing to the deterministic splay tree results, the randomized inputs run approximately 35% faster with Scheme I, marginally faster with Scheme II, and about 25% faster with Scheme III. On the adversarial inputs, we see smaller gains, with Scheme I and III's runtime being similar to the standard splay tree, while Scheme II is faster by approximately 10%. We were unable to replicate the 25% speedup of Scheme II claimed in [4].

In addition, the comparisons across schemes align well with our intuition. For adversarial inputs, significant restructuring is necessary, and this explains our results for Schemes I and

III, which both choose not to splay/rotate with a nontrivial chance. This effectively forces the splay tree to complete more expensive accesses in these inputs, and thus causes the runtime to be worse than that of Scheme II. Curiously, the performance matches that of deterministic splay trees - but this could be due to the fact that rotations are useful in the very beginning but not so necessary or wasteful after some small number of accesses. Scheme II exceeds the performance of the former by significant margins, because after the initial few accesses, the significant restructuring that is guaranteed by Scheme II is not completely necessary. However, for the randomized inputs, we see that in many cases it becomes optimal to not splay certain nodes, or in general forego splaying, as this is expensive and can potentially ruin a good tree structure. Especially in the uniformly random data, moving nodes to the top becomes less important as each node is equally likely to be chosen to be accessed; while for Zipf's Law, this effect is less prominent, but the randomness implies that on expectation, commonly accessed nodes will still move to the top.

All randomized schemes do better for the adversarial inputs as we increase k, but do poorer on the randomized inputs. This is the same result as we see in the deterministic schemes - the greater number of rotations allows for faster tree balancing, reducing overall runtime.

While we already saw significant improvements with the randomization schemes for $p = \frac{1}{2}$, we wanted to see if we could do better by varying p in our randomized schemes. Since we achieved the best results on randomized inputs when $k = 2$, we use it as a model for how these randomized schemes behave when different probabilities are used. We exclude the $k = 1$ scheme as the scheme was unable to properly handle adversarial inputs, though it was marginally faster on randomized inputs.

For Scheme I, the access times for the randomized distributions decrease by approximately 70% as the probability of splaying decreases. We expect these results because we simply reduce the number of splays, cutting down on the overhead for balancing an already-balanced tree. On the contrary, the adversarial inputs' access times grow quite quickly. This is in line with our previous observations, as we do not induce the restructuring required for good runtimes with low p.

Interestingly, the runtime increases for Zipf's distribution after p decreases from 2^{-6}. Zipf's distribution increases the probability of accessing certain elements, and splaying helps keep access runtime of commonly accessed elements low. Thus, when p decreases so much that we do not do many splays at all, Scheme I can be expected to perform slightly worse since it no longer maintains common elements directly at the root of the tree.

Another point to note is that for the Stack test, the number of rotations required for each probability are all in the range of $[10^6 - 50, 10^6]$. This means we end up having to do the same number of rotations regardless of the number of splays we end up doing. This is an intuitive result, because performing fewer splays would lead to a deeper tree which means we have to do more rotations per splay. This would also explain the rapid increase in cost as we decrease the probability of splaying, because we have to account for a deeper tree with more pointer traversals.

Scheme I (I-Rand-2)

p	Uniform	Zipf	Stack	Line	Random Line
$1-2^{-7}$	8.50e7	4.48e7	3.31e6	1.46e7	2.07e7
$1-2^{-6}$	8.45e7	4.46e7	3.32e6	1.46e7	2.07e7
$1-2^{-5}$	8.36e7	4.41e7	3.33e6	1.46e7	2.06e7
$1-2^{-4}$	8.17e7	4.31e7	3.37e6	1.45e7	2.08e7
$1-2^{-3}$	7.80e7	4.12e7	3.44e6	1.45e7	2.05e7
$1-2^{-2}$	7.06e7	3.74e7	3.63e6	1.59e7	2.01e7
2^{-1}	5.57e7	2.98e7	4.30e6	1.50e7	2.17e7
2^{-2}	4.08e7	2.22e7	6.28e6	1.59e7	2.41e7
2^{-3}	3.34e7	1.85e7	1.03e7	2.55e7	3.34e7
2^{-4}	2.97e7	1.69e7	1.82e7	2.66e7	4.41e7
2^{-5}	2.78e7	1.63e7	3.40e7	7.31e7	1.40e8
2^{-6}	2.69e7	1.62e7	6.58e7	8.68e7	3.19e8
2^{-7}	2.64e7	1.64e7	1.29e8	2.07e8	4.67e8

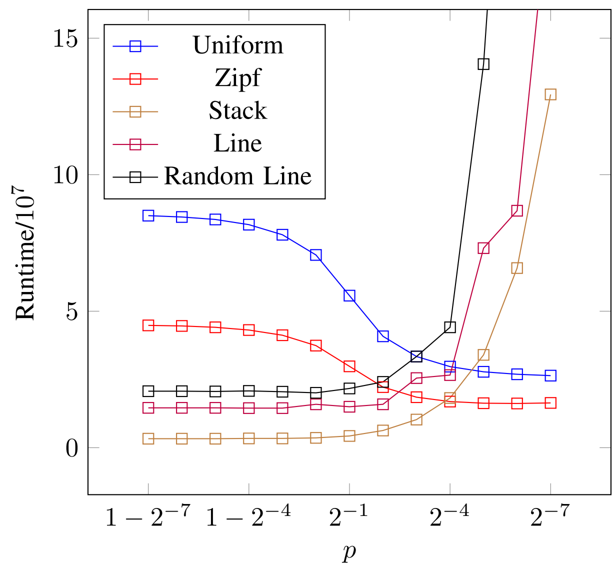

Fig. 7: Runtime of I-Rand-2 Scheme with Varying Probability.

When the probability of splaying increases from $\frac{1}{2}$, we converge to the runtime of a standard splay tree, as expected. The runtime on the Uniform and Zipf tests increasing significantly, while the runtime on other tests have few fluctuations. This implies that, as seen before, we are still splaying too much for randomized inputs, while we only get marginal benefit from splaying more often in the adversarial inputs.

When decreasing the probability for Scheme II, we saw a decrease in the runtime for the Uniform, Random Line, and Line tests, and an increase in the runtime for the Zipf and Stack tests. This is the only scheme for which decreasing the probability of doing a normal splay increases performance for the Line test. In fact, it decreases the runtime at low probability so much that it is able to match the best deterministic algorithm 4-Rot's runtime for this test case. Unfortunately, the other performance changes are marginal, with a max deviation of approximately 20% across the entire probability range.

Similar to Scheme I, as p increases, we converge to the

Scheme II (II-Rand-2)

p	Uniform	Zipf	Stack	Line	Random Line
$1-2^{-7}$	8.54e7	4.51e7	3.31e6	1.46e7	2.07e7
$1-2^{-6}$	8.54e7	4.51e7	3.32e6	1.45e7	2.07e7
$1-2^{-5}$	8.53e7	4.51e7	3.33e6	1.45e7	2.06e7
$1-2^{-4}$	8.52e7	4.51e7	3.36e6	1.45e7	2.06e7
$1-2^{-3}$	8.51e7	4.52e7	3.43e6	1.44e7	2.04e7
$1-2^{-2}$	8.47e7	4.54e7	3.55e6	1.41e7	2.00e7
2^{-1}	8.39e7	4.61e7	3.80e6	1.33e7	1.93e7
2^{-2}	8.31e7	4.77e7	4.05e6	1.23e7	1.85e7
2^{-3}	8.27e7	4.92e7	4.17e6	1.17e7	1.81e7
2^{-4}	8.25e7	5.04e7	4.24e6	1.14e7	1.79e7
2^{-5}	8.24e7	5.12e7	4.27e6	1.12e7	1.78e7
2^{-6}	8.24e7	5.17e7	4.28e6	1.12e7	1.77e7
2^{-7}	8.24e7	5.21e7	4.29e6	1.11e7	1.77e7

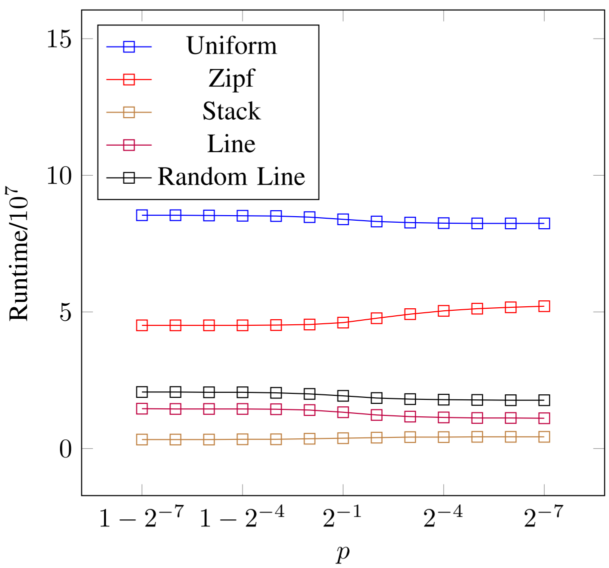

Fig. 8: Runtime of II-Rand-2 Scheme with Varying Probability.

Scheme III (III-Rand-2)

p	Random	Zipf	Stack	Line	Random Line
$1-2^{-7}$	8.51e7	4.49e7	3.32e6	1.43e7	2.03e7
$1-2^{-6}$	8.47e7	4.47e7	3.33e6	1.40e7	2.00e7
$1-2^{-5}$	8.39e7	4.43e7	3.36e6	1.39e7	1.97e7
$1-2^{-4}$	8.24e7	4.36e7	3.43e6	1.37e7	1.94e7
$1-2^{-3}$	7.94e7	4.21e7	3.57e6	1.35e7	1.91e7
$1-2^{-2}$	7.34e7	3.91e7	3.90e6	1.34e7	1.91e7
2^{-1}	6.16e7	3.33e7	4.97e6	1.40e7	2.02e7
2^{-2}	4.99e7	2.77e7	8.00e6	1.77e7	2.54e7
2^{-3}	4.42e7	2.52e7	1.40e7	2.50e7	3.66e7
2^{-4}	4.13e7	2.43e7	2.60e7	3.88e7	5.87e7
2^{-5}	3.99e7	2.43e7	5.00e7	6.47e7	1.02e8
2^{-6}	3.93e7	2.48e7	9.80e8	1.15e8	1.87e8
2^{-7}	3.89e7	2.55e7	1.94e8	2.13e8	3.57e8

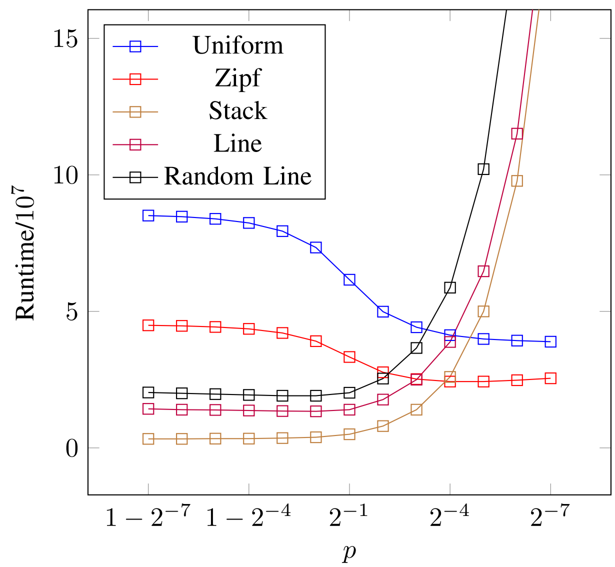

Fig. 9: Runtime of III-Rand-2 Scheme with Varying Probability.

runtimes of the classic splay tree, while as p decreases, we expect, similar to Even Splaying, a tree that performs better on distributions that are more uniform and worse on distributions that have a few, commonly accessed elements (such as the test based on Zipf's Law). We actually end up outperforming Even Splaying on the Line, Uniformly Random, Stack, and Random Line tests by a small factor, which could be attributed to the increased flexibility of this strategy with being able to choose both options (splaying itself and splaying its parent).

We conclude that Scheme II works best against adversarial inputs (because, unlike the other schemes, we are splaying on every step) and passably against randomized inputs (due to the overhead cost). This is apparent in the graphs, where we see no divergence as p decreases unlike in Schemes I and III.

Scheme III produces similar outputs to Scheme I for both the deterministic and randomized test suites. The sharp increase in cost for the Stack, Line, and Random Line tests as we increase the probability is expected, as in Scheme I, because we are not performing as many rotations as we should be for adversarial test cases. Similarly, the convergence to a standard splay tree as $p \to 1$ is also apparent.

Comparing with our results from Scheme I, the table above shows that performing random rotations along the splay path is worse at balancing the tree for all cases aside from Random Line. Interestingly, Random Line achieves a better result for Scheme III than Scheme I for small p, even though Line performs worse for Scheme III than Scheme I. This can intuitively be explained by the idea that performing random rotations can sometimes be more effective at balancing a randomly shaped tree.

VI. Conclusions

In this paper, we explored a variety of proposed deterministic and random schemes that on average performed around the same as that of the original Splay Tree that was proposed by Sleator and Tarjan, but performed better relative to each other on different types of inputs. On randomized inputs, we interestingly see that the data structures that have the worst bounds theoretically (the static Binary Search Tree and the 1-rotation splay tree) actually performed better; we can attribute the former performing better due to the lack of expensive rotations which still keeps the expected runtime on these inputs as $O(\log n)$; and for the latter, due to the lack of pointer traversals needed to check which case the node satisfies in order to splay it. However, as expected, when we tested both of these structures against the general "worst-case" scenario, which is a starting configuration of all the inputs being in a line, we see that these structures fail to complete the accesses in a reasonable number of pointer traversals.

In this same test, we found that 3-rotation and 4-rotation trees performed better in these worst case scenarios, compared to normal 2-rotation splay trees. In order to explore this further, we looked at the maximum depth of the trees as elements were accessed. We found that the maximum depths shrank much quicker for higher values of k; however, near the end, while the depths for $k = 3$ and $k = 4$ flatlined, the maximum depth for $k = 2$ continued to shrink below that of these higher k values. This intuitively made sense, especially with the higher constant factor bounds that we got with higher k; highlighting the tradeoff that different values of k would have - in place of average performance, these higher k values had a higher resilience in terms of reaching more optimal configurations quicker than the vanilla splay trees. Thus, we see this work can be useful for applications where almost-sorted inputs need to be accessed quickly, or other instances where the initial tree constructed is largely unbalanced.

In terms of adding randomization, we found that Scheme II, a randomization scheme that chooses to splay with probability p or splay its parent with probability $1 - p$, was able to significantly improve upon the runtime of standard splay trees in adversarial cases. Then, we found that Scheme I, a randomization scheme that splays with probability p or not splay with probability $1-p$, did better than the standard splay tree on randomized inputs, but failed to do better than the binary search tree.

For future work, it would be worthwhile to explore the properties of normal splay trees that generalize to k-rotation splay trees. One could note that the Access Lemma variants that were proved in the various sections can show that all such variants satisfy Static Optimality. Another direction would be to formulate more randomized schemes, or taking other existing schemes that were proposed in papers and performing the same sort of testing and analysis presented in this paper. A third direction would be to combine the randomized strategies of splaying such that they would all be more resilient to an adversarial input. Finally, while we proved that all schemes for k-rotations run in (expected) $O(\log n)$, future work could go into better bounding the constant factors γ_k. These grew exponentially in k, but in practice, k-rotation splay trees for $k = 3, 4$ were quite competitive with their $k = 2$ counterpart, and sometimes even outperformed them.

VII. Acknowledgements

We would like to thank Richard Wang, Catherine Wu, and Thanadol Chomphoochan for their constructive comments on drafts of this paper. We would also like to thank Ben Eysenbach and Robi Bhattacharjee for providing us access to their 18.416 Project entitled 'Randomized Splay Trees,' which we had referred to and built off of. Finally, we would like to thank David Karger and all the TAs - Josh Brunner, Thiago Bergamaschi, and Christian Altamirano - for their excellent and thorough teaching in 18.415.

References

[1] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, p. 652–686, Jul. 1985. [Online]. Available: https://doi.org/10.1145/3828.3835

[2] S. Albers and M. Karpinski, "Randomized splay trees: Theoretical and experimental results," *Information Processing Letters*, vol. 81, no. 4, pp. 213 – 221, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019001002307

[3] R. Bhattacharjee and B. Eysenbach, "Randomized splay trees 6.856 final project," 2016.

[4] M. Furer, "Randomized splay trees," Jan. 1999, pp. S903–S904, proceedings of the 1999 10th Annual ACM-SIAM Symposium on Discrete Algorithms ; Conference date: 17-01-1999 Through 19-01-1999.

[5] N. K. Vitanov and M. Ausloos, "Test of two hypotheses explaining the size of populations in a system of cities," *Journal of Applied Statistics*, vol. 42, no. 12, pp. 2686–2693, 2015. [Online]. Available: https://doi.org/10.1080/02664763.2015.1047744

[6] D. H. Zanette, "Zipf's law and the creation of musical context," *Musicae Scientiae*, vol. 10, no. 1, pp. 3–18, 2006. [Online]. Available: https://doi.org/10.1177/102986490601000101