Randomized Splay Trees: Theoretical and Experimental Results

Susanne Albers* Marek Karpinski†

Abstract

Splay trees are self-organizing binary search trees that were introduced by Sleator and Tarjan [12]. In this paper we present a randomized variant of these trees. The new algorithm for reorganizing the tree is both simple and easy to implement. We prove that our randomized splaying scheme has the same asymptotic performance as the original deterministic scheme but improves constants in the expected running time. This is interesting in practice because the search time in splay trees is typically higher than the search time in skip lists and AVL-trees. We present a detailed experimental study of our algorithm. On request sequences generated by fixed probability distributions, we can achieve improvements of up to 25% over deterministic splaying. On request sequences that exhibit high locality of reference, the improvements are minor.

*Lehrstuhl Informatik II, Universität Dortmund, 44221 Dortmund, Germany. Part of this work was done while at the Max-Planck-Institut für Informatik, Saarbrücken, Germany. Email: `albers@ls2.cs.uni-dortmund.de`

†Department of Computer Science, University of Bonn, 53117 Bonn. E-mail: `marek@cs.uni-bonn.de`

1 Introduction

Splay trees are self-organizing binary search trees that were introduced by Sleator and Tarjan [12]. They represent an elegant solution to the well-known *dictionary problem*, where we have to maintain a set S of elements under a sequence of *operations*. We assume that every element $x \in S$ consists of a key $k(x)$, which is drawn from a totally ordered universe, and some additional information. Each operation is either (a) an *access* to an element in S, (b) an *insertion* of a new element into S, or (c) a *deletion* of an element from S.

As all binary search trees, splay trees store the elements $x \in S$ in the nodes of the tree, each node holding exactly one element. The important feature of splay trees is that they do not maintain any height or balance constraint but rather have a restructuring rule that modifies the tree after each operation. More precisely, after each access to an element $x \in S$, the node storing x is moved to the root of the tree using a special sequence of rotations that depends on the structure of the access path. This reorganization of the tree is called *splaying*.

Sleator and Tarjan [12] analyzed splay trees and proved a series of interesting results. They showed that the amortized asymptotic time of access and update operations is as good as the corresponding time of balanced trees. More formally, in an n-node splay tree, the amortized time of each operation is $O(\log n)$. It was also shown [12] that on any sequence of accesses, a splay tree is as efficient as the optimum static search tree. Moreover, Sleator and Tarjan [12] presented as series of conjectures, some of which have been resolved or partially resolved [3, 4, 5, 15]. On the other hand, the famous splay tree conjecture is still open: It is conjectured that on any sequence of accesses splay trees are as efficient as any dynamic binary search tree. We refer the reader to [2, 8, 10, 13, 14] for further work on splay trees.

Splay trees have the advantage that they need no knowledge of the properties of the input sequence, but adapt automatically to best suit the input. Moreover, the access and update operations are very simple and easy to implement. However, splay trees have the disadvantage that they do rotations even during access operations. This leads to a high overhead. In fact, in practice, the access time in splay trees is generally higher than the access time in AVL trees [1] and skip lists [11]. On insert and delete operations, splay trees perform well.

In this paper we present the first randomized variant of splay trees. Our work is motivated by the goal of reducing the access time in splay trees and, thus, bringing splay trees closer to practice. Our randomized splaying scheme for reorganizing the tree is very simple and easy to implement. Existing splay tree codes can be adapted with very little extra work.

The randomized restructuring algorithm, called *Rand-Splay*(p), is presented in Section 4 and works for any fixed probability $p \in [0, 1]$. During each access operation, *Rand-Splay*(p) only splays the tree with probability p. Intuitively, this reduces the number of rotations performed on a sequence of accesses while, if p is not too small, frequently accessed elements are still close to the root of the tree.

In order to compare *Rand-Splay*(p) to deterministic splay trees, we first have to introduce a refined cost model (Section 3). We assume that during each access operation, following an edge on the access path into the tree incurs a cost of 1, whereas a rotation incurs a cost of d, $d > 0$. In our experimental tests we observed $d \approx 2.75$. In Section 4 we show that if the amortized time of an access operation is $(1+d)T$ in deterministic splay trees, then the expected amortized time in our randomized splay trees is $(1+pd)T$. More generally, we prove that many of the results shown by Sleator and Tarjan [12] for deterministic splay trees carry over to randomized splay trees but with smaller constants in the O-notation, i.e. the expected running time is smaller.

In Section 5 we present a detailed experimental study of *Rand-Splay*(p). We study its performance on various types of request sequences. We show that if requests are generated by a

fixed probability distribution, then *Rand-Splay*(p) can achieve improvements of up to 25% over deterministic splaying. On the other hand, if request sequences exhibit high locality of reference, then the improvements are almost negligible.

Independent of the work presented in this paper, Fürer [6] recently presented randomized splaying algorithms. He presented theoretical results but gave no experimental study.

2 Splay trees

We briefly review deterministic splay trees and concentrate on the access operation.

As mentioned before, splay trees are binary search trees. The elements of the set S to be represented are stored in the nodes of the tree. We assume that the keys $k(x)$, $x \in S$, are pairwise distinct. At any time, the elements in the tree T are stored in *symmetric order*, i.e., for every node u in T, the keys in the left subtree of u are smaller than the key stored at u, and the keys in the right subtree of u are greater than the key at u.

It is obvious how to access an element $x \in S$ in a binary search tree T. Starting at the root of T, we repeatedly follow edges into the tree until x is found. At every node we either choose the edge into the left or into the right subtree, depending on whether $k(x)$ is smaller or greater than the key at the current node.

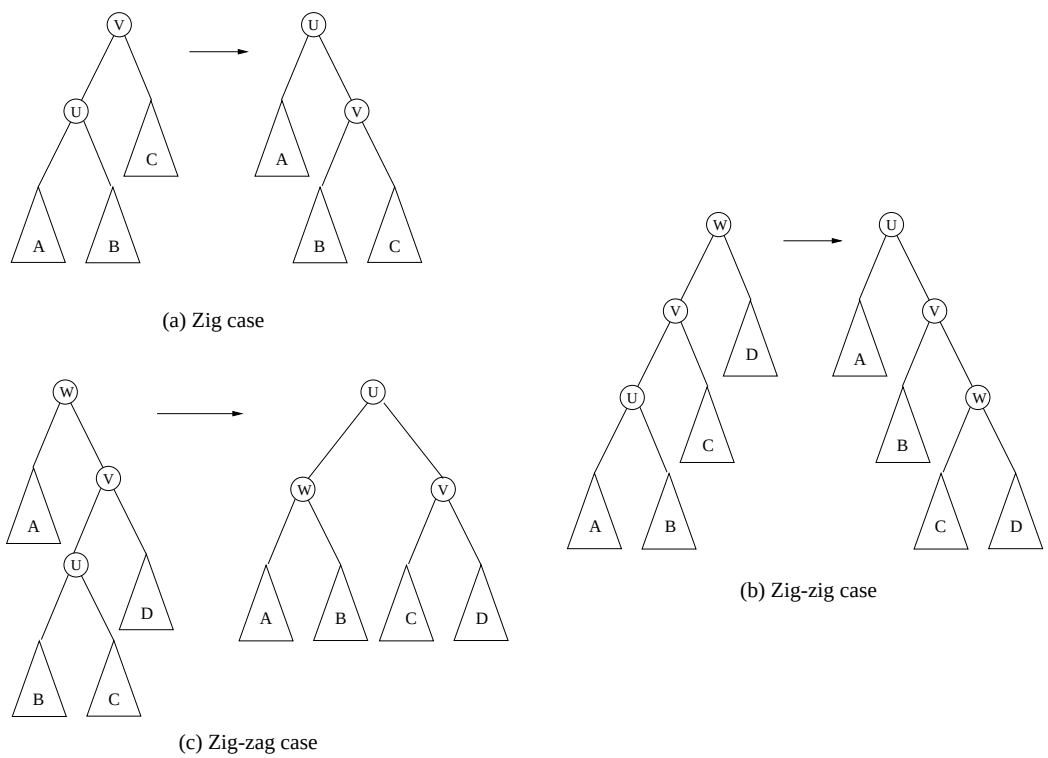

Figure 1: Three cases of splaying

In a splay tree, each time an element x is accessed, the node $u \in T$ holding x is moved to the root of T using a special sequence of rotations. These rotations are illustrated in Figure 1. There are three different cases. Each case has a symmetric variant which is not shown. (1) In the *zig* case, the parent of u, $p(u)$, is the root of T. We rotate the edge between u and $p(u)$. (2) In the *zig-zig* case, $p(u)$ is not the root and u and $p(u)$ are both left children. In this case, we

first rotate the edge between $p(u)$ and the grandparent $g(u)$. Then we rotate the edge between u and $p(u)$. (3) In the *zig-zag* case, $p(u)$ is not the root, and u is a left child and $p(u)$ is a right child. Here, we first rotate the edge between u and $p(u)$ and then the edge between u and the new $p(u)$.

Sleator and Tarjan [12] analyze the amortized time incurred on a sequence of accesses using a potential function Φ. Consider a sequence of m accesses. The amortized time t_j^a of the j-th access, $1 \leq j \leq m$, is defined as $t_j^a = t_j + \Phi(j) - \Phi(j-1)$, where t_j is the actual time of the access and $\Phi(j) - \Phi(j-1)$ is the change in potential. The total time of the entire sequence of accesses is

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (t_j^a + \Phi(j-1) - \Phi(j)) = \sum_{j=1}^{m} t_j^a + \Phi(0) - \Phi(m).$$

The potential function defined by Sleator and Tarjan is quite general and allows them to derive a series of results. More precisely, Φ is defined as follows. Each element $x \in S$ is assigned a *weight* $w(x)$. For every node $u \in T$, the *size* $\sigma(u)$ of u is the sum of the weights of all elements in the subtree rooted at u. The *rank* $\rho(u)$ of u is $\log(\sigma(u))$, where the base of the logarithm is 2. Now, $\Phi = \sum_{u \in T} \rho(u)$. Using this potential function, Sleator and Tarjan developed the following bound on the time needed by an access. Suppose that there is an access to an element x and that u is the node storing x. Then the amortized time to access x and splay T at u is $1 + 3(\rho(r) - \rho(u))$, where r is the root of T.

3 A refined cost model

In order to compare the performance of randomized splay trees to that of deterministic splay trees we have to introduce a refined cost model. We restate the most important results from [12] in this model. We avoid O-notation, used in [12], and explicitly give the hidden constants.

When analyzing the time of an access to x, Sleator and Tarjan do not distinguish between the time needed to search for x in the tree and the time needed to splay the tree at the node u holding x. They assume that whenever an edge $e = (v, w)$ in the tree is rotated, the rotation time also includes the time incurred in traversing e to continue search in the subtree rooted at w; here w is the child of v. Thus, a zig case costs 1 time unit, whereas a zig-zig or a zig-zag case costs 2 time units. In the randomized splaying scheme that we will develop in the next section, we will reduce the number of rotations, i.e., a node holding the accessed element is not always moved to the root of the tree. Thus, we have to distinguish between search and rotation time. We assume that a traversal of an edge during a search incurs a cost of 1 and that a rotation of an edge incurs a cost of d, $d > 0$. Using this cost model, a zig case in Sleator and Tarjan's deterministic splaying scheme costs $1 + d$, and a zig-zig or zig-zag case costs $2(1 + d)$.

Now, let

$$\Phi = (1 + d) \sum_{u \in T} \rho(u).$$

Lemma 1 *Suppose that there is an access to an element x and that $u \in T$ is the node holding x. Then the amortized time to access x and splay the tree at u is at most $(1+d)+3(1+d)(\rho(r) - \rho(u)) = (1 + d) + 3(1 + d)\log(\sigma(r)/\sigma(u))$, where r is the root of the tree.*

This lemma can be shown in the same way as the corresponding lemma in [12]. Using Lemma 1, it is possible to derive a series of theorems. These theorems can be proved in the same way as in [12]. For each theorem we give a brief sketch of the proof that demonstrates how the overall

change in potential $\Phi(0) - \Phi(m)$ affects the performance bounds. This will be crucial in the next section. In the following theorems C always reflects $\Phi(0) - \Phi(m)$.

Consider a sequence of m accesses, and suppose that the tree contains n elements $x_1, \ldots, x_n$. Recall that $w(x_i)$ is the weight of x_i. Note that $\Phi(0) - \Phi(m) \leq (1+d)\sum_{i=1}^{n} \log(W/w(x_i))$, where $W = \sum_{i=1}^{n} w(x_i)$.

Theorem 1 *The total access time is at most* $(1+d)(3m\log n + m) + C$, *where* $C = (1+d)n\log n$.

Proof: Assign a weight of $1/n$ to each element x_i. Then, $W = 1$. Each access needs an amortized time of $(1+d) + 3(1+d)\log n$ and $\Phi(0) - \Phi(m) \leq (1+d)n\log n = C$. □

The next theorem is called the *static optimality theorem* [12]. It implies that on any sequence of accesses, the time needed by a splay tree is no more than a constant multiple of the time needed by an optimum static search tree. For any element x_i, let $q(x_i)$ be the number of times x_i is accessed in the entire sequence.

Theorem 2 *If every element is accessed at least once, then the total access time is at most* $(1+d)(3\sum_{i=1}^{n} q(x_i)\log(m/q(x_i)) + m) + C$, *where* $C = (1+d)\sum_{i=1}^{n}\log(m/q(x_i))$.

Proof: Assign a weight of $q(x_i)/m$ to each element x_i. The change in potential is at most $(1+d)\sum_{i=1}^{n}\log(m/q(x_i)) = C$. □

Theorem 3 is the *static finger theorem*. Let x_{i_j} be the element accessed during the j-th access in the sequence.

Theorem 3 *If f is any fixed element, the total access time is at most* $(1+d)(6\sum_{j=1}^{m}\log(|k(x_{i_j}) - k(f)+1|) + 7m) + C$, *where* $C = (1+d)2n(\log n + 1)$.

Proof: Assign a weight of $1/(|k(x_i) - k(f)+1|)^2$ to element x_i. Then $W \leq 2\sum_{i=1}^{\infty} 1/i^2 = \pi^2/3$. The amortized time of the j-th access is $(1+d) + 3(1+d)(2\log(|k(x_{i_j}) - k(f)+1|) + \log W) \leq (1+d)(6\log(|k(x_{i_j}) - k(f)+1|) + 7)$. The change in potential is $(1+d)\sum_{i=1}^{n}\log(W/w(x_i)) \leq (1+d)(n\log W + \sum_{i=1}^{n}\log(1/w(x_i)) \leq (1+d)(2n\log(\pi/3) + 2n\log n) \leq (1+d)2n(\log n + 1)$. The second inequality follows because $w(x_i) \geq 1/n^2$. □

In the *working set theorem* below, $t(j)$ is the number of different elements accessed before the j-th access since the last access to element x_{i_j}.

Theorem 4 *The total access time is at most* $(1+d)(6\sum_{j=1}^{m}\log(t(j)+1) + 4m) + C$, *where* $C = (1+d)(2n\log n + n)$.

Proof: (Sketch) At any time, the weights assigned to the n elements form a permutation of $1, 1/4, 1/9, \ldots, 1/n^2$. Initially the weight $1, 1/4, 1/9, \ldots, 1/n^2$ are assigned to the elements in the order of first access. The element accessed first gets weight 1. During the processing of accesses, the weights are reassigned. Whenever an element x with weight $1/i^2$ is accessed, x gets weight 1 and every element y with weight $1/(i')^2$, $i' < i$, gets weight $1/(i'+1)^2$. We have $W \leq \sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$. During the j-th access, the weight of the accessed element is $1/(t(j)+1)^2$. Thus, the amortized time of an access is $(1+d)+3(1+d)(2\log(t(j)+1)+\log W) \leq (1+d)(6\log(t(j)+1)+4)$. As in the proof of Theorem 3, we can estimate the change in potential. □

4 The randomized splaying scheme

In deterministic splay trees, the expensive part of an access operation is the splaying step. Motivated by this observation, our randomized splaying scheme executes the splaying step only with a certain probability p. This will decrease the expected number of rotations done on a sequence of accesses. If the probability p is not too small, then frequently accessed elements will still be close to the root of the tree, and the expected search time during an access should not increase.

The following randomized splaying scheme works for any real number $p \in [0, 1]$.

Rand-Splay(p): When an element x is accessed, with probability p, splay the tree at the node holding x. With probability $1 - p$, leave the tree as it is.

We analyze *Rand-Splay(p)* using the potential function

$$\Phi = (1/p + d) \sum_{u \in T} \rho(u).$$

Loosely speaking, we will show that the $(1 + d)$ factors in Lemma 1 and Theorems 3 – 4 reduce to $(1 + pd)$.

Lemma 2 *Suppose that there is an access to an element x and that $u \in T$ is the node containing x. Then the expected amortized time incurred by Rand-Splay(p) to serve the access is at most $(1 + pd) + 3(1 + pd)(\rho(r) - \rho(u)) = (1 + pd) + 3(1 + pd)\log(\sigma(r)/\sigma(u))$, where r is the root of the tree.*

Proof: In the tree T, consider the path from the root to the node u. On this path, the algorithm encounters zig, zig-zig and zig-zag cases. Given one of these cases, let $E[t^a]$ be the expected amortized time needed by the algorithm. Then

$$E[t^a] = E[t] + E[\Delta\Phi],$$

where $E[t]$ is the expected actual time of the step and $E[\Delta\Phi]$ is the expected change in potential. For any node in the tree, let ρ and ρ' denote the ranks before and after the step. We will show that $E[t^a] \leq (1+pd)+3(1+pd)(\rho'(u)-\rho(u))$ in the zig case and that $E[t^a] \leq 3(1+pd)(\rho'(u)-\rho(u))$ in the zig-zig and zig-zag cases. The lemma follows by summing these bounds for all the cases that occur on the access path.

Let v denote the parent of u and w denote the parent of v if it exists.

1. Zig case:

The algorithm needs one time unit for searching. With probability p, one rotation is done. Thus $E[t] = 1 + pd$. If the edge between u and v is rotated, then the change in potential is $(1/p + d)(\rho'(u) + \rho'(v) - \rho(u) - \rho(v))$ because the ranks only change at u and v. If the edge is not rotated, then the potential does not change. Hence,

$$\begin{aligned} E[t^a] &= E[t] + E[\Delta\Phi] \\ &= 1 + pd + p(1/p + d)(\rho'(u) + \rho'(v) - \rho(u) - \rho(v)) \\ &\leq (1 + pd) + (1 + pd)(\rho'(u) - \rho(u)) \;\leq\; (1 + pd) + 3(1 + pd)(\rho'(u) - \rho(u)). \end{aligned}$$

The last line follows because $\rho(v) \geq \rho'(v)$ and $\rho'(u) \geq \rho(u)$.

2. Zig-zig case:

The algorithm needs two time units for searching. With probability p, two rotations are done.

Thus, $E[t] = 2 + 2pd$. If the rotations are done, then the change in potential is $(1/p+d)(\rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w))$ because the ranks only change at u, v and w. Otherwise the potential does not change. Therefore,

$$\begin{aligned} E[t^a] &= E[t] + E[\Delta\Phi] \\ &= 2(1+pd) + p(1/p+d)(\rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)) \\ &= (1+pd)(2 + \rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)). \end{aligned}$$

Using the same techniques as in [12] we can show that $(2 + \rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)) \leq 3(\rho'(u) - \rho(u))$. Hence, $E[t^a] \leq 3(1+pd)(\rho'(u) - \rho(u))$.

3. Zig-zag case:
We have

$$\begin{aligned} E[t^a] &= E[t] + E[\Delta\Phi] \\ &= 2(1+pd) + p(1/p+d)(\rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)) \\ &= (1+pd)(2 + \rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)). \end{aligned}$$

Again, as in [12] we can show $(2 + \rho'(u) + \rho'(v) + \rho'(w) - \rho(u) - \rho(v) - \rho(w)) \leq 3(\rho'(u) - \rho(u))$. We conclude $E[t^a] \leq 3(1+pd)(\rho'(u) - \rho(u))$. $\square$

The following theorems can be developed in the same way as in Section 3.

Theorem 5 *The expected total access time incurred by Rand-Splay(p) is at most* $(1+pd)(3m\log n + m) + C$, *where* $C = (1/p+d)n\log n$.

Theorem 6 *If every element is accessed at least once, then the expected total access time incurred by Rand-Splay(p) is at most* $(1+pd)(3\sum_{i=1}^{n} q(x_i)\log(m/q(x_i)) + m) + C$, *where* $C = (1/p+d)\sum_{i=1}^{n}\log(m/q(x_i))$.

Theorem 7 *If f is any fixed element, the expected total access time incurred by Rand-Splay(p) is at most* $(1+pd)(6\sum_{j=1}^{m}\log(|k(x_{i_j}) - k(f) + 1|) + 7m) + C$, *where* $C = (1/p+d)2n(\log n + 1)$.

Theorem 8 *The expected total access time incurred by Rand-Splay(p) is at most* $(1+pd)(6\sum_{j=1}^{m}\log(t(j)+1) + 4m) + C$, *where* $C = (1/p+d)(2n\log n + n)$.

The bounds in Lemma 2 and Theorems 5 – 8 show that our randomized splaying scheme improves the deterministic scheme. At first sight Lemma 2 seems to imply that the probability p should be chosen as small as possible. However, this is not true. In Theorems 5 – 8, the value of C, which reflects the overall change in potential, increases as p decreases. Thus, p should be chosen in such a way that $1/p$ is a "constant" in the additive term C. The optimal choice of p depends on the size of the tree.

So far we have only considered accesses to elements in the tree. Suppose that we also want to execute update operations on the set S. Obviously, an insertion of a new element x can also be implemented using our randomized splaying strategy. First, we insert a new leaf node containing x at the appropriate position in the tree. Then, with probability p, we splay the tree at this new node. Similarly, we can adapt the delete operation in deterministic splay trees. However, the improvement obtained using randomized splaying can be small since a delete operation involves making a join of two subtrees, see [12] for details.

5 Experimental results

We have implemented *Rand-Splay*(p) and tested it on randomly generated request sequences. More precisely, we modified an existing splay tree code by Daniel Sleator that implements *top-down splaying*. In top-down splaying the rotations are performed while the algorithm moves down along the access path.

In the following we report on experiments that we have done on a test set consisting of $N = 2^{16}$ elements. First, $N = 2^{16}$ elements with distinct integer keys from $1, \ldots, N$ were inserted into the tree in a random order. Then, kN access operations were executed; we considered the values $k = 2^i$, $i = 0, 1, \ldots, 5$. Finally, the N elements were deleted from the tree. We tested *Rand-Splay*(p) for $p = 2^{-i}$, $i = 0, 1, \ldots, 8$, and $p = 0$. Note that, for $p = 1$, *Rand-Splay*(p) is identical to the deterministic splaying scheme by Sleator and Tarjan. For $p = 0$, *Rand-Splay*(p) works with the static tree that is generated during the N insertions of elements. We did experiments with request sequences that were generated according to the following schemes.

(1) *Random sequences*: We considered request sequences generated by probability distributions and concentrated on two classical distributions.

   (a) *Uniform distibution*: Each access was made to an element chosen uniformly at random from among the N elements in the tree. This distribution has been used in the experimental analysis of other data structures, see for instance [11].

   (b) *Zipf's distibution*: For every request, the element with key i is requested with probability

$$q_i = c/i \qquad \text{where } c = 1/H_N,$$

   and $H_N = \sum_{i=1}^{N} 1/i$ is the N-th Harmonic number. This distribution has been studies extensively, see for instance Knuth [7]. The distribution is motivated by Zipf's observation, that in a natural language text, the n most common word occurs roughly with frequence proportional to $1/n$. He observed the same phenomenon in census tables and other areas.

(2) *Random sequences with high locality of reference*: Request sequences generated by applications executed on a computer system often exhibit locality of reference: Within a short interval of the request sequences, all requests are made to elements that come from a relatively small set W of elements. Such a set is called a *working set*. The working set may change over time. We model this as follows. We assume that a working set W changes after roughly l requests, where l is a multiple of $w = |W|$, i.e. $l = k \cdot w$ for some constant k. In our experiments we considered the values $w = 50, 100, 150$ and $k = 1, 2, 3$. Each working set W consists of w elements chosen uniformly at random from among the elements in the tree. We investigated subsequences of length l generated according to the following schemes.

   (a) *Uniform distribution*: Each of the l requests is an element chosen uniformly at random from W.

   (b) *Zipf's distribution*: The probabilities according to which the elements from W are requested satisfy Zipf's law.

We next describe our results. All the experiments were executed on a SPARC Ultra 10. Our experimental results are quite stable in the sense that, for each of the request schemes (1a/b) and (2a/b), a doubling of the number of accesses results in a doubling of the total access time.

Therefore Tables 1– 4 only show the results for $32 \cdot N$ requests. The same qualitative behavior shows for shorter request sequences. Each table shows the time, in seconds, that is necessary to execute $32 \cdot N$ accesses using *Rand-Splay*(p). Additionally, we have computed the relative access times, assuming that the deterministic splaying scheme *Rand-Splay*(1) takes 100% of time. Tables 3 and 4 show the results for working sets of size $w = 100$ and subsequences of length $l = 2w$. For other values of w and l we investigated, the same qualitative performance showed.

The figures do not include the time required for the N insertions and deletions of elements because our goal is to concentrate on the improvement achieved on accesses. As mentioned before, deterministic splay trees have a good performance on insertions and deletions.

p	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	0
sec.	7.47	6.56	6.08	5.87	5.77	5.74	5.70	5.69	5.74	5.79	6.0
%	100	87.8	81.3	78.6	77.2	76.8	76.3	76.1	76.8	77.5	80.3

Table 1: Access times (in seconds) and relative access times, assuming that deterministic splaying takes 100% time, of *Rand-Splay*(p) on sequence generated according to scheme (1a).

p	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	0
sec.	2.03	1.76	1.61	1.53	1.49	1.47	1.44	1.45	1.47	1.58	4.0
%	100	86.7	79.3	75.3	73.4	72.4	70.9	71.4	72.4.8	77.8	197.0

Table 2: Results for request sequence generated according to scheme (1b).

On request sequences generated according to scheme (1), for $p = \frac{1}{2}$ and $p = \frac{1}{4}$, *Rand-Splay*(p) achieves improvements of 12% to 20% over deterministic splaying,. For smaller values of p the improvements are even greater, the best results being obtained using the probability $p = \frac{1}{128}$, with improvements of about 25%. For smaller probabilities $p, p < \frac{1}{128}$, we typically observe again an increase in the access time. The same qualitative behaviour also shows for other dictionary sizes of, for instance, $N = 2^{15}$ or $N = 2^{17}$ elements. On the other hand, on request sequences generated according to scheme (2) the results are negative. Only for probabilities (around) 1/2 we observe an improvement over deterministic splaying. The improvements are in the range of 2 %, which is almost marginal.

p	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	0
sec.	4.54	4.49	4.70	4.86	5.0	5.27
%	100	98.9	103.5	107.0	110.1	116.0

Table 3: Results for request sequences generated according to scheme (2a).

The results can be explained as follows. *Scheme (1):* If requests are generated by the uniform distribution, then it is not worthwhile to rotate the elememt currently accessed to the root of the tree because it is not likely to be accessed in the near future. Executing *Rand-Splay*(p) for small probabilities p leads to good results. If requests are generated according to Zipf's law, then it is worthwhile to rotate elements to the root of the tree because only a small set of items are

p	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	0
sec.	3.49	3.41	3.50	3.72	3.97	4.8
%	100	97.7	100.2	106.6	113.7	137.5

Table 4: Results for request sequence generated according to scheme (2b).

ever requested. Our randomized algorithm achieves improvements over deterministic splaying because *Rand-Splay*(p) saves a lot of rotations once these items are near the root of the tree. *Scheme (2)*: On request sequences that exhibit high locality of reference it is necessary to rotate the requested elements to the root of the tree. In *Rand-Splay*(p) for $p > 1/2$, these movements of elements are delayed for too long, diminishing the advantage of savings in rotations.

The access times shown in the table do not include the time needed to generate pseudo-random numbers. If we add this time, then the access times are slightly (i.e. about 1%) higher and depend on the pseudo-random number generator that is used. (In our experiments we have used the `C` library functions `rand()`, `random()` and `lrand48()`.) Our motivation for neglecting the time used in random number generation is that, in our experiments, we can achieve the same results without any pseudo-random numbers. We obtain the same access times if we determinstically execute a splaying operation on every 2^j-th access, $j \in \{1, 2, \ldots, 8\}$. In about half of the cases, the counter scheme even ranks slightly better. Thus, in practice, instead of using "expensive" pseudo-random numbers, one might as well splay deterministically on every 2^j-th access.

Our overall recommentation is to execute *Rand-Splay*(p) for relatively high probabilities $p \approx 1/2$ (or the deterministic counter scheme for small values $j \approx 2$). This choice gives a reasonable improvement on request sequences generated by a fixed distribution and should not harm the performance on sequences with high locality of reference.

insertions/deletions

6 Conclusions

In this paper we investigated randomized splay trees from a theroretical and practical point of view. The algorithm presented in Section 4 is not the only natural randomized version of splay trees. Consider the following algorithm. During an access operation, whenever we encounter a zig, zig-zig, or zig-zag case on the access path, we execute the corresponding rotations with probability p. It is not hard to show that Lemma 2 and Theorems 5 – 8 also hold for this algorithm. In this paper we have examined *Rand-Splay*(p) in more detail because it is very easy to implement and uses only one random number during each access.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.

[2] R. Chaudhuri and H. Hoft. Splaying a search tree in preorder takes linear time. *SIGACT News*, 24(2):88–93, Spring 1993.

[3] R. Cole. On the dynamic finger conjecture for splay trees. part 2: Finger searching. Technical Report 472, Courant Institute, NYU, 1989.

[4] R. Cole. On the dynamic finger conjecture for splay trees. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 8–17, 1990.

[5] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part 1: Splay sorting $\log n$-block sequences. Technical Report 471, Courant Institute, NYU, 1989.

[6] M. Fürer. Randomized splay trees. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, short form abstract, pages S903–904, 1999.

[7] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addion Wesley, 1973.

[8] J. M. Lucas. On the competitiveness of splay trees; Relations to the Union-Find Problem. In *L.A. McGeoch and D.D. Sleator, On-Line Algorithms*. DIMACS Series in Discrete Mathematics and Computer Science, pages 95–124, 1992.

[9] J.I. Munro, T. Papadakis and R. Sedgewick. Deterministic skip lists. In *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 367–375. 1992.

[10] G. Port and A. Moffat. A fast algorithm for melding splay trees. In *Proc. Workshop on Algorithms and Data Structures (WADS '89)*, Spriner Lecture Notes in Computer Science, Volume 382, pages 450–459, 1989.

[11] W. Pugh. Skip lists: A probabilistic alternative to balances trees. *Communications of the ACM*, 33:668–676, 1990.

[12] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[13] A. Subramaniam. An explanation of splaying. *Journal of Algorithms*, 20:512–525, 1996.

[14] R. Sundar. Twists, turns cascades, deque conjecture, and scaning theorem. In *Proc. 30th IEEE Foundations of Computer Science*, pages 555–559, 1989.

[15] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.