Properties of Multi-Splay Trees

Jonathan Derryberry **Daniel Sleator**
Chengwen Chris Wang

November 20, 2009
CMU-CS-09-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We show that multi-splay trees have most of the properties that splay trees have. Specifically, we show that multi-splay trees have the following properties: the access lemma, static optimality, the static finger property, the working set property, and key-independent optimality. Moreover, we prove that multi-splay trees have the deque property, which was conjectured by Tarjan in 1985 for splay trees, but remains unproven despite a significant amount of research toward proving it.

This research was sponsored by National Science Foundation (NSF) grant no. CCR-0122581.

Keywords: Dynamic Optimality, Binary Search Trees, Splay Trees, Competitive Algorithms

1 Introduction

Efficiently maintaining and manipulating sets of elements from a totally ordered universe is a fundamental problem in computer science. Specifically, many algorithms need a data structure that can efficiently support at least the following operations: insert, delete, predecessor, and successor, as well as membership testing. A standard data structure that maintains a totally ordered set and supports these operations is a binary search tree (BST), which has the added benefit of being augmentable (i.e., BSTs can store, for example, the sum of auxiliary values that have been stored in each subtree). Various types of BSTs were independently developed by a number of researchers in the early 1960s [Knu73]. Many of them achieved the theoretical minimum of $O(\log n)$ key comparisons per operation, and it is easy to see that these are optimal (up to a constant factor) using worst-case analysis.

However, for many sequences σ of m operations, the optimal cost for serving the sequence is $o(m \log n)$, which is lower than the theoretical minimum that uses worst-case, per-operation bounds. A number of BST algorithms have been developed to exploit the patterns in query sequences from specific applications. A few examples of such exploitable patterns are the following: a skewed probability distribution[1], sequential queries[2], spatial locality (i.e., most accesses are close to the previous access, or some specified finger)[3], and the combination of spatial and temporal locality specified in the unified bound[4].

One of these BST algorithms, splay trees [ST85, Tar85], can provably optimally exploit patterns of many of the above types as well as other types of patterns. In addition to satisfying the balance theorem [ST85] (which states that the amortized access cost is $O(\log n)$), splay trees satisfy the following properties: static optimality [ST85], the static finger theorem [ST85], the working set theorem [ST85], the scanning theorem [Sun89a, Tar85, Sun92, Elm04], the reweighting lemma [Geo04], the dynamic finger theorem [CMSS00, Col00], key independent optimality [Iac02], and competitiveness to parametrically balanced trees [Geo04]. Because splay trees were shown to have such properties, Sleator and Tarjan [ST85] conjectured them to be *dynamically optimal*, meaning they are $O(1)$-competitive to the optimal off-line BST. After more than 20 years, this conjecture remains an open problem.

Since no one has shown that splay trees, or any other BST, is $O(1)$-competitive, Demaine *et al.* suggested searching for alternative BST algorithms that have small but non-constant competitive factors [DHIP04]. They proposed *tango*, a BST algorithm that achieves a competitive ratio of $O(\log \log n)$. Tango is the first BST data structure to provably achieve a nontrivial competitive factor. Unfortunately, tango does not satisfy many of the necessary conditions to be a constant-competitive BST, including some that splay trees are known to satisfy. For example, it does not satisfy the scanning theorem, or even the balance theorem. The journal version of the tango paper, which was published after the initial version of this paper was written, did suggest a modification to tango that would satisfy the balance theorem, but details were omitted.[5]

[1]See [Knu71, Fre75, Meh75, Meh79, GW77, HT71, HKT79, Unt79, Hu82, Kor81, KV81, BST85]

[2]See [Tar85, Sun89a, Sun92, Elm04]

[3]See [BY76, GMPR77, Tsa86, TvW88, HL79, Har80, HM82, Fle93, SA96, BLM$^+$03, Pug89, Pug90, Bro05]

[4]See [Iac01, DS09]

[5]The original version of this paper was not published, but the results in this paper were included in Wang's the-

Two alternative BST algorithms to tango have been proposed. Georgakopoulos [Geo05] modified splaying [ST85] to create the chain-splay algorithm, which he showed to be $O(\log \log n)$-competitive with an amortized running time of $O(\log n)$ per operation. Nevertheless, he did not show that it achieved any of the other necessary conditions to be a constant-competitive BST. Independently, Wang *et al.* developed an alternative to tango called the multi-splay algorithm [WDS06]. Using a similar technique to tango, multi-splay trees solved some of the theoretical shortcomings of tango by provably satisfying the balance theorem and the scanning theorem while maintaining the $O(\log \log n)$-competitiveness of tango. Multi-splay trees also added support for updates (insertions and deletions). However, multi-splay trees still were not proven to have some of the important properties that splay trees have, most notably the access lemma, a powerful lemma proven for splay trees from which many other properties follow (See Theorem 1).

In Section 2 of this paper, we show that multi-splay trees satisfy a property very similar to the access lemma, and many of the original splay tree theorems follow, including the balance theorem, the static optimality theorem, the static finger theorem, and the working set theorem. With this multi-splay tree version of the access lemma in hand, multi-splay trees are now known to have all of the properties of splay trees that are special cases of dynamic optimality except the dynamic finger theorem.

Additionally, we show that multi-splay trees satisfy the deque theorem, which is currently an unproven conjecture for splay-trees. This theorem states that if the only operations ever used on a multi-splay tree are push, pop, inject, and eject (i.e., insertion or deletion of the smallest or largest element), then each operation runs in $O(1)$ amortized time. For splay trees, Sundar proved that splay trees satisfy the deque property to within a factor of $\alpha(n)$ [Sun89a, Sun89b, Sun92], and Pettie later improved this bound to $\alpha^*(n)$ amortized time per operation [Pet08], and this is the best bound that has been proven so far for the performance of splay trees when used as deques.

1.1 Background

To understand the results in this paper, a basic understanding of the multi-splay algorithm is required. For the sake of brevity, we will assume the reader is familiar with the description of the multi-splay algorithm in the static case (no insertions or deletions) presented in either [Wan06] or [WDS06]. In particular, the reader should understand the relationship between the reference tree (which is not explicitly stored) denoted by P and the actual BST data structure T. The reader should understand that each query in T is a simulation of a series of "switches" in P, similar to what happens in link-cut trees, creating a solid path from the accessed node in P to the root of P. Each switch is simulated in T by a series of up to three splays and two `isRoot` bit togglings. An example of a left-to-right switch is shown in Figure 1.

In Section 3, an understanding of the multi-splay algorithm in the dynamic case (insertions and deletions allowed) is helpful but not required. Within Section 3, we supply the reader with the important details for understanding how the multi-splay algorithm behaves when a multi-splay tree is used as a deque.

In addition to having a basic understanding of the multi-splay algorithm, it is also helpful to be

sis [Wan06].

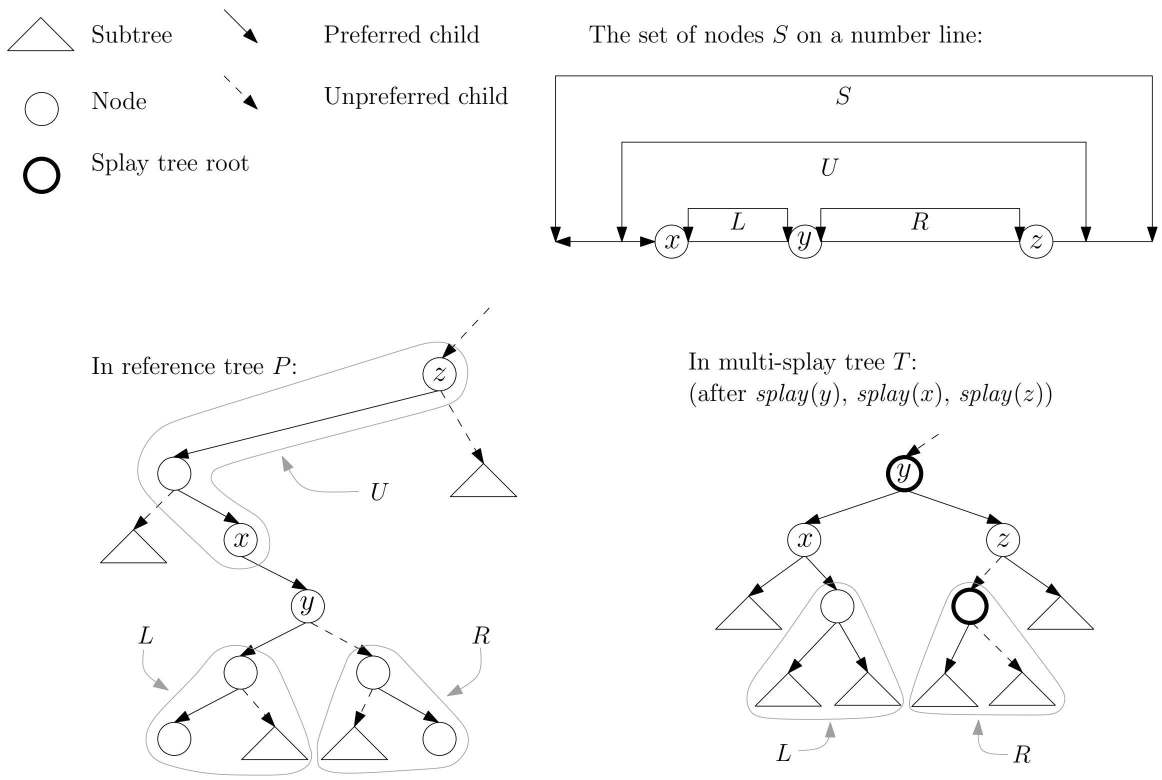

Figure 1: This figure shows the three nodes, x, y, and z, that are splayed during a left-to-right switch on y, immediately before the marking and unmarking of the `isRoot` bits. In this figure, S denotes the set of elements stored in the BST, U denotes the set of elements above y in the reference tree that are part of y's preferred path, and L and R denote, respectively, the elements of the left and right subtrees of y in the reference tree.

familiar with the following two properties of splay trees, which will be relied upon heavily in the proofs in this paper.

Theorem 1 (Access Lemma)**.** *[ST85] The amortized time to splay a node v in a BST currently rooted at r is at most $c_{sa} + c_s \lg(s(r)/s(v))$, where $c_{sa} = 1$ and $c_s = 3$.*

Theorem 2 (Reweighting Lemma)**.** *[Geo04] For any sequence of interleaving splays and reweight operations on nodes in a BST, the amortized time to splay a node v in a splay tree currently rooted at r is at most $c_s \lg(s(r)/w(v)) + c_{sa}$, and the amortized time to reweight a node v from $w(v)$ to $w'(v)$ is $max(0, c_r \lg(w'(v)/w(v)))$, where $c_r = c_s + 1 = 4$.*[6]

[6] Note that the denominator inside $c_s \lg(s(r)/w(v)) + c_{sa}$ is $w(v)$, which is different from in the splay tree access lemma.

2 Multi-Splay Tree Access Lemma

In this section, we show that multi-splay trees satisfy a property that is similar to the access lemma for splay trees. Using this lemma, we can easily prove the static finger theorem, static optimality, and many other properties proved for splay trees using the access lemma. The statement of the multi-splay tree access lemma is the following.

Lemma 1. *In a multi-splay tree T with an arbitrary (not necessarily balanced) reference tree P, let $mass(x)$ be any positive weight assignment on the nodes, and let $\sigma = \sigma_1 \cdots \sigma_m$ be a sequence of elements to query. The amortized cost of multi-splaying σ_j is $O((\lg \frac{M}{\hat{w}(\sigma_j)}) + 1)$, where $M = \sum_y mass(y)$ and $\hat{w}(\sigma_j)$, which is at least $mass(x)$, is defined in Equation 1 below.*

Our overall approach to proving Lemma 1 will be to assign weights to the elements of the BST T roughly based on their mass assignments, and repeatedly use the reweighting lemma of [Geo04]. The amortized cost of each switch will be bounded by the cost of the three splays according to the access lemma for splay trees, plus the change in potential due to the addition/removal of weight to/from the tree due to root markings (for the purpose of analysis, T is assumed to be broken into multiple splay trees, which are linked together to form one larger BST), plus the cost of reweighting some nodes in T as will be described later. The total amortized cost of all of the switches will form a telescoping sum and give us the required bound.

Before we can define the weight of each element in the splay trees that constitute a multi-splay tree T with reference tree P, we need the following definitions. Let $uchild(x)$ be the unpreferred child of x in P. Let $A(x)$ be the set of proper ancestors of x in P. Let $refLeftParent(x)$ be the predecessor of x in $A(x)$, if it exists, and define $refRightParent(x)$ analogously. Let $lip(x)$ be the set of nodes in x's left inner path in P, the set of nodes reachable starting at x's left child in P and following right child pointers in P, and let $rip(x)$ be defined analogously. Let $refSubtree(x)$ be the set of nodes in x's subtree in P. In addition, to help define the node weights we will use when we prove the multi-splay tree access lemma, we will use the following notation, some of which is shown in Figure 2.

$$\begin{aligned} US(x) &= refSubtree(uchild(x)) \\ \hat{w}(x) &= mass(x) + \sum_{y \in US(x)} mass(y) \\ \Diamond(x) &= lip(x) \cup rip(x) \cup \{x\} \\ w(x) &= \max_{y \in \Diamond(x)} \hat{w}(y). \end{aligned} \tag{1}$$

We assign a weight of $w(x)$ to each element in a multi-splay tree for our analysis. The size, $s(x)$, of node x is equal to $\sum_{y \in splaySubtree(x)} w(x)$, where $splaySubtree(x)$ is the subtree rooted at x restricted to the splay tree containing x. The potential of T is $\sum_{x \in T} \lg(s(x))$. If we were to use $\hat{w}(x)$ as the weight assignment, then this would essentially be the same weight assignment as that which is used in link-cut tree analysis [ST85].

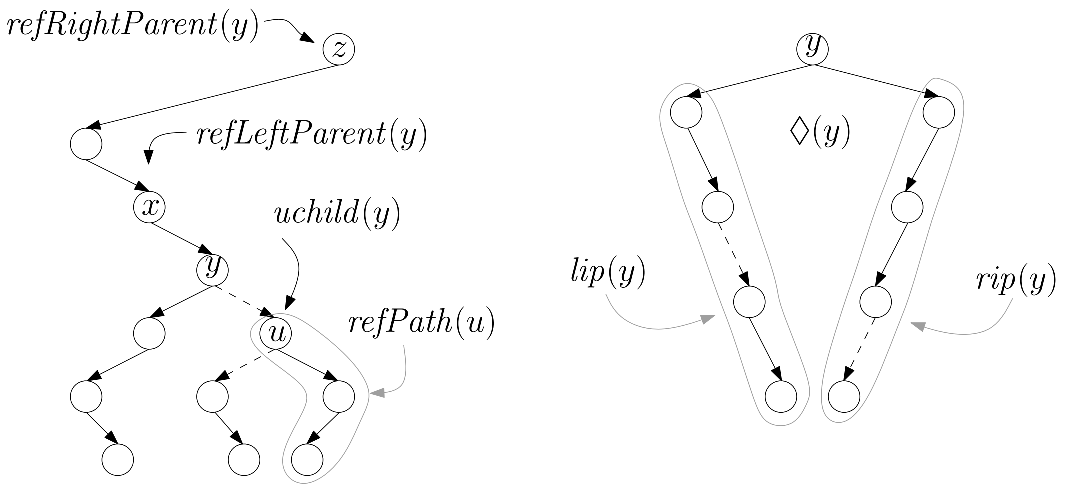

Figure 2: A depiction of the notation in the reference tree

However, each time we switch a node y in a multi-splay tree, in addition to splaying y, we also splay $refLeftParent(y)$ and $refRightParent(y)$, if they are in y's current splay tree. On the other hand, in link-cut trees, we would only splay y. To pay for the cost of the extra two splays, we choose the weight assignment carefully so that the extra two splays will be relatively cheap. Our definition of $w(y)$ above gives us the following invariant.

Invariant 1. *For all nodes y, $w(refLeftParent(y)) \geq \hat{w}(y)$, $w(refRightParent(y)) \geq \hat{w}(y)$, and $w(y) \geq \hat{w}(y)$ whenever $refLeftParent(y)$ or $refRightParent(y)$ exists.*

Note that for a fixed reference tree and mass assignment, different choices of preferred children can result in different weight assignments. Thus, as the algorithm performs switches to change the preferred children, the weights some nodes may change. Such a change in weight will be accounted for by using the reweighting lemma [Geo04].

Lemma 2. *In a multi-splay tree T with reference tree P, let $refPath(z)$ be the set of nodes in z's preferred path that are at least as deep as z in P. For every $x \in P$,*

$$\sum_{y \in refPath(uchild(x))} w(y) \leq 3 \sum_{u \in US(x)} mass(u) \leq 3\hat{w}(x).$$

Proof. First, it is clear that $3\sum_{u \in US(x)} mass(u) \leq 3\hat{w}(x)$ by the definition of $\hat{w}(x)$, so we only need to show that

$$\sum_{y \in refPath(uchild(x))} w(y) \leq 3 \sum_{u \in US(x)} mass(u).$$

In order to see this, we will show that

$$\sum_{y \in refPath(uchild(x))} (\hat{w}(y_L) + \hat{w}(y) + \hat{w}(y_R)) \leq 3 \sum_{u \in US(x)} mass(u),$$

where $y_L = \operatorname{argmax}_{z \in lip(y)} \hat{w}(z)$ and y_R is defined analogously (y_L or y_R may not exist, in which case we assume $\hat{w}(y_L) = 0$ or $\hat{w}(y_R) = 0$ as appropriate).

Notice that $\sum_{y \in refPath(uchild(x))} \hat{w}(y) = \sum_{u \in US(x)} mass(u)$ by definition, so it suffices to show that

$$\sum_{y \in refPath(uchild(x))} \hat{w}(y_L) \leq \sum_{y \in refPath(uchild(x))} \hat{w}(y).$$

The y_R case is symmetrical. To show that this inequality is true, for each $y \in refPath(uchild(x))$, we map y_L to a distinct member $f(y_L)$ such that $f(y_L) \in refPath(uchild(x))$ and $\hat{w}(y_L) \leq \hat{w}(f(y_L))$ (if y_L does not exist we assume for convenience that it maps to some fake node $f(y_L)$ such that $\hat{w}(f(y_L)) = 0$). This, in addition to the fact that each y_L is distinct, suffices to prove the result. Hence, for each y_L that exists, we define

$$f(y_L) = \operatorname*{argmax}_{z \in refPath(uchild(x)) \cap (A(y_L) \cup y_L)} \texttt{refDepth}(z),$$

where $\texttt{refDepth}(z)$ is the depth of z in P. Note that either $f(y_L) = y_L$ or $y_L \in US(f(y_L))$ so $\hat{w}(y_L) \leq \hat{w}(f(y_L))$. Further, note that it is the case that $refRightParent(f(y_L)) = y$ and $f(y_L) \in refPath(uchild(x))$, so for each y for which y_L exists, $f(y_L)$ is a distinct member of $refPath(uchild(x))$. Thus,

$$\sum_{y \in refPath(uchild(x))} \hat{w}(y_L) \leq \sum_{y \in refPath(uchild(x))} \hat{w}(f(y_L)) \leq \sum_{y \in refPath(uchild(x))} \hat{w}(y).$$

□

With Lemma 2 in hand we are ready to proceed with the following proof of the multi-splay tree access lemma.

Proof. Each query consists of a sequence of k switches and a final switch on the queried element. Each switch consists of at most three splays and at most two changes to `isRoot` bits. Let y_i be the i^{th} node being switched going up T's access path toward T's root and r_i be the root of the splay tree T_i containing y_i (y_1 is the first node switched, and by convention the splay tree rooted at r_0 contains the queried element $y_0 = \sigma_j$). Let x_i and z_i denote $refLeftParent(y_i)$ and $refRightParent(y_i)$, respectively (if these nodes exist), and let L_i and R_i denote the elements of the (possibly empty) subtrees of P containing the intervals (x_i, y_i) and (y_i, z_i), respectively. The amortized cost of a switch consists of three parts (here we assume the switch is from left to right): splaying y_i up to r_i's location, x_i until it is the left child of y_i (if $x_i \in T_i$), and z_i until it is the right child of y_i (if $z_i \in T_i$); marking the `isRoot` bit of the least common ancestor (LCA) of L_i if $L_i \neq \emptyset$ (i.e., marking the right child of x_i if x_i exists and is in T_i) and unmarking the `isRoot` bit of the LCA of R_i; and reweighting x_i, y_i, and z_i if they exist, so as to restore Invariant 1 (even if x_i and z_i exist but are not in T_i). We bound each of these costs in the following few paragraphs.

First, by Invariant 1 and the splay tree reweighting lemma, the amortized cost of the three splays is at most,

$$3\left(\lg\left(\frac{s(r_i)}{w(y_i)}\right) + \lg\left(\frac{s(r_i)}{w(x_i)}\right) + \lg\left(\frac{s(r_i)}{w(z_i)}\right)\right) + O(1) = O\left(\lg\left(\frac{s(r_i)}{\hat{w}(y_i)}\right) + 1\right).$$

Second, it is free to mark the LCA of L_i as a root because this decreases the potential of T. As for unmarking the LCA of R_i, by Lemma 2, the increase in $s(y_i)$ and $s(z_i)$ is bounded by $3\hat{w}(y_i)$ since R_i is fully contained within y_i's unpreferred subtree, and by Invariant 1, $\hat{w}(y_i) \leq w(z_i)$. Hence, the increase in potential resulting from the increased sizes of y_i and z_i is bounded by $2\lg(3+1) = 4$.

Third, after a switch on y_i, the specified value of $\hat{w}(y_i)$ may have increased or decreased. For all other nodes x, $\hat{w}(x)$ remains the same. This change in $\hat{w}(y_i)$ can only affect the weights of x_i, y_i, and z_i (even if x_i or y_i is not in T_i). If $\hat{w}(y_i)$ decreases, then $w(x_i)$, $w(y_i)$, and $w(z_i)$ cannot increase. So we can apply the reweighting Lemma and pay a cost of 0. On the other hand, if $\hat{w}(y_i)$ increases, we have to bound the changes in weights. To account for the amortized cost of the changes in weights, we lower-bound the weights before reweighting occurs and we upper-bound the weights after reweighting occurs.

By Invariant 1, before reweighting x_i, y_i, and z_i,

$$\begin{aligned} w(x_i) &\geq \hat{w}(y_i) \\ w(y_i) &\geq \hat{w}(y_i) \\ w(z_i) &\geq \hat{w}(y_i). \end{aligned} \tag{2}$$

After reweighting, when $\hat{w}(y_i)$ has its new value $\hat{w}'(y_i)$, if $w(x_i)$, $w(y_i)$, or $w(z_i)$ increases (to $w'(x_i)$, $w'(y_i)$, or $w'(z_i)$), it must increase to $\hat{w}'(y_i)$. Because y_i is in T_i, it is true that $\hat{w}'(y_i) \leq s(r_i)$. Thus, for all $v \in \{x_i, y_i, z_i\}$ for which $w'(v) > w(v)$

$$w'(v) \leq s(r_i). \tag{3}$$

Thus, by the reweighting lemma, Equation 2, and Equation 3, the amortized cost of reweighting is at most $O(\lg(s(r_i)/\hat{w}(y_i)))$, which is the same (up to a constant) as the upper bound on the amortized cost of the splays.

We still need to account for the amortized cost of the series of switches on $y_1, y_2, \ldots, y_k$ and the amortized cost of the final switch on the queried element. By Lemma 2, $s(r_i) \leq 3\hat{w}(y_{i+1})$, so the series of k switches costs at most

$$\begin{aligned} \sum_{i=1}^{k} O\left(\lg\left(\frac{s(r_i)}{\hat{w}(y_i)}\right) + 1\right) &\leq \sum_{i=1}^{k} O\left(\lg\left(\frac{3s(r_i)}{s(r_{i-1})}\right)\right) + O(k) \\ &\leq O\left(\lg\left(\frac{s(r_k)}{w(y_0)}\right)\right) + O(k). \end{aligned}$$

Since k is smaller than the number of rotations needed for splaying y_0 during the final switch, $O(k)$ can be charged to the last switch if each splaying step pays for an additional unit of work. Using the same analysis as that of the previous switches, the amortized cost of the final switch is

$$O\left(\lg\left(\frac{s(\mathit{splayRoot})}{\hat{w}(y_0)}\right) + 1\right),$$

where *splayRoot* is the root of T.

Note that $s(\mathit{splayRoot}) \leq 3\sum_v \mathit{mass}(v) = 3M$. Summing the bounds on the costs for the series of k switches and the final switch proves the multi-splay tree access lemma. $\square$

We did not make any assumptions about the reference tree in this proof. Thus, the proof works on any reference tree. (However, for a multi-splay tree to be provably $O(\log \log n)$-competitive, it is still important to have a balanced reference tree.) Moreover, we did not account for the initial and final potential, which needs to be accounted for when we apply this lemma. We note that if the ratio of the maximum mass to the minimum mass is bounded by $O(poly(n))$, then the maximum difference in potential is bounded by $O(\log n)$ for each node, so the difference between the initial and final potentials is bounded by $O(n \log n)$.

Corollary 1. *[ST85] Multi-splay trees satisfy the static finger property.*

Proof. The access lemma implies the static finger property [ST85]. □

Corollary 2. *[ST85] Multi-splay trees satisfy the static optimality property.*

Proof. The access lemma implies the static optimality property [ST85]. □

Corollary 3. *Multi-splay trees satisfy the working set property.*

Proof. The techniques used in this proof are identical to the proof of the working set theorem for splay trees [ST85]. For the purpose of mass assignment, we maintain a linked list of all the keys. Whenever a key is queried, we move the key to the front of the list. This is essentially a move-to-front list of all the keys. Let $p(v)$ denote the position of key v in the move-to-front list. We assign $mass(v)$ a value of $1/p(v)^2$. Note that $M = \sum_v mass(v) = O(1)$.

Whenever we query v, the cost of the query is $O(\lg(M/mass(v)) + 1) = O(\lg(p(v)) + 1)$ by the access lemma. After we query v, we increase $mass(v)$ to 1, and decrease the mass of all other nodes. Because of this change in mass, we increase the weight of 3 nodes. Specifically, we increase $w(refLeftParent(v))$, $w(v)$, and $w(refRightParent(v))$. By Invariant 1, the weight of each of these three nodes is at least $mass(v)$. Thus, the cost of reweighting each node up to 1 is at most $O(\lg(1/mass(v))) = O(\lg(p(v)))$. □

Corollary 4. *[Iac02] Multi-splay trees satisfy the key independent optimality property.*

Proof. The working set property implies the key-independent optimality property [Iac02]. □

3 Deque Theorem

Before we prove the deque theorem, we give a brief description of what happens during a deque operation (e.g., push, pop, inject, or eject) performed on a multi-splay tree T with reference tree P, where P is a red-black tree. (For a more thorough description of how insertion and deletion work, see [Wan06], which has a more detailed and slightly modified description of insertion and deletion compared to the original version that appeared in [WDS06].) Because of the similarity between push and inject, and between pop and eject, this description will focus only on push and pop. This description is not using special case code for handling push and pop operations. It is just the simplified view of what insertion and deletion look like when they happen to be performed using only pushes/pops (and injects/ejects).

To perform $push(x)$, suppose y is the minimum element of T. We first query y and then insert x as y's left child. We then set x's `refDepth` and `minRefDepth` fields, which contain, respectively, the depth of x in the reference tree and the minimum value of `refDepth` from among the nodes in $splaySubtree(x)$. Both of these fields are set to be one more than y's `refDepth`, and this effectively virtually inserts x into P as the left child of y. Note that as described in [WDS06] and [Wan06], we do not store these fields explicitly. We store only the difference in values between the node and its parent in T so that we can update them efficiently.

Once these fields have been set, we are ready to virtually rebalance P, which includes performing a series of virtual pointer traversals followed by a constant number of virtual rotations and a switch on x. (We switch x even though it has at most one child in P so that all our costs can be phrased in terms of switching costs.) Note that virtual traversal of a node v to $\texttt{refParent}(v)$ is executed by two switches to v followed by two switches to $\texttt{refParent}(v)$, and these switches cause splays that compress each path that is traversed in T. Virtual traversal of v to its right-child w is executed by two switches to v, which suffices to isolate the preferred path from w to a leaf of P into its own subtree. Next, we search for w in w's splay tree by using the `minRefDepth` fields. This is possible because w is the element in this splay tree whose `refDepth` field is minimal. Finally, we perform two switches on w so that the path we traversed is compressed.

To perform a virtual rotation of w over v, we first perform a switch on v, if necessary, so that v prefers away from w. Next, we perform a switch on w, if necessary, so that w prefers toward v. Then, we perform a switch on v so that v prefers w. Finally, we update the fields of the nearby nodes so as to modify the depth of v and w in P. In this paper, we further make the simplifying assumption that after this rotation, v is switched, if necessary, to prefer in the direction of the previous query in its subtree of P. Aside from facilitating our deque analysis by helping to ensure that splays in the inner portion of P do not interact with elements on the outer paths of P (i.e., the left and right paths, starting at the root of P), this is also a natural way to define a rotation in general on a multi-splay tree, in keeping with the idea that nodes should prefer in the direction of the most recent query in their subtree of P.

During a pop, we query the smallest element x of T, and then query the successor of x so that x becomes a leaf of T. Virtual removal of x from both T and P is then straightforward, and includes performing a constant number of field updates. Virtual deletion is completed by performing virtual rebalancing similarly to how it is done following a push.

Our proof of the deque theorem will use a potential function, which we will now define. First, define the *outer shell* of P to be the union of the left and right paths of P. For each node x in the outer shell of P, the *black height* of x, denoted by $bh(x)$, is defined to be the number of black nodes on any path from x to a leaf. We assign weights $w(x) = 1/2^{bh(x)}$ to every node on the outer shell of P with the exception of the root, which is given weight 1. The size of node x, denoted by $s(x)$ is defined to be the sum of the weights of the nodes in x's subtree in T (ignoring the root markings that partition T). The rank of node x, denoted by $r(x)$, is defined to be $\lg s(x)$, and we assign a potential of $r(x)$ only to nodes located on the outer shell of P. Every node that is not a member of the outer shell of P is assigned a weight and rank of 0. The potential of T is the sum of all of these ranks.

We begin by proving the following simple lemma regarding the structure of the reference tree and its preferred children assignments.

Lemma 3. *If a push or pop (inject or eject) has been executed since the last time the root of P was rotated, then the left (right) path of P has at most one unpreferred edge, which can only be at the root of P. This is true following every query, virtual pointer traversal, and virtual rotation.*

Proof. For simplicity, we consider only the case of pushes and pops. The case of injects and ejects is analogous. We can use a simple induction argument. As a base case, note that after the first query during any push or pop, the left path is a preferred path in P. For the inductive step, we note that there are four types of operations that can disrupt this preferred path. First, an inject or eject could be performed, causing the root to prefer right. This switch does not violate the claim of the lemma. Second, a virtual pointer traversal could be executed, causing a switch on the left path. In this case, the switch is immediately undone. Third, a virtual rotation may switch a node on the left path. In this case, before the rotation is complete, the switched node will be switched again to prefer left because the previous query in its subtree of P is to its left. This is true because a push or pop has been executed since the last time the root was rotated. Fourth, a rotation of the root may disrupt the left path from being a preferred path. However, in this case, there has no longer been a push or pop since the last rotation of the root. □

We are now ready to proceed with proving that multi-splay trees satisfy the deque property. First, we note that red-black trees have the following property, which will be useful in our proof later on.

Lemma 4. *During any sequence of m rebalancing operations following insertions and deletions of nodes of height 0 or 1 in a red black tree, the number of times we touch a node at height h is at most $c_1 m/2^{(h/c_2)} + c_3$, where c_1, c_2, and c_3 are fixed constants.*

Proof. See [HM82]. □

We continue by proving two lemmas. First, we prove a bound on the cost of the switches as a function of the height of the switched node in the reference tree. Second, we prove a bound on the amortized cost of a virtual rotation in P that includes the amortized cost of the field updates resulting from a rotation (i.e., due to reweighting as a result of changes in black heights, as well as due to the change in potential when nodes enter and leave the outer shell of P). Our proof will use the reweighting lemma for splay trees [Geo04] because we need to change some of the weights in T. Here, and elsewhere in the rest of the paper, we will use $refRoot$ to refer to the current root of P.

Lemma 5. *In a sequence of deque operations, if a push or pop (inject or eject) operation has been executed since the last time the root of P was virtually rotated, then the amortized cost of a switch of node x resulting from a push or pop (inject or eject) at height h in P is $O(h)$, unless the switch is at $refRoot$, in which case the amortized cost is $O(1)$.*

Proof. We will only prove the lemma for a push or pop. The proof for an inject or eject is analogous. We will break our proof of lemma 5 for a push or pop into three cases, which encompass the types of switches that can occur as a result of a push or pop during a sequence of deque operations.

First, *refRoot* can be switched if the previous deque operation was an inject or eject. The amortized cost of switching *refRoot* is $O(1)$ because switching *refRoot* consists only of performing a constant number of field updates in addition to one splay of *refRoot*, whose weight is a constant fraction of the total weight of T.

Second, the switch could be performed on a node x at height h not on the outer shell of P. This can only happen as the result of a virtual pointer traversal or a virtual rotation. When such a switch occurs during a sequence of deque operations and a push has been performed since the last rotation of *refRoot*, we know by Lemma 3 and the definitions of a virtual pointer traversal and virtual rotation that x is separated from the outer shell by an unpreferred edge, so the number of nodes in x's splay tree is at most h. Thus, the *worst-case* cost of such a switch is $O(h)$, because there is no change to the potential of T.

Third, the switch could be performed on a node x at height h on the outer shell of P. Such a switch consists of two splay operations in T (on x and $\texttt{refParent}(x)$), in addition to a constant number of field updates (this can include the field updates due to a virtual rotation, if the switch is part of a virtual rotation). Because the weight of x is $2^{-bh(x)}$ and $h \leq 1 + 2bh(x)$, it follows that the amortized cost of switching x is $O(bh(x)) = O(h)$ using the reweighting lemma. (We are not reweighting here, but must invoke the reweighting lemma because we will be reweighting elsewhere.) The analysis of this case relies on the fact that if a push has been executed since the previous rotation of *refRoot*, no node is ever rotated over top of a node on the outer shell. This fact follows immediately from Lemma 3 and the definitions of a virtual pointer traversal and virtual rotation. $\square$

Lemma 6. *Let x be the highest node in P that is virtually traversed or involved in a virtual rotation during a virtual rebalancing operation. The amortized cost of the field updates due to this operation is $O(h)$, where h is the height of x in P.*

Proof. There are two types of changes to potential that can occur as a result of a virtual rotation. First, if the black height of a node x in the outer shell of P changes when the fields of the relevant nodes are updated to reflect this change, we need to reweight x. However, note that the black height of x can only change if x is touched during the rebalancing operation. Therefore, if the highest node touched is at height h, then only $O(h)$ nodes need to be reweighted to reflect their new black heights.

Furthermore, even if x's black height changes from $bh(x)$ to $bh'(x)$ during a rebalancing operation, it is true that $|bh'(x) - bh(x)| \leq 1$. Hence, the cost of reweighting each node whose black height changes is $O(\lg \frac{w_{new}(x)}{w_{old}(x)}) = O(\lg \frac{2^{-bh'(x)}}{2^{-bh(x)}}) = O(1)$ in the case in which the reweighted node is not *refRoot*, and $O(\lg \frac{1}{2^{-bh(y)}}) = O(bh(\textit{refRoot}))$ when a node y is rotated over *refRoot*.

Second, for each virtual rotation in P, a node may join or leave the outer shell of P. When a node x leaves the outer shell, the potential of T decreases when its weight is removed, but increases when its negative rank is removed from the total potential. However, the increase in potential due to the removal of x's rank from the total potential is bounded by $O(bh(x))$, as required.

On the other hand, when a node x joins the outer shell at height h in P, it is the case that x has at most 2 ancestors in T because the virtual rotation that placed x in the outer shell involved a switch of both x and $\texttt{refParent}(x)$, which were (and are) in the same splay tree as *refRoot*. Thus,

x's newly added weight only increases the rank of a constant number of nodes. Further, for each ancestor a of x, we know that $\mathit{refHeight}(a) \leq \mathit{refHeight}(x) + 1$. Therefore, the increase in rank at each ancestor is bounded naïvely by $\lg 3$, and x has a negative rank when it is added to the outer shell, so the inclusion of x's rank does not add any positive potential. Thus, the total increase in potential caused by a virtual rotation of x onto the outer shell of P is $O(1)$. □

Theorem 3 (Deque Theorem)**.** *In a multi-splay tree, a sequence of m deque operations (push, pop, inject, and eject) starting from an empty tree costs $O(m)$.*

Proof. First, we note that the root of a red-black tree can only rotate once every $\Omega(n)$ operations [HM82]. Further, the worst-case cost of an operation in a multi-splay tree is $O(\lg^2 n)$, and the maximum difference in potential is also bounded by $O(\lg^2 n)$. Therefore, we can pay for the first push or pop following a rotation of the root of P by amortizing this pessimistic cost of $O(\lg^2 n)$ over the $\Omega(n)$ operations that preceded the rotation of the root of P. This type of epoch argument is similar to that which was used in [Sun89a, Sun92].

Therefore, in this proof, we will be analyzing the cost of all pushes and pops that are not the first push or pop following a rotation of the root of P. Injects and ejects can be handled analogously and separately.

We will break the cost of such a push or pop into two parts. First, the cost of the initial query is $O(1)$ because it involves at most 2 switches, one at the root and one at the queried node (of black height 0), both of which cost $O(1)$ by Lemma 5.

Second, we account for the cost of virtual rebalancing. The nodes that we virtually traverse and rotate in P are the same as the nodes we would actually traverse in P if it existed. Each such virtual traversal or rotation consists of a constant amount of bookkeeping plus a constant number of switches to nodes whose black heights are no larger than the black height of the highest node that would actually be touched during that operation in a real red-black tree. Lemmas 5 and 6 show that the total amortized cost of traversing or rotating a node x is $O(bh(x))$. Thus, by Lemma 4, the total cost of virtual rebalancing over the entire sequence of pushes and pops that are not the first push or pop following a rotation of the root of P is

$$O\left(\sum_{h=0}^{\infty} h(c_1 m/2^{(h/c_2)} + c_3)\right) = O(m).$$

Finally, notice that the potential of T is 0 initially, and after m operations, we can pop all elements in at most m additional operations while bringing the potential back to 0. Thus, the total cost of a sequence of m deque operations starting from an initially empty tree and ending with a possibly non-empty tree is $O(m)$. □

Intuitively, it is easier to argue that multi-splay trees support efficient deque operations than for splay trees because the left and right paths of the reference tree of a multi-splay tree do not interfere with one another. To see this, consider what happens when we are trying to find the queried element. If the search does not cause *refRoot* to switch, then finding the queried element takes constant time because it is always at the root of the multi-splay tree. If an operation causes *refRoot* to switch, after we perform one switch at *refRoot*, the element being queried must have depth 2 or 3 unless a large number of injects have been performed. In other words, $refRoot$ essentially acts as a "divider" in T, which helps ensure that restructuring due to pushes and pops does not interfere with restructuring due to injects and ejects.

4 Conclusion

In this paper, we proved two key results. First, we showed that multi-splay trees satisfy an access lemma that is similar to that which was shown for splay trees [ST85]. This version of the access lemma is sufficient to show that multi-splay trees, like splay trees, satisfy the working set property. Next, we proved that multi-splay trees satisfy the deque property, which remains an unsolved problem for splay trees that has been open since it was conjectured in 1985 [Tar85]. Together, these results show that multi-splay trees satisfy many of the important properties of a dynamically optimal BST algorithm, in addition to being provably $O(\log \log n)$-competitive [WDS06].

Proving the deque theorem has proved to be significantly easier for multi-splay trees than for splay trees, largely due to the fact that multi-splay trees have much of the same behavior as splay trees, but have a fair amount of structure thanks to the underlying preferred path decomposition. Several interesting open problems remain regarding splay trees and multi-splay trees. First, proving that splay trees satisfy the deque property is an obvious direction for future work. Second, another potential area of future work involves proving additional adaptive properties of multi-splay trees. For example, one might be able to show that multi-splay trees satisfy the dynamic finger property or the unified bound proposed by Iacono [Iac01]. The unified bound is a generalization of the dynamic finger property, the working set property, and the deque property. Finally, the most desirable, and perhaps most difficult, future result is to prove that some BST algorithm has a competitive ratio that is better than $O(\log \log n)$.

References

[BLM$^+$03] Gerth S. Brodal, George Lagogiannis, Christos Makris, Athanasios Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences, Special issue on STOC 2002*, 67(2):381–418, 2003.

[Bro05] Gerth S. Brodal. Finger search trees. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 11, page 11. CRC Press, 2005.

[BST85] S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14:545–568, 1985.

[BY76] J.L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[CMSS00] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay Sorting log n-Block Sequences. *Siam J. Comput.*, 30:1–43, 2000.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The Proof. *Siam J. Comput.*, 30:44–85, 2000.

[DHIP04] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic Optimality–Almost. *FOCS*, 2004.

[DS09] Jonathan C. Derryberry and Daniel D. Sleator. Skip-splay: Toward achieving the unified bound in the bst model. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS 2009)*, pages 194–205, Berlin, Heidelberg, 2009. Springer-Verlag.

[Elm04] Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314:459–466, 2004.

[Fle93] Rudolf Fleischer. A simple balanced search tree with o(1) worst-case update time. In *ISAAC '93: Proceedings of the 4th International Symposium on Algorithms and Computation*, pages 138–146, London, UK, 1993. Springer-Verlag.

[Fre75] M. L. Fredman. Two applications of a probabilistic search technique: sorting x + y and building balanced search trees. *Proc. Seventh ACM symposium on Theory of Computing*, pages 240–244, 1975.

[Geo04] George F. Georgakopoulos. Splay trees: a reweighing lemma and a proof of competitiveness vs. dynamic balanced trees. *Journal of Algorithms*, 51(1):64–76, April 2004.

[Geo05] George F. Georgakopoulos. How to splay for loglogn-competitiveness. In Sotiris E. Nikoletseas, editor, *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005*, 2005.

[GMPR77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60, New York, NY, USA, 1977. ACM Press.

[GW77] A. M. Garsia and M. L. Wachs. A new algorithm for minimal binary search trees. *SIAM Journal on Computing*, 6:622–642, 1977.

[Har80] D. Harel. Fast updates of balanced search trees with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980.

[HKT79] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary search trees optimum under various criteria. *SIAM J. Appl. Math.*, 37:246–256, 1979.

[HL79] D. Harel and G.S. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979.

[HM82] S. Huddleston and K. Mehlhorm. A new data structure fore representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM J. Appl. Math.*, 21:514–532, 1971.

[Hu82] T. C. Hu. *Combinatorial Algorithms*. Addison-Wesley, Reading, MA, 1982.

[Iac01] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[Iac02] John Iacono. Key independent optimality. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 25–31, London, UK, 2002. Springer-Verlag.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[Knu73] D. E. Knuth. *The art of computer programming, vol. 3: Sorting and searching*. Addison-Wesley, Reading, MA, 1973.

[Kor81] J. F. Korsch. Greedy binary search trees are nearly optimal. *Inform. Proc. Letters*, 13:16–19, 1981.

[KV81] H. P. Kriegel and V. K. Vaishnavi. Weighted multidimensional b-trees used as nearly optimal dynamic dictionaries. *Mathematical Foundations of Computer Science*, 1981.

[Meh75] K. Mehlhorn. Nearly optimal binary search trees. *Acta Inform.*, 5:287–295, 1975.

[Meh79] K. Mehlhorn. Dynamic binary search. *SIAM Journal on Computing*, 8:175–198, 1979.

[Pet08] Seth Pettie. Splay trees, davenport-schinzel sequences, and the deque conjecture. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA 2008)*, pages 1115–1124, 2008.

[Pug89] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Dept. of Computer Science, University of Maryland, 1989.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Sun89a] R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. *Proceedings of the 13th Symposium on Foundations of Computer Science*, pages 555–559, 1989.

[Sun89b] R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. Technical Report 427, Courant Institue, New York University, 1989.

[Sun92] R. Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12:95–124, 1992.

[Tar85] R. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

[Tsa86] Athanasios K Tsakalidis. Avl-trees for localized search. *Inf. Control*, 67(1-3):173–194, 1986.

[TvW88] Robert E. Tarjan and Christopher van Wyk. An o (n log log n)-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17(1):143–178, 1988.

[Unt79] K. Unterauer. Dynamic weighted binary search trees. *Acta Inform.*, 11:341–362, 1979.

[Wan06] Chengwen Chris Wang. *Multi-splay trees*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2006. Adviser-Daniel Sleator.

[WDS06] C. C. Wang, J. Derryberry, and D. D. Sleator. O(log log n) competitive dynamic binary search tree. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2006.