$O(\log \log n)$-Competitive Dynamic Binary Search Trees*

Chengwen Chris Wang Jonathan Derryberry Daniel Dominic Sleator
{chengwen, jonderry, sleator}@cs.cmu.edu
Computer Science Department, Carnegie Mellon University

Abstract

The Dynamic Optimality Conjecture [ST85] states that splay trees are competitive (within a constant competitive factor) among the class of all binary search tree (BST) algorithms. Despite 20 years of research this conjecture is still unresolved. Recently, Demaine *et al.* [DHIP04] suggested searching for alternative algorithms which have small but non-constant competitive factors. They proposed *Tango*, a BST algorithm which is nearly dynamically optimal – its competitive ratio is $O(\log \log n)$ instead of a constant. Unfortunately, for many access patterns, such as random and sequential, Tango is worse than other BST algorithms by a factor of $\log \log n$.

In this paper, we introduce the multi-splay tree (MST) data structure, which is the first $O(\log \log n)$-competitive BST to simultaneously achieve $O(\log n)$ amortized cost and $O(\log^2 n)$ worst-case cost per query. We also prove the sequential access lemma for MSTs, which states that sequentially accessing all keys takes linear time. Thus, MSTs are $O(\log \log n)$-competitive like Tango but, unlike Tango, require only $O(\log n)$ amortized time per access in an arbitrary sequence and only $O(1)$ amortized time per access during a sequential access sequence.

Furthermore, we generalize the standard framework for competitive analysis of BST algorithms to include updates (insertions and deletions) in addition to queries. In doing so, we extend the lower bound of Wilber [Wil89] and Demaine *et al.* [DHIP04] to handle these update operations. We show how MSTs can be modified to support these update operations and be $O(\log \log n)$-competitive in the new framework while maintaining the rest of the properties above.

1 Introduction

A splay tree [ST85] is a self-adjusting form of binary search tree where each time a node in the tree is accessed, that node is moved to the root according to an algorithm called *splaying*. In a splay tree, all accesses and updates (e.g. insert, delete, join, split) are accomplished by using the splaying algorithm. Splay trees have been shown to have a number of remarkable properties, including the Balance Theorem [ST85], the Static Optimality Theorem [ST85], the Static Finger Theorem [ST85], the Working Set Theorem [ST85], the Scanning Theorem [Sun89], the Sequential Access Theorem [Tar85, Sun92, Elm04], and the Dynamic Finger Theorem [CMSS00, Col00].

The Dynamic Optimality Conjecture [ST85] states that on any sequence of accesses, the cost of splay trees on that sequence is within a constant factor of any other BST algorithm for processing that sequence of accesses. All of the properties of splay trees cited in the above paragraph are special cases of Dynamic Optimality. Dynamic Optimality is equivalent to the statement that splay trees are c-competitive [ST85] for some constant c. Resolving this conjecture seems difficult – it has defied concerted attempts to solve it for about 20 years.[1]

Demaine *et al.* [DHIP04] suggested searching for alternative binary search tree algorithms that have small but non-constant competitive factors. They proposed *Tango*, a BST algorithm that achieves dynamic optimality with a competitive ratio of $O(\log \log n)$. They achieved this ratio by making use of a variation of Wilber's first lower bound on the cost of an access sequence [Wil89].

We introduce the multi-splay tree (MST[2]) data structure. In addition to being $O(\log \log n)$-competitive, MSTs also simultaneously achieve $O(\log n)$ amortized cost and $O(\log^2 n)$ worst-case cost per query. We also

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

[1] It is even apparently difficult to obtain any (online exponential time [BCK02] or offline polynomial time) binary search tree algorithm that is c-competitive.

[2] Throughout this paper, MST always means multi-splay tree and not minimum spanning tree.

SODA '06, January 22-26, Miami, FL
 ISBN 0-89871-605-5/06/01

prove the sequential access lemma for MSTs, which states that sequentially accessing all keys takes linear time. Although MSTs are quite similar to Tango, they do achieve improved performance in several ways. Tango does not have the sequential access property, nor does it achieve $O(\log n)$ amortized cost per operation.[3]

The original framework in which the $O(\log \log n)$-competitive bounds for Tango and MSTs are proven does not allow for insertions or deletions. We generalize this framework to include these update operations, and extend the lower bound appropriately. We also show how to modify the MST data structure to handle insertions and deletions, and prove that it remains $O(\log \log n)$-competitive while preserving the rest of the properties mentioned above.

1.1 Model In order to discuss optimality of BST algorithms, we need to give a precise definition for this class of algorithms, and their costs. The model we use is that implied by Sleator and Tarjan [ST85] and developed in detail by Wilber [Wil89]. A static set of n keys is stored in the nodes of a binary tree. The keys are from a totally ordered universe, and they are stored in symmetric (left to right) order. Each node has a pointer to its left child, to its right child, and to its parent. Also, each node may keep a constant[4] amount of additional information but no additional pointers.

A BST algorithm is required to process a sequence of queries $\sigma = \sigma_1, \ldots, \sigma_m$. Each access σ_i is a query to a key $\hat{\sigma_i}$ in the tree[5], and the requested nodes must be accessed in the specified order. Each access starts from the root and follows pointers until the desired node (the one with key $\hat{\sigma_i}$) is reached. The algorithm is allowed to update the fields and pointers in any node that it touches along the way. The cost of the algorithm to satisfy the sequence of queries is defined to be the number of nodes that it touches. Finally, we do not allow any information to be preserved from one access to the next, other than in the nodes' fields, and a pointer to the root of the tree. It is easy to see that this definition is satisfied by all of the standard BST algorithms, such as red-black trees and splay trees. This model does not handle insertions and deletions, but we generalize it to handle them in Section 6.

[3] On a random access pattern, Tango uses $\Theta(\log n \log \log n)$ time per query. However, it has been pointed out that Tango can be modified to achieve $O(\log n)$ amortized performance, at a cost of changing the worst-case to $\Theta(n)$.

[4] To be consistent with standard conventions, here we consider $O(\log n)$ bits to be "constant."

[5] This model is only concerned with successful searches.

1.2 Interleave Lower Bound Given an initial tree T_0 and an m-element access sequence σ, for any BST algorithm satisfying these requests there is a cost, as defined above. Thus, we can define $\mathrm{OPT}(T_0, \sigma)$ to be the minimum cost of any BST algorithm for satisfying these requests starting with initial tree T_0. Wilber [Wil89] derived a lower bound on $\mathrm{OPT}(T_0, \sigma)$, and this was modified to be the *interleave bound* by Demaine *et al.* [DHIP04].

Let $\mathrm{IB}(P, \sigma)$ denote the interleave lower bound on the cost of accessing the sequence σ, where P is a BST (later called a *reference tree*) over the same set of keys as T_0. Define $\mathrm{IB}(P, \sigma) = \sum_{v \in P} \mathrm{IB}(P, \sigma, v)$, where for each node v, $\mathrm{IB}(P, \sigma, v)$ is defined as follows. First, restrict σ to the set of nodes in the subtree of P rooted at v (including v). Next, label each access in this restricted σ as either "left" (or "right") depending on whether the accessed element is in the left subtree (including v) or right subtree of v. Now, $\mathrm{IB}(P, \sigma, v)$ is the number of times the labels switch.

THEOREM 1.1. *[Wil89, DHIP04]*

$$OPT(T_0, \sigma) \geq IB(P, \sigma)/2 - O(n) + m$$

Culik and Wood [CW82] proved that the number of rotations needed to change any binary tree of n nodes into another one is at most $2n - 2$.[6] It follows that $\mathrm{OPT}(T_0, \sigma)$ differs from $\mathrm{OPT}(T_0', \sigma)$ by at most $2n - 2$. Thus, as long as $m = \Omega(n)$, the initial tree is irrelevant. We shall make this assumption.

1.3 The Access Lemma for Splay Trees Sleator and Tarjan [ST85] proved that the amortized cost of splaying a node is bounded by $O(\log n)$ in a tree of n nodes. By the use of the flexible potential described below, they proved tighter bounds on the amortized cost of splaying for access sequences that are non-uniform (e.g., the Static Optimality Theorem). This framework is essential for the analysis of multi-splay trees.

For an arbitrary positive weight function w over the nodes of a splay tree, they defined the size $s(v)$ of node v to be $\sum_{v \in subtree(v)} w(v)$, the sum of the weights of all nodes in v's subtree. They defined the potential of the tree to be $\sum_{v \in V} \lg s(v)$, where V is the set of nodes in the splay tree.

As a measure of the cost (running time) of a splaying operation, they used the distance from the node being splayed to the root plus 1. With these definitions, Sleator and Tarjan proved the following theorem about the amortized cost of splaying.

[6] Sleator, Tarjan and Thurston [STT86] subsequently showed that only $2n - 6$ rotations (for $n \geq 10$) are necessary.

THEOREM 1.2. *(Access Lemma) [ST85] The amortized time to splay a node v in a tree currently rooted at r is at most $O(1 + \lg(s(r)/s(v)))$.*

THEOREM 1.3. *(Generalized Access Lemma) Given a pointer to an ancestor node a, the amortized time to splay a node v with respect to an ancestor a in the same splay tree is at most $O(1 + \lg(s(a)/s(v)))$.*

The main difference between this and the original access lemma is that we are allowed to stop at any ancestor a. Its truth follows from the proof of the original access lemma because that proof does not require splaying to go all the way to the root.

2 The Multi-Splay Tree Data Structure

Consider a *balanced*[7] BST P made up of n nodes, which we will refer to as the *reference tree*. Because P is balanced, the depth of any node in P is at most $2\lg(n+1)$. (The depth of the root is defined to be 1.) Each node in the reference tree has a *preferred child*. The structure of the reference tree is static (but we will generalize it to support insert and delete in Section 6), except that the preferred children will change over time, as explained below. We call a maximal chain of preferred children a *preferred path*. The nodes of the reference tree are partitioned into approximately $n/2$ sets, one for each preferred path. The reference tree is not explicitly part of our data structure, but is useful in understanding how it works.

A multi-splay tree is a BST T (over the same set of n keys contained in the reference tree P) that evolves over time, and preserves a tight relationship to the reference tree. Each edge of a multi-splay tree is either *solid* or *dashed*. We call a maximal set of vertices connected by solid edges a *splay tree*. There is a one-to-one correspondence between the splay trees of a multi-splay tree and the preferred paths of its reference tree. The set of nodes in a splay tree is exactly the same as the set of nodes in its corresponding preferred path. In other words, at any point in time a multi-splay tree can be obtained from its reference tree by viewing each preferred edge as solid, and executing a series of rotations on only the solid edges.[8]

Each node of a multi-splay tree T has several fields in it, which we enumerate here. First of all, it has the usual *key* field, and pointers *leftChild*, *rightChild*, and *parent*. Although the reference tree P is not explicitly represented in T, each node stores information related to P. In each node's *refDepth* field, we keep its depth in P.[9] Note that every node in the same splay tree has a different depth in P. In addition, each node v stores the minimum depth of all of the nodes in *splaySubtree*(v) in its *minDepth* field (*splaySubtree*(v) contains all of the nodes in the same splay tree as v that have v as an ancestor, including v). Finally, to represent the solid and dashed edges, each node has an *isRoot* boolean variable that indicates if the edge to its parent is dashed.

3 The Multi-Splay Algorithm

In this section, we first explain the algorithm assuming we have the reference tree P, then we explain how to implement the corresponding operations in our actual representation T.

As stated above, the preferred edges in P evolve over time. A *switch* at a node just swaps which child is the preferred one. For each access, switches are carried out, from the bottom up, so that the accessed node v is on the same preferred path as the root of P. In addition, one last switch is carried out on the node that is accessed.

In other words, traverse the path from v to the root doing a switch at each parent of a non-preferred child on the path, and then finally switch v. That is the whole algorithm from the point of view of the reference tree. The tricky part is to do it without the reference tree. Note that if the multi-splay algorithm did not make the final switch on the queried node, the number of switches caused by single query would equal the increase in interleave bound. With the extra switch, the amortized number of switches only increases by at most 2 per query.

Unfortunately, P is not our representation, T is. To achieve $O(\log\log n)$-competitiveness, we can only afford to spend $O(\log\log n)$ amortized time per switch. As shown below, we can simulate a switch in P with at most three splay operations, and two changes of *isRoot* bits in T.

More specifically, suppose we want to switch y's preferred child from left to right. To understand the effect of this, temporarily make both children of y preferred. Now, consider the set S of nodes in P reachable from y using only preferred edges. This set can be partitioned into four parts: L, those nodes in the left subtree of y in P; R, those nodes in the right subtree of y in P; U those nodes above y in P; and y. When set S is sorted by key, L and R form contiguous

[7] By "balanced" we mean that every subtree t has height at most $2\lg(|t|)$

[8] An example of P and T is shown in the extended version of the paper at http://www.cs.cmu.edu/∼chengwen/paper/MST.pdf.

[9] Note that this quantity is static in our initial description of multi-splay trees, but becomes dynamic in Section 6 when we extend multi-splay trees to support insert and delete.

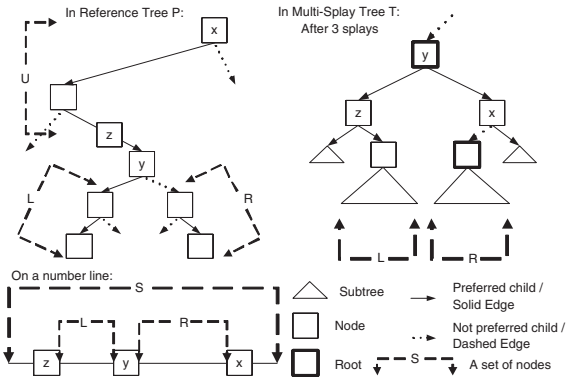

Figure 1: Graphical representations of S, U, L, R, x, y, and z during a single left-to-right switch.

regions of keyspace, separated by y (See Figure 1).

Let us see what this means in a multi-splay tree T. The splay tree in T containing y consists of nodes $L \cup U \cup \{y\}$. After the switch it consists of $R \cup U \cup \{y\}$. To do this transformation we need to remove L and add in R. Because L and R are contiguous regions in the symmetric ordering, we can use splaying to efficiently split off the tree containing L and join in the tree containing R. We first find z, the predecessor of L in S, using the *minDepth* field. (Note that z is the largest node less than y with depth less than y, and z must be a member of U.) Then, we splay y and splay z until z becomes the left child of y. This ensures that the set of nodes in the right subtree of z is L. Thus, we mark the right child of z as a root in order to remove L from y's splay tree. As for joining in R, we simply splay the successor of y (called x) in U until x becomes the right child of y, so that unmarking the *isRoot* bit of the left child of x joins in R.[10]

However, an access is not just a single switch in P, it is a sequence of switches. For the purposes of our running time analysis, we do these from bottom to top. Also, we perform a final switch on the accessed node to pay for the traversal from the root of T to the accessed node. Notice that this final switch brings the accessed node to the root of T.[11]

This description has glossed over a number of subtle details, like how to determine if the switch is from left to right or from right to left. In addition, we have not discussed the boundary cases such as when z or x does not exist.

In more detail, when serving a query σ_i for key $\hat{\sigma}_i$, we traverse the MST T to find $\hat{\sigma}_i$. As we traverse, we maintain $v_j = pred(\hat{\sigma}_i)$ and $w_j = succ(\hat{\sigma}_i)$ for the j^{th} splay tree encountered, where $pred(\hat{\sigma}_i)$ denotes the predecessor of $\hat{\sigma}_i$ and $succ(\hat{\sigma}_i)$ denotes the successor of $\hat{\sigma}_i$. Notice that the switch in the j^{th} splay tree must occur at the deeper of v_j and w_j in the reference tree (this is where the access path in the reference tree diverges from the preferred path corresponding to the j^{th} splay tree). Let α_j be the node we switch, and β_j be the other node. To decide the direction of the switch, observe that if $\alpha_j < \beta_j$, we switch from left to right. Otherwise, we switch from right to left. After we finish all of these switches from the bottom up, we switch $\hat{\sigma}_i$.

4 Running Time Analysis for an Arbitrary Sequence

For the purpose of analysis, we define the potential of a multi-splay tree T as follows. If each node v has an arbitrary positive *weight* $w(v)$, define the *size* $s(v)$ of node v to be $\sum_{v \in splaySubtree(v)} w(v)$ (i.e., the sum of the weights of all descendants of v in T reachable by traversing only solid edges). Define the potential of the tree to be $\sum_{v \in T} \lg s(v)$.

THEOREM 4.1. *For any query in a multi-splay tree, the worst-case cost is* $O(\log^2 n)$.

Proof. This follows from the fact that to query a node, we visit at most $O(height(P))$ splay trees. Because the size of each splay tree is $O(\log n)$, the total number of nodes we can possibly touch is $O(\log^2 n)$.

THEOREM 4.2. *For an arbitrary access sequence* $\sigma = \sigma_1 \cdots \sigma_m$ *in a multi-splay tree with* n *elements, the cost of* σ *is* $O(OPT(\sigma) * \log\log n)$.

Proof. The total number of switches in a multi-splay tree during σ is at most $IB(P, \sigma) + 2m$ [DHIP04] (the extra $2m$ term results from the additional switch on $\hat{\sigma}_i$, which may need to be undone later in the access sequence), so it suffices to show that the amortized cost of each switch is $O(\log\log n)$.

Each switch at an arbitrary node y (with corresponding x and z[12]) consists of up to 3 splays followed

[10] To prove MSTs use only $O(\log n)$ amortized cost per query, we can only afford to splay nodes that are in $\{y\} \cup U$. As a result, we cannot split L by splaying y and then splaying l, the leftmost node in L (stopping at the left child of y). This technique would have been analogous to the technique used in [DHIP04].

[11] If the accessed node $\hat{\sigma}_i$ has one or fewer children, then the final switch on $\hat{\sigma}_i$ still induces the corresponding splays, but no root marking will occur.

[12] Here and throughout the paper, we use x_i to denote a node whose child will be unmarked as a root during the i^{th} switch of an access and z_i to denote a node whose child will be marked as a root during the i^{th} switch of an access. If we omit the subscript i, then we are referring to any switch.

by up to 2 changes to *isRoot* bits. To analyze the amortized cost of each of these operations, we invoke the access lemma for splay trees, and recall that it uses the following potential function for a splay tree T_S : $\sum_{v \in T_S} \log s(v)$. The analysis in this sub-section assumes uniform constant weights are used for all nodes in all splay trees comprising a multi-splay tree.

The amortized cost of each of the 3 splays is $O(\log s(r))$, where r is the root of the splay tree corresponding to y's preferred path in the reference tree. Because $s(r) = O(\log n)$, the amortized cost of the 3 splays is $O(\log \log n)$.

The amortized cost of marking *child*(z) (if it exists[13]) is $O(1)$ because it does not increase the size of any subtrees in any splay trees, so the overall potential does not increase. The amortized cost of unmarking *child*(x) (if it exists) is $O(\log \log n)$ because the only nodes whose size increase are x and y, and the increase in each of their sizes is bounded by the size of the splay tree rooted at *child*(x), which is $O(\log n)$.

To summarize, the amortized cost of each switch is:

$$\begin{aligned} \text{Amortized cost} &= \text{cost of splays} \\ &\quad + \text{root marking cost} \\ &\quad + \text{root unmarking cost} \\ &= O(\log \log n + 1 + \log \log n) \\ &= O(\log \log n). \end{aligned}$$

THEOREM 4.3. *Each query σ_j in a multi-splay tree costs $O(\log n)$ amortized time.*

Proof. To analyze the amortized cost of an access in a multi-splay tree T, we assign a weight $w(v)$ to each node in T according to its depth in the reference tree as follows: $w(v) = 2^{-refDepth(v)}$. One key fact to notice is that this implies that the sum of the weights on a path to a leaf from v (but not including it) is less than $w(v)$.

Again, each switch during an access consists of at most 3 splays, and at most 2 changes to *isRoot* bits. Because the amortized cost of a splay is $O(\log(s(r)/s(v)))$ when v is being splayed in a tree rooted at r, the cost of the 3 splays is at most

$$\log(s(r_i)/s(y_i)) + \log(s(r_i)/s(x_i)) + \log(s(r_i)/s(z_i)),$$

where y_i is i^{th} node being switched going up the multi-splay tree access path to the root of the multi-splay tree and r_i is the root of the splay tree containing y_i (y_1 is the first node switched, and by our convention the splay tree rooted at r_0 contains $\hat{\sigma_j}$).

Because the elements in the splay tree rooted at r_{i-1} comprise a path to a leaf in the reference tree starting below y_i, we have $s(y_i) \geq w(y_i) > s(r_{i-1})$. Moreover, because x_i and z_i are ancestors of y_i in the reference tree, $w(x_i)$ and $w(z_i)$ are both larger than $w(y_i)$ (and, hence, $s(r_{i-1})$). Therefore, the total amortized cost of splaying all x_i during a single access (similar math applies to y and z) is

$$\begin{aligned} \sum_{i=1}^{k} \lg\left(\frac{s(r_i)}{s(x_i)}\right) &\leq \sum_{i=1}^{k} \lg\left(\frac{s(r_i)}{w(x_i)}\right) \leq \sum_{i=1}^{k} \lg\left(\frac{s(r_i)}{w(y_i)}\right) \\ &\leq \sum_{i=1}^{k} \lg\left(\frac{s(r_i)}{s(r_{i-1})}\right) \leq \lg\left(\frac{s(r_k)}{s(r_0)}\right). \end{aligned}$$

Next, we account for the cost of marking and unmarking the root bits of *child*(z_i) and *child*(x_i) respectively. (We refer to the children in $L \cup R$.) First, notice that marking *child*(z_i) as a root reduces the overall potential of the collection of splay trees, so it has $O(1)$ amortized cost per switch. Second, notice that unmarking *child*(x_i) only increases $s(x_i)$ and $s(y_i)$ by at most $w(y_i)$ because the nodes of the tree rooted at *child*(x_i) make up a path to a leaf in the reference tree starting below y_i. Thus, $s(x_i)$ and $s(y_i)$ at most double because $s(y_i) > s(x_i) \geq w(x_i) > w(y_i)$, and potential increases by at most 2. There are $O(height(P))$ switches per access, so the total cost of root marking/unmarking per access is $O(\log n)$ because the reference tree is balanced.

Finally, note that the last switch (on the accessed node) costs $O(\log(s(r_k)/s(\hat{\sigma_j}))$. Because the smallest weight in the tree is at least $2^{-height(P)}$, and the largest size is less than 1, the total amortized cost of each access is,

$$\begin{aligned} \text{Cost} &= \text{Cost of } k \text{ switches} + \text{Cost of root markings} \\ &\quad + \text{Cost of final switch} \\ &= O\left(\log\left(\tfrac{s(r_k)}{s(r_0)}\right) + h(P) + \log\left(\tfrac{s(r_k)}{s(\hat{\sigma_j})}\right)\right) \\ &= O\left(\log\left(\tfrac{1}{2^{-h(P)}}\right) + h(P) + \log\left(\tfrac{1}{2^{-h(P)}}\right)\right) \\ &= O(\log n), \end{aligned}$$

where $h(P) = height(P)$.

To further generalize the above proofs, we define *descendants*(v, P) to be all of the descendants of v in P. We also define *paths*(v, P) to be the set of all paths from a node v in P to any descendant leaf.

THEOREM 4.4. *(Multi-Splay Access Lemma) Let P be any initial reference tree with root r, f be any multiplier greater than 2, and $w(x)$ be any positive weight assignment satisfying the following two conditions:*

[13] As a caveat, we note that determining whether z and x exist can be done in constant time, and if they do exist the cost of finding z and x (and also y) is proportional to their depth in their splay tree. The cost of this traversal can be charged to the splay operations.

$$\begin{aligned} w(v) &\geq \max_{u \in descendants(v,P)} w(u) \\ f * w(v) &\geq \max_{p \in paths(v,P)} \sum_{u \in p} w(u). \end{aligned}$$

Then the running time to access the sequence $\sigma = \sigma_1, \ldots, \sigma_m$ *is amortized*

$$O\left(\sum_{i=1}^{m} \log(w(r)/w(\hat{\sigma_i})) + (\log f) * (IB(P,\sigma) + m)\right).$$

The proof for this theorem[14] is similar to the proofs of the preceding theorems. With this theorem, one can prove $O(\log \log n)$-competitiveness by choosing a balanced reference tree P, setting the weight of every node v to 1, and choosing f to be $O(\log n)$. One can also prove the $O(\log n)$ amortized bound by choosing a balanced reference tree P, setting the weight of node v to be $\frac{1}{2^{refDepth(v)}}$, and choosing f to be $O(1)$. Unfortunately, the two constraints above prevent the use of the Multi-Splay Access Lemma for proving the classical splay tree properties.

5 Sequential Access Takes Linear Time

We begin with several simple lemmas.[15]

LEMMA 5.1. *The cost of a switch is* $O(\log n)$ *worst-case.*

LEMMA 5.2. *During a sequential access of all nodes of* T*, when a node with a left child (in* P*) is accessed, exactly one switch occurs.*

LEMMA 5.3. *In a splay tree* T_S *with root* r *(*r *changes as the root changes), if all splay operations are performed on a connected set of nodes* $S \subseteq T_S$ *, and* $r \in S$*, then the splay algorithm will never rotate any node outside of* S*. (This allows us to analyze the cost of splaying assuming all nodes in* $(T_S \setminus S)$ *do not exist.)*

LEMMA 5.4. *During a sequential access sequence, when accessing nodes from the right ref-subtree* R *of* y*, the multi-splay algorithm touches at most 2 nodes outside of* R*.*

LEMMA 5.5. *In a red-black tree* T_{RB} *with* n *nodes,* $\sum_{v \in T_{RB}} \lg |subtree(v)| = O(n)$*.*

[14]This proof is in the extended version of the paper at http://www.cs.cmu.edu/∼chengwen/paper/MST.pdf.

[15]The proofs are in the extended version of the paper at http://www.cs.cmu.edu/∼chengwen/paper/MST.pdf.

THEOREM 5.1. *In any multi-splay tree* T *of* n *nodes, the cost of the access sequence* $\sigma = \sigma_1, \cdots, \sigma_n$*, where* $\hat{\sigma_i} < \hat{\sigma_{i+1}}$ *is* $O(n)$*.*

Proof. In this proof, we assume that P is a full red-black tree [GS78]. Using the previous lemmas, we can develop a recurrence for the cost of sequential access. First, we define *rightParent*(v) to be p if the left child of p is v. Also, we define the *right ascending path* of v to be the set of nodes u, such that *rightParent*$^*(v) = u$. Finally, we define $A(v)$ to be the size of the right ascending path of v. We analyze the cost of sequentially accessing all of the nodes of an MST T in terms of the cost of sequentially accessing subtrees of P. More specifically, we recursively account for the cost as follows:

$$\begin{aligned} \text{Time}(t) = \; & \text{Time}(\textit{leftRefSubtree}) + \text{Time}(\textit{root}(t)) \\ & + \text{Time}(\textit{rightRefSubtree}), \end{aligned}$$

where t is some subtree of P, and Time(t) is the amortized time used when sequentially accessing the nodes of t *within* the context of sequential access to *all* nodes of T, not just the ones in t.

However, to tightly bound the time for accessing the root of t, we need to incorporate $A(root(t))$. Hence, we define

$$\begin{aligned} \text{Time}(t,a) = \; & \text{Time to sequentially access all nodes} \\ & \text{in } t, \text{ where } A(\textit{root}(t)) = a, \end{aligned}$$

where t is a subtree of P (taken within the context of T's full reference tree, so that t's root may have a non-trivial right ascending path). With this expanded accounting method, the cost of sequentially accessing all of the nodes of T is Time$(P, 1)$.

In general, we can write

$$\text{Time}(t,a) = \text{Time}(t_L, a+1) + \text{Time}(t_R, 1) + O(a + \log|t|),$$

for the case in which *root*(t) is an internal node because *root*(t_L) has a right ascending path with one more node than the path of *root*(t), *root*(t_R) has a right ascending path including just itself, and accessing *root*(t) causes at most one switch by Lemma 5.2, whose running time is $O(a + 1 + \log|t|)$ worst-case because the number of nodes touched during a switch at node *root*(t) is $O(2 + A(root(t)) + \log|t|) = O(A(root(t)) + \log|t|)$. The $O(A(root(t)) + \log|t|)$ bound is true because at most 2 nodes higher in P than *root*(t)'s right ascending path are touched as seen by Lemma 5.4, and the number of nodes in *root*(t)'s splay tree including *root*(t)'s right ascending path and below is $A(root(t)) + height(t)$, which is $O(A(root(t)) + \log|t|)$.

For the base case in which *root*(t) is a leaf in P, we have

$$\text{Time}(t,a) = O(a^2)$$

because at most a switches occur during the access of $root(t)$[16], each of which costs $O(a)$ using similar logic to above, for a total of $O(a^2)$.

To see that this recurrence solves to $O(n)$, we show how to account for all of the $O(a + \log|t|)$ terms and all of the $O(a^2)$ terms so that their costs total $O(n)$. For each t such that $root(t)$ is not a leaf, note that if we spread the $O(a) = O(A(root(t)))$ portion of the cost evenly among the nodes of $root(t)$'s right ascending path, each node v in the reference tree is charged at most $O(height(v)) = O(\log|subtree(v)|)$. Similarly, to account for the $O(a^2)$ cost for each leaf l, we charge $\Theta(k+1)$ to $rightParent^k(l)$ so that each node is charged at most $O(height(v)) = O(\log|subtree(v)|)$. Thus, it suffices to show that $\sum_{v \in P} O(\log|subtree(v)|) = O(n)$, which is true by Lemma 5.5.

6 Making the Data Structure Dynamic

With a slight modification, our data structure can support insert and delete while maintaining all of the properties of Section 4, including $O(\log\log n)$-competitiveness. We describe the details of how to dynamize MSTs in Sections 6.3 and 6.4. To think about what is necessary for supporting insert and delete, it is illustrative to think about the effect of insert and delete on the reference tree. When nodes are inserted into and deleted from the reference tree we need to maintain the invariants that that the tree is balanced and that every internal node has exactly one preferred child. We meet the balance requirement by allowing rotations on the reference tree P (after insertion and deletion), and making P a dynamic red-black tree. We meet the single preferred child requirement by making a constant number of switches prior to each rotation. Because the reference tree is implicitly maintained, we need to be able to simulate the update operations over the reference tree (e.g., rotations, pointer traversals) efficiently. Simulating each of these operations turns out to cost $O(\log\log n)$ amortized time in an MST, so it is important that the corresponding reference tree requires only $O(m)$ virtual traversals and virtual rotations during a sequence of m operations. (Finding the *location* of the update does *not* involve virtual traversals.) Red-black trees meet this requirement because they require only $O(1)$ amortized time to rebalance after an insert or delete [Tar83].

6.1 Defining Competitive Analysis in a Dynamic BST Before we can argue about the competitiveness of dynamic multi-splay trees, we must introduce an intuitive definition of what it means for a dynamic BST to be competitive. We assume an arbitrary dynamic BST algorithm A must execute a sequence of operations $\sigma = \sigma_1, \ldots, \sigma_m$, each of which is $query(\hat{\sigma}_i)$, $insert(\hat{\sigma}_i)$, or $delete(\hat{\sigma}_i)$. For each σ_i, we assume A must pay the following costs:

- To execute $query(\hat{\sigma}_i)$, it must pay for touching each node on the path from the root to $\hat{\sigma}_i$.
- To execute $insert(\hat{\sigma}_i)$, it must insert the node at a leaf and must pay for the traversal to get there. This is reasonable because A must search for $\hat{\sigma}_i$ to realize its BST does not contain $\hat{\sigma}_i$.
- To execute $delete(\hat{\sigma}_i)$, it must pay for accessing $\hat{\sigma}_i$ and for performing rotations until $\hat{\sigma}_i$ has no children (at which time, the node can be removed).[17]

During (or after) each operation, a BST algorithm may perform any rotations it wishes at a cost of one per rotation. The cost of an operation is simply the total number of nodes touched, plus the number of rotations. Without insert and delete, this definition would be identical to the one in Section 1.1. From this point onward, we use $\mathrm{OPT}(\sigma)$ to refer to the cost of an optimal dynamic BST algorithm serving σ.

6.2 Dynamic Interleave Lower Bound With our new definitions, we must prove a new lower bound for $\mathrm{OPT}(\sigma)$. Fortunately, techniques similar to those in [Wil89] suffice. Our new lower bound is an extension of the one in [DHIP04], which is a variant of Wilber's first lower bound.

As in the original definition of the interleave bound, for each node v in the initial reference tree P_0, we track if the last query in $refSubtree(v)$ is in either $L^v = leftRefSubtree(v) \cup \{v\}$ or $R^v = rightRefSubtree(v)$. Whenever the tracking for a node changes, we increment the dynamic interleave bound, $\mathrm{DIB}(\rho, \sigma)$, by one. For an insert of v, we add the cost of querying $pred(v)$ followed by $succ(v)$ (because both of these nodes must be touched to insert v at a leaf). For a delete of v, we add the cost of querying $pred(v)$, v, and $succ(v)$ in succession because all three of these nodes must be touched in order to rotate v to a leaf of the BST. Whenever we rotate a node v, we reset the tracking of v and $refParent(v)$ to L^v but do not increase the interleave bound. Without insertions, deletions, and rotations, this definition would be identical to the original interleave bound.

[16] Because the deepest left ancestor v of $root(t)$ was just queried, there is always a preferred path from the root of P to v, and the number of nodes between v and $root(t)$ is at most a.

[17] In this model, we do not allow BSTs to swap nodes and contract edges during deletion. This implies that it must additionally pay for accessing both $pred(\hat{\sigma}_i)$ and $succ(\hat{\sigma}_i)$. As a result, this model is slightly more restrictive.

THEOREM 6.1. **Dynamic Interleave Bound**[18] *For a sequence of operations $\sigma = \sigma_1, \ldots, \sigma_m$ where each σ_i is a query, insert, or delete, the cost of an arbitrary BST algorithm A on σ is at least $\Omega(DIB(\rho, \sigma)/2 - n - 2k + cm)$, where n is the number of nodes in P_m, $\rho = \rho_1, \ldots, \rho_m$ is a sequence of changes to P, where each ρ_i contains a* sequence *of rotation operations to be performed on P (insertions and deletions in P correspond to those in σ), and k is the number of rotate operations in ρ (i.e., $k = \sum_{i=1}^{m}(\#$ of rotations in $\rho_i)$).*

In the Dynamic Interleave Bound reference tree, we assume deletion of node v is accomplished as in [Tar83], by "splicing out" v unless it has two non-null children, in which case v is swapped with its predecessor and then spliced out.[19]

The operations ρ_i are the changes to P that occur between successive operations of σ. (For MSTs ρ_i represents the rebalancing rotations performed on its reference tree following an insert or a delete.) Different ρ sequences give different lower bounds on the cost of executing σ.

6.3 Simulating Reference Tree Traversals and Rotations To simulate a reference tree pointer traversal from node v in an MST T with reference tree P, we need only to search for the relevant parent or child node, which can be accomplished if we add 3 new fields to store the values of the parent and children of each node in the reference tree. The cost of this search can be paid for by performing a constant number of switches (notice that the path from v to v's child or parent in the reference tree spans at most two splay trees), for a total amortized cost of $O(\log \log n)$. Essentially, each path we traverse in the MST will be splayed, which ensures the amortized cost bound. We omit the details for brevity.

To simulate a right rotation of a node v over its parent in the reference tree, a multi-splay tree first ensures that v's preferred child is its right child, and v's parent's preferred child is its left by performing either 1 or 2 switches on v and v's parent. By meeting these requirements T ensures that if its reference tree satisfies the preferred path property before the rotation, it will still satisfy that property after the rotation, as seen in Figure 2.

We also need to be able to quickly update the fields in each of T's nodes v when a virtual rotation is performed in P. Recall that we store *refDepth*

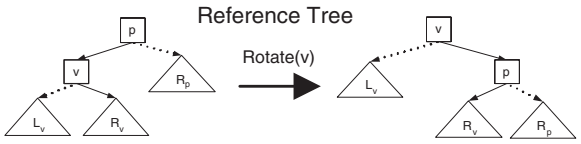

Figure 2: A rotation on v in the reference tree. For a right (left) rotation, we must make sure v's preferred child is right (left), and p's preferred child is v.

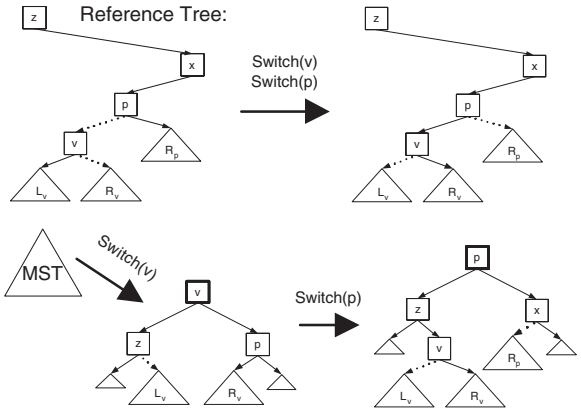

Figure 3: Observe that after we call *switch*(v) and *switch*(p), the sets of nodes in L_v and R_p form two subtrees in an MST. A rotation of v over p in the reference tree decreases the depth value of each of the nodes in L_v by one, and increases the depth value of each of the nodes in R_p by one (Shown in Figure 2). Because L_v and R_p are grouped together by the switches, the updates in depth values cost $O(1)$ after performing the switches.

(the depth of v in the reference tree) and *minDepth* (the minimum *refDepth* of all the nodes in v's splay subtree). To update these values efficiently, we do not store the values explicitly. Instead, in v we store *refDepth*(v) − *refDepth*(*parent*(v)) and *minDepth*(v) − *minDepth*(*parent*(v)), except if v is the root of T, in which case it simply stores its *refDepth* and *minDepth*. This is analogous to the technique used in link-cut trees [ST85].

Let v be the node we rotate in the reference tree P (and the corresponding node in the MST T). Let p be the parent of v in P. Without loss of generality, we assume v is the left child of p. At first glance, a rotation of v over p in P changes the *refDepth* value for many nodes, so it would be difficult to update. However, the sets of nodes whose depths change constitute two subtrees in the reference tree. More specifically, the *refDepth* of each node in *leftRefSubtree*(v), L_v, decreases by one, while the *refDepth* of each node in *rightRefSubtree*(p),

[18] The proofs are in the extended version of the paper at http://www.cs.cmu.edu/∼chengwen/paper/MST.pdf.

[19] Although our model for BST deletion does not allow such swapping/splicing, MSTs will only be *simulating* them while adhering to our dynamic BST model.

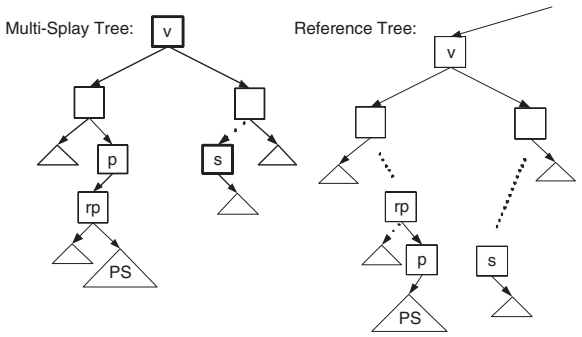

Figure 4: An example of what an MST looks like during deletion of node $\hat{\sigma}_i$, with $v = \hat{\sigma}_i$, $p = pred(\hat{\sigma}_i)$, $s = succ(\hat{\sigma}_i)$, $rp = refParent(pred(\hat{\sigma}_i))$ after the sequence $query(p)$, $switch(rp)$, $query(p)$, $query(s)$, and $query(v)$. Here we show the case in which $rp < p$.

R_p, increases by one. Using this observation, we can decrease the depth value of all of the nodes in v's ref-subtree by executing $switch(v)$ and $switch(p)$ in T, which isolates L_v and R_p as shown in Figure 3 so we can change the difference value at a single node to decrease (or increase) the stored *refDepth* of each node in L_v (or R_p) by one. This method can be used for the *minDepth* field as well.

Hence, a rotation in P can be simulated in T using a constant number of switches and field updates, so its amortized cost is $O(\log \log n)$ if the reference tree is balanced.

6.4 Implementing Insertion and Deletion To insert $\hat{\sigma}_i$, we perform a normal BST insert, access the inserted node, and then rebalance the reference tree using amortized $O(1)$ simulated rotations and pointer traversals. We also insert it into the virtual reference tree by finding its *refParent* on the way down $\hat{\sigma}_i$'s access path in T ($refParent(\hat{\sigma}_i)$ is the node of maximum *refDepth* on the access path) and setting the appropriate fields of $\hat{\sigma}_i$ and $refParent(\hat{\sigma}_i)$.

For deletion, we consider the case in which $\hat{\sigma}_i$ has two children in P (the other two cases are simpler). Before rebalancing the reference tree using amortized $O(1)$ simulated rotations and pointer traversals, we must first swap $\hat{\sigma}_i$ with $pred(\hat{\sigma}_i)$ and splice out $\hat{\sigma}_i$ using a constant number of switches, rotations, and field updates in addition to a constant number of accesses to $pred(\hat{\sigma}_i)$, $\hat{\sigma}_i$, and $succ(\hat{\sigma}_i)$, which will be justified in Section 6.5. To accomplish this, we first perform the sequence: $query(pred(\hat{\sigma}_i))$, $switch(refParent(pred(\hat{\sigma}_i)))$, $query(pred(\hat{\sigma}_i))$, $query(succ(\hat{\sigma}_i))$, and $query(\hat{\sigma}_i)$. Notice that this sequence adheres to our cost specification, and results in an MST that looks like the one in Figure 4.

There are two important aspects of this MST. First, $pred(\hat{\sigma}_i)$, $\hat{\sigma}_i$, and $pred(\hat{\sigma}_i)$ are located close together, so that $O(1)$ rotations suffices to make $\hat{\sigma}_i$ a leaf so that it can be deleted. Second, $pred(\hat{\sigma}_i)$'s subtree is isolated in its own subtree of the MST (the subtree PS in Figure 4), so that we can decrement all of its nodes' *refDepth* and *minDepth* fields in $O(1)$ time just by changing the fields in the root of PS. We omit most of the details, but remark that once $\hat{\sigma}_i$ is rotated to a leaf and deleted, the depth values of PS are adjusted, and $pred(\hat{\sigma}_i)$ has its fields set so that it takes $\hat{\sigma}_i$'s place in the reference tree, we need only to recompute the *minDepth* fields of the ancestors of $pred(\hat{\sigma}_i)$ and reset the parent and child value fields (used during virtual pointer traversals as mentioned in Section 6.3) of the nodes whose parents or children change in the reference tree (i.e., the parents and children of $\hat{\sigma}_i$ and $pred(\hat{\sigma}_i)$ in the reference tree), requiring only a constant number of field updates and reference pointer traversals, each costing $O(\log \log n)$.

6.5 Proof of Running Time Bounds Without insert and delete, the analysis in Section 4 applies. For the insert and delete operations, only $O(1)$ amortized reference tree rotations are required to rebalance the tree, so that the total amortized cost is $O(\log \log n)$, which does not affect the $O(\log \log n)$-competitiveness of the operations or the $O(\log n)$ amortized cost of them.

For each insert operation, the number of switches, which each cost amortized $O(\log \log n)$, performed during the insert's query is equal to the increase in the Dynamic Interleave Bound. The rest of the $O(\log \log n)$ amortized cost is charged to the minimum cost of 1 per operation in any BST algorithm.

For each delete operation, the number of switches performed during the queries to $pred(\hat{\sigma}_i)$, $\hat{\sigma}_i$, and $succ(\hat{\sigma}_i)$ is bounded by 3 times the maximum number of switches caused by queries to these 3 nodes plus a constant number to account for the extra switches performed on the queried nodes and the lowest common ancestors between pairs of these 3 nodes in the reference tree (and additionally, in our case, a switch on $refParent(pred(\hat{\sigma}_i))$). The constant number of extra switches and the rest of the additional $O(\log \log n)$ amortized cost (the virtual traversals, virtual rotations, MST rotations, field updates, and the actual deletion) is charged to the minimum cost of 1 per operation in any BST algorithm.

Finally, because the number of rotations performed on the reference tree is $O(1)$ worst-case per operation, we can afford to pay for the $-2k$ term in the lower bound with the $+cm$ term (for a suitable constant c), it follows that dynamic MSTs are $O(\log \log n)$-competitive. Also, because the strongest assumption about the balance of

the reference tree in Sections 4 and 5 was that the reference tree was a red-black tree, all of the proofs in Sections 4 and 5 still apply.

7 Conclusions and Future Work

In this paper we showed that multi-splay trees achieve $O(\log \log n)$-competitiveness, $O(\log n)$ amortized cost, and $O(\log^2 n)$ worst-case cost per query. We then combined these proofs to show the access lemma for multi-splay trees – a parameterizable theorem for analyzing multi-splay tree query sequences. We also proved that sequential access in multi-splay trees costs $O(n)$.

We extended the interleave lower bound to allow insertions and deletions, and showed how to carry out these operations in multi-splay trees. We proved that the same bounds quoted above for the query-only case apply when insertions and deletions are also allowed if we use a red-black tree with $O(1)$ amortized rebalancing cost for the reference tree.

The multi-splay algorithm is similar to splaying, but differs in a few important ways. Consider modifying the algorithm so that it does not splay z_i. In this modified algorithm, an access to a node v is then a series of partial splays (ones that stop before getting all the way to the root) of nodes on v's path to the root. Because of this similarity to splay trees, these partial splays do not keep MSTs balanced.

However, with the additional splay on z_i, MSTs become somewhat balanced (i.e., their maximum depth becomes bounded by $O(\log^2 n)$). Moreover, one way of thinking about the marking of root bits is that it effectively "removes" from the tree a large amount of weight. This allows us to prove tighter bounds on the competitiveness than can be proven for splay trees.

Given the similarities between multi-splay trees and classical splay trees, it is natural to ask whether splay trees are also $O(\log \log n)$-competitive. It is also natural to ask whether multi-splay trees share some of the other nice properties of splay trees, such as static optimality.

As far as we know, multi-splay trees may be dynamically optimal. Is this true? One big difficulty in addressing this problem is the lack of tight lower bounds on the cost of accessing a sequence. The static interleave bound is insufficient, because it is known to be off by a factor of $\log \log n$ for some sequences.

8 Acknowledgments

Thanks to Gary Miller for his comments and feedback.

References

[BCK02] Avrim Blum, Shuchi Chawla, and Adam Kalai. Static optimality and dynamic search-optimality in lists and trees. *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–8, 2002.

[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.

[CW82] K. Culik, II and D. Wood. A note on some tree similarity measures. *Inform. Process. Lett.*, pages 39–42, 1982.

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

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

[GS78] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *Nineteenth Annual IEEE Symposium on Foundations of Computer Science*, pages 8–12, 1978.

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

[STT86] D. D. Sleator, R. E. Tarjan, and W. P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 122–135, 1986.

[Sun89] R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. *Proceedings of the 13th Symposium on Foundations of Computer Science*, pages 555–559, 1989.

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

[Tar83] Robert Endre Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, 1983.

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

[Wil89] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.