Dynamic Optimality and Multi-Splay Trees[1]

Daniel Dominic Sleator and Chengwen Chris Wang

November 5, 2004
CMU-CS-04-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Dynamic Optimality Conjecture [ST85] states that splay trees are competitive (with 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, *tango* is worse than other BST algorithms by a factor of $\log \log n$.

In this paper we introduce multi-splay trees, which can be viewed as a variant of splay trees. We prove the multi-splay access lemma, which resembles the access lemma for splay trees. With different assignment of weights, this lemma allows us to prove various bounds on the performance of multi-splay trees. Specifically, we prove that multi-splay trees are $O(\log \log n)$-competitive, and amortized $O(\log n)$. This is the first BST data structure to simultaneously achieve these two bounds. In addition, the algorithm is simple enough that we include code for its key parts.

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

Keywords: Dynamic Optimality, Binary Search Tree, Splay Tree, Competitive Algorithm, Amortized Analysis

Dynamic Optimality and Multi-Splay Trees

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

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 of accesses is within a constant factor of any other binary search tree algorithm for processing that sequence of accesses. That is, it states that splay trees are c-competitive [ST85] for some constant c. This problem seems difficult – it has defied concerted attempts to solve it for about 20 years. [1]

Recently Demaine *et al*. [DHIP04] suggested searching for alternative binary search tree algorithms which have small, but non-constant competitive factors. They proposed *tango*, a MST algorithm which achieves dynamic optimality with a competitive ratio of $O(\log\log(n))$. However, for many access patterns, *tango* is worse than standard BST algorithms by a factor of $\log\log n$.

In this paper, we show how to obtain $O(\log\log n)$ competitiveness and preserve $O(\log n)$ amortized performance with multi-splay tree access lemma. Our data structure is not only the first to achieve both of these bounds, but also easy to implement.

1.1 Model

In order to discuss optimality of BST algorithms, we need to give a precise definition for this class of algorithms. The model we use is that implied by Sleator and Tarjan [ST85] and developed in detail by Wilber [Wil89] and Demaine *et al*. [DHIP04] 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're stored in symmetric (left to right) order. Each node has pointers to its left child, its right child, and its parent. Also, each node may keep a constant[2] amount of additional information but no additional pointers.

The BST algorithm is required to process a sequence of access requests $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$. Each access σ_i is a key in the tree[3], and the requested nodes must be accessed in the specified order. Each access starts from the root and follows pointers until the desired node (one with key σ_i) is reached. The algorithm is allowed to update the fields and pointers in any node that the algorithm touches along the way. The cost of the algorithm to satisfy the sequence of requests is defined to be the number of nodes that it touches. To enforce the requirement that the set of keys actually be stored in a binary search tree (rather than some other data structure) we do not allow any information to be preserved from one access to the next, other than that in the nodes themselves, and a pointer to the root of the tree. It's easy to see that this definition is satisfied by any of the standard BST algorithms, such as red-black trees and splay trees.

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

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

[3] 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 way of satisfying these requests there is a cost, as defined above. Thus we can define $OPT(T_0, \sigma)$ to be the minimum cost of any BST algorithm for satisfying these requests. Wilber [Wil89] derived a lower bound on $OPT(T_0, \sigma)$, and this was simplified and renamed the *interleave* bound by Demaine *et al.* [DHIP04]. Let $IB(T_0, \sigma)$ denote the interleave lower bound. This is a sum of bounds, one for each node. Let x be a node of the tree, then we can define $IB(T_0, \sigma, x)$ as follows. First, restrict σ to the set of nodes in the subtree of T_0 rooted at x (including x). Now label each access in this restricted σ either "left" (or "right") depending on if the accessed element is in the left subtree including x (or right subtree) of x. Now $IB(T_0, \sigma, x)$ is the number of times the labels switch.

Theorem 1. *[Wil89] [DHIP04]* $OPT(T_0, \sigma) \geq IB(T_0, \sigma)/2 - O(n) + m$

Culik and Wood [IW82] proved that the number of rotations needed to change any binary tree of n nodes into another one [4] is at most $2n-2$. It follows that $OPT(T, \sigma)$ differs from $OPT(T', \sigma)$ 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 analyzing multi-splay trees.

Each node x in a splay tree can be assigned an arbitrary positive weight $w(x)$. We define the size $s(x)$ of a node to be the sum of the weights of all nodes in the subtree rooted at x. We define the rank $r(x)$ of node x to be $\lg s(x)$. Finally, we define the potential of the tree to be the sum of the ranks of all of its nodes. As a measure of the cost (running time) of a splaying operation, we use the distance from the node being splayed to the root of the tree (unless the root is being splayed, in which case the cost is 1). With these definitions, Sleator and Tarjan prove the following theorem about the amortized cost of splaying.

Theorem 2. *(ACCESS LEMMA) [ST85] The amortized time to splay a node x in a tree rooted at t is at most* $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$.

2 The Multi-Splay Tree Data Structure

We assume that there are exactly $n = 2^k - 1$ nodes. Consider the perfectly balanced binary search tree made up of these n nodes. This tree is called the *reference tree*, and will be denoted by P. The depth of any node in P is at most $\lg(n+1)$. (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, except that the preferred children will change over time, as explained below. We call a chain of preferred children a *preferred path*. The nodes of the reference tree are partitioned into 2^{k-1} 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.

The multi-splay trees data structure is a binary search tree (over the same set of n keys contained in the reference tree) that evolves over time, and preserves a tight relationship to the reference tree. Each edge of the multi-splay trees is either *solid* or *dashed*. We'll call a set of vertices connected by solid edges a *splay tree*. There is a one-to-one correspondence between the splay trees of the multi-splay tree and the

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

preferred paths of the reference tree. The set of nodes in a splay tree is exactly the same as the nodes in its corresponding preferred path. In other words, at any point in time the multi-splay trees can be obtained from the reference tree by viewing each preferred edge as solid, and doing rotations on the solid edges.

It is important to remember that both the reference tree and the multi-splay trees represent the same set of nodes, in the same symmetric order. Thus, the multi-splay trees is, in fact, a valid binary search tree representation of the given set of nodes. We'll use T to denote a multi-splay tree. An example of P and T is shown in Figure 2.

Every node of the multi-splay tree T has several fields in it, which we enumerate here. First of all, there are the usual keys, left, right, and parent pointers. Although the reference tree P is not explicitly represented in T, we do keep several pieces of information related to the reference tree. In each node we keep its depth in P, and its height in P. (The height of a leaf in P is zero, and the depth of the root of P is 1.) Both of these quantities are static. (Note that every node in the same splay tree has a different depth in P.) Another field we store in each node is *mindepth*. This this is the minimum depth of all the nodes in the *splay subtree* rooted there. By *splay subtree* of x we mean all the nodes in the same splay tree as x that have x as an ancestor (which includes x). Similarly we store *treesize*, which is the number of nodes in the splay subtree rooted at this node. To represent the solid and dashed edges, we keep a boolean variable in each node that indicates if the edge from this node to its parent is dashed. We'll call this the *isroot* bit.

3 The Multi-Splaying Algorithm

Our data structure and algorithm are fairly simple, but there are subtle details that must be correct in order for the running time analysis to go through. We've included C++ code for the important parts of the algorithm in the appendix. Here we give a high-level description.

First, we explain the algorithm assuming we have the reference tree P, then we explain how to update 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 bottom up, so that the accessed node x is on the same preferred path as the root of P. In other words, traverse the path from x to the root doing a switch at each non-preferred child on the path. That's the whole algorithm from the point of view of the reference tree. The tricky part is to do it without the reference tree.

Remark. The number of switches that occur in the reference tree due to a single access is exactly the increase in interleave bound due to that access.

Unfortunately, P is not our representation, T is. To achieve $O(\log \log n)$ competitive bound, we can only afford to spend $O(\log \log n)$ amortized time per path. It turns out that we can simulate a switch in P with at most three splay operations, and two changes in isroot bits in T.

More specifically, say we have a node y and we want to change its 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 the subtree of P containing 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 continuous regions, separated by y.

Lets 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. This can be done efficiently because L and R are continuous in the symmetric ordering. So one way to do it using splaying would be to split off the tree containing L by splaying y, then splaying l the leftmost node in L (stopping at the left child of y). This node, l, is the leftmost node deeper (in P) than y. We could find this in T if we had a *maxdepth* field instead mindepth. Adding R is done simply by

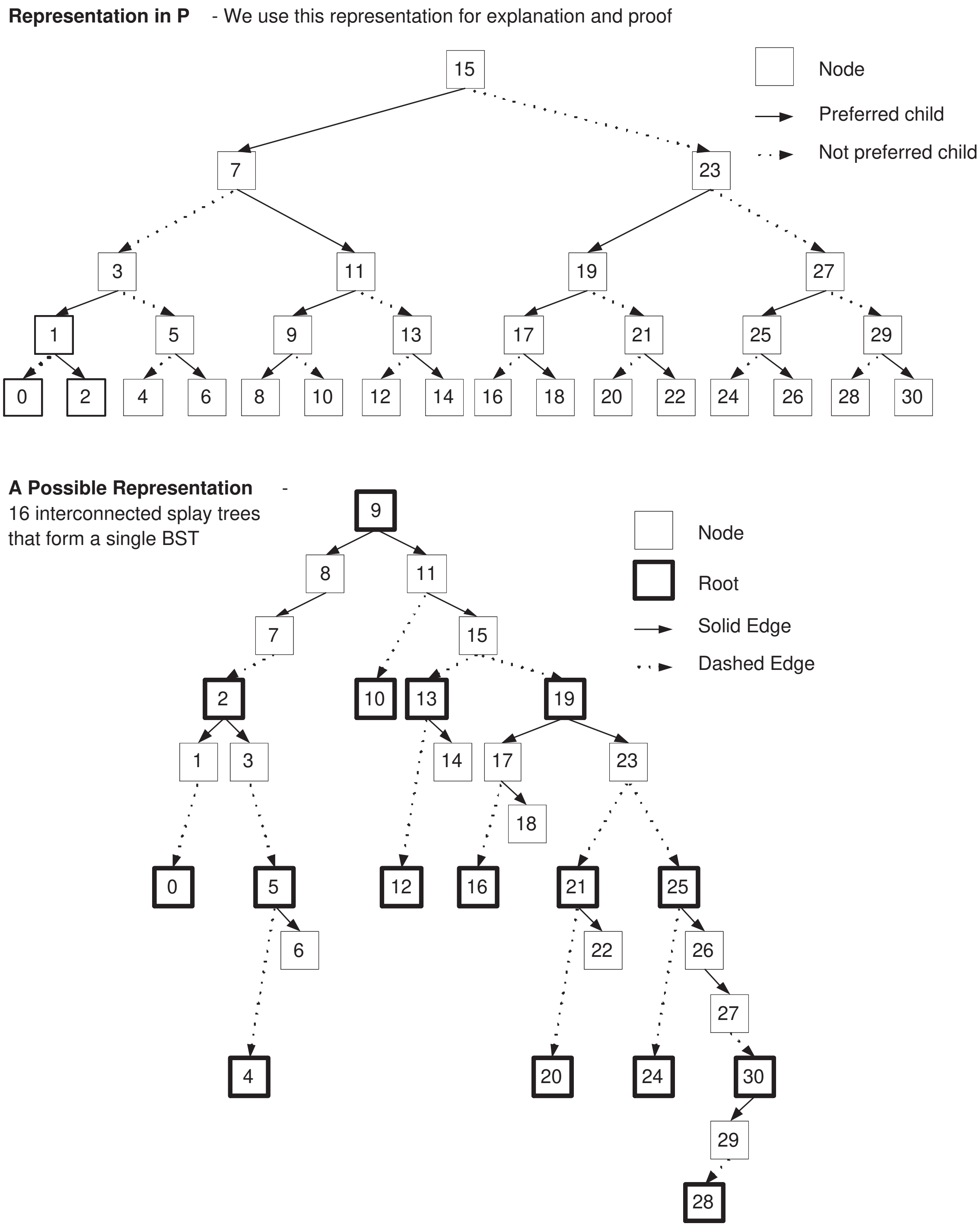

Figure 1: Multi-Splay Data Structure – One can always obtain P from T by a set of rotations on solid edges.

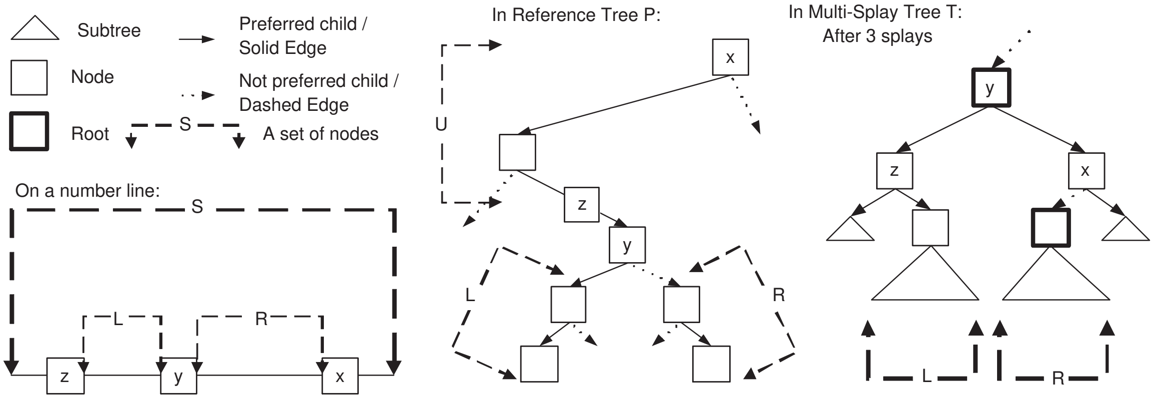

Figure 2: Graphical representations of S, U, L, R, x, y, and z during a single switch.

setting isroot to false for the node that is least common ancestor of the set R in T. This method would then be analogous to the technique used by Demaine *et al.* [DHIP04].

Unfortunately, this technique does not suffice to prove the $O(\log n)$ amortized bound. To obtain the desired bound, we can only afford to splay nodes that are in $\{y\} \cup U$. We first find z the predecessor of L in S, using the mindepth field. Then we splay y and splay z until it become the left child of y. This ensures that the right child of z is the least comment ancestor of L. So we simply set the right child of z as root. This is equivalent to removing L from y's splay tree. As for merging R, we simply splay the successor of y (called x) in U to be the right child of y. This ensures that the left child of x is the least common ancestor of R in T.

But an access is not just a single switch in P, it is a sequences of switches. For the purposes of our running time analysis, we do these from bottom to top. And also we must splay the accessed node after all the switches so that it becomes the root of T.

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. And we've not discussed the boundary cases such as when z or x does not exist. This is where the treesize and height fields are used. The interested reader is invited to study the C++ implementation in the appendix.

4 Running Time Analysis

To define the potential of a multi-splay trees, we assume that each item x has an arbitrary positive *weight w(x)*. We define the *size s(x)* of a node x in the tree to be the sum of the individual weights of all items in the splay subtree rooted at x. (i.e. all items in the subtree of x reachable by traversing only solid edges). We define the *rank r(x)* of node x to be $\lg s(x)$. Finally, we define the potential of the tree to be the sum of the ranks of all its nodes.

Remark. If we view T as a collection of splay trees, then this potential is the one defined by Sleator and Tarjan for a single splay tree, summed over all the splay trees.

Theorem 3. *(Generalized Access Lemma) Given a pointer to a node x, the amortized time to splay a node x with respect to an ancestor a in the same splay tree is at most* $3(r(a) - r(x)) + 1 = O(\lg(s(a)/s(x)))$.

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

Multi-Splay Access Lemma: To further generalize the access lemma to multi-splay trees, we define $p(x,P)$ to be the set of all paths from node x in P to a descendant of x with no children. We also define $d(x,P)$ to be all the descendants of x in P.

Theorem 4. *(Multi-Splay Access Lemma)*

Let P be an initial reference tree with root t, $f \geq 2$ be a multiplier, and $w(x)$ be any positive weight assignment satisfying these two conditions:

$$w(x) \geq \max_{v \in d(x,P)} w(v) \tag{1}$$

$$f * w(x) \geq \max_{t \in p(x,P)} \sum_{v \in t} w(v). \tag{2}$$

Then the running time to access the sequence $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$ is amortized

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

Remark. In this paper we assume that the reference tree is perfectly balanced. However, the multi-splay access lemma is true without the assumption of a balanced reference tree.

Intuitively, the first weight condition forces the shallower nodes in P to have bigger weight. This is necessary because multi-splay trees tends to access the nodes with lower depth significantly more frequently than the nodes with higher depth in P. As for the second weight condition, it forces P to be somewhat balanced to achieve a reasonable upper bound. As for the multiplier f, it significantly relaxes the second constraint on the growth of $w(x)$.

In the proof, we first bound the time for each switch. Then we bound the time for each access as a function of the number of switches. Then we relate the number of switches to the interleave bound.

Proof: Let the set of keys be $S = a_1, a_2, \ldots, a_n$. For any access σ_m, let $\sigma' = \sigma_1, \sigma_2, \ldots, \sigma_{m-1}$ be the access sequence before σ_m, and $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$. Let the k switches made by accessing σ_m be $Y = y_1, y_2, \ldots, y_k$, and t_i be the root of the splay tree containing y_i before the access. Define t_{k+1} to be the root of the splay tree containing σ_m.

For a particular switch y_i, we define $r(v)$ and $s(v)$ to be the rank and size of v before the changes in isroot bit. Similarly, $r'(v)$ and $s'(v)$ be the rank and size of v after the changes in isroot bit. Each switch operation consists of at most 3 splays (on z, y_i, x), setting the isroot bit of cz, and clearing the isroot bit of cx (where x is parent of cx, z is parent of cz). As a result, the root changes affect the potential of nodes z, y_i, x. Specifically, $s(z)$ decreases by $s(cz)$; $s(y_i)$ changes by $s(cx) - s(cz)$; and x increases by $s(cx)$. In addition, from the second condition on the weight assignment, $f * w(y) \geq s(cx)$,

$$\begin{aligned}
\Delta\Phi &= (r'(x) - r(x)) + (r'(y) - r(y_i)) + (r'(z) - r(z)) \\
&< (r'(x) - r(x)) + (r'(y_i) - r(y_i)) \\
&= \lg(s'(x)/s(x)) + \lg(s'(y_i)/s(y_i)) \\
&< \lg((s(x) + s(cx))/s(x)) + \lg((s(y_i) + s(cx))/s(y_i)) \\
&< \lg(1 + s(cx)/w(y_i)) + \lg(1 + s(cx)/w(y_i)) \\
&< 2\lg(1 + f) \\
&= O(\lg f)
\end{aligned}$$

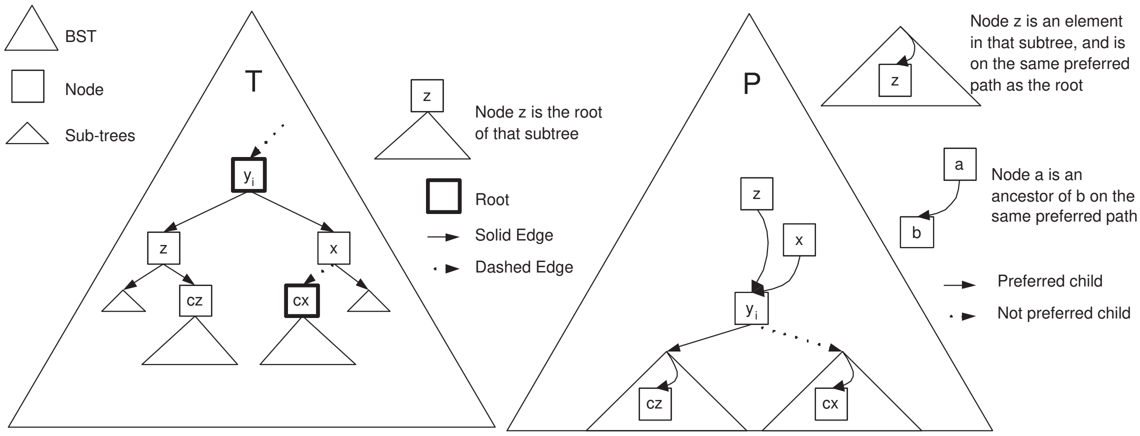

Figure 3: The relationship of x, y_i, z, cx and cz in both multi-splay tree T (left) and reference tree P (right). This is right after the 3 splays, but before setting the isroot bit for cz and clearing the isroot bit for cx. It is important to observe that $s(cx)$ corresponds exactly to the sum of the nodes on a path in $p(y, P)$

Because both x and z are ancestors of y in P, $w(x) \geq w(y)$ and $w(z) \geq w(y)$ by the first condition,

$$\begin{aligned}
Time(switch(y_i)) &= Time(splay(y_i)) + Time(splay(px)) + Time(splay(pz)) + \Delta\Phi \\
&< O(\lg s(t_i)/s(y_i)) + O(\lg s(t_i)/s(px)) + O(\lg s(t_i)/s(pz)) + O(\lg f) \\
&\leq O(\lg s(t_i)/w(y_i)) + O(\lg s(t_i)/w(px)) + O(\lg s(t_i)/w(pz)) + O(\lg f) \\
&\leq O(\lg s(t_i)/w(y_i)) + O(\lg s(t_i)/w(y_i)) + O(\lg s(t_i)/w(y_i)) + O(\lg f) \\
&= O(\lg s(t_i)/w(y_i)) + O(\lg f) \\
&\leq O(\lg s(t_i)/(s(t_{i+1})/f)) + O(\lg f) \\
&= O(\lg s(t_i)/s(t_{i+1})) + O(\lg f)
\end{aligned}$$

When we access a node in a multi-splay tree, we are just making a series of switches and then doing a final splay. Therefore,

$$\begin{aligned}
Time(Access(\sigma_m)) &= \sum_{i=1}^{k}(switch(y_k)) + Time(splay(\sigma_m)) \\
&= \sum_{i=1}^{k} O(\lg s(t_i)/s(t_{i+1})) + \sum_{i=1}^{k} O(\lg f + 1) + O(\lg(s(t_1)/s(\sigma_m))) \\
&= O(\lg(s(t_1)/s(t_{i+1}))) + O(k * (\lg f)) + O(\lg(f * w(t)/w(\sigma_m))) \\
&= O(\lg(w(t)/w(\sigma_m))) + O((k+1) * (\lg f))
\end{aligned}$$

Because a switch occurs when the preferred child changes from left to right (or right to left), this is exactly when the previous access in y_i's subtree in P is in the left subtree (or right subtree) of y, while σ_m is in the right subtree (or left subtree) of y. Thus, $\forall_{0<i\leq k}(IB(T_0, \sigma', y_i) + 1 = IB(T_0, \sigma, y_i))$. In addition, for all other nodes $v \neq y_i$, the $IB(T_0, \sigma', v) = IB(T_0, \sigma, v)$. Hence,

$$k = IB(T_0, \sigma) - IB(T_0, \sigma')$$

Thus, exactly $IB(P, \sigma)$ switches are made for an m element access sequence σ. The amortized running time for the access sequence σ is:

$$
\begin{aligned}
Time(Access(\sigma)) &= \sum_{i=1}^{m} Time(Access(\sigma_i)) \\
&= \sum_{i=1}^{m} O(\lg(w(t)/w(\sigma_i)) + \sum_{i=2}^{m} O((1 + IB(P, \sigma_1 \ldots \sigma_i) - IB(P, \sigma_1 \ldots \sigma_{i-1})) * (\lg f)) \\
&= \sum_{i=1}^{m} O(\lg(w(t)/w(\sigma_i)) + O((IB(P, \sigma) + m) * (\lg f)) \\
&= O(\sum_{i=1}^{m} \lg(w(t)/w(\sigma_i)) + (\lg f) * (IB(P, \sigma) + m)).
\end{aligned}
$$

□

Theorem 5. *Multi-Splaying is $O(\log \log n)$-competitive.*

Proof: Let P be a balanced tree of depth at most $(\lg n + 1)$. Set $w(v) = 1$ for all node v, and choose $f = 2 \lg n$. This assignment satisfies the first weight condition of theorem 4 because the maximum of any set of weight is 1. The assignment also fulfills the second condition because the length of each path is at most $(\lg n + 1) < f$. Applying the multi-splay access lemma, the running time for an access sequence σ is

$$
O(\sum_{i=1}^{m} \log(1/1) + (\log \log n) * (IB(P, \sigma) + m)) = O((\log \log n) * OPT(\sigma)).
$$

□

Theorem 6. *Multi-Splaying is amortized $O(\log n)$.*

Proof: Let P be a balanced tree of depth at most $(\lg n + 1)$. Let $h(v)$ be the height of node v, where the height of leaf is 0. Set $w(v) = 2^h(v)$, and $f = 2$. The first condition is trivially true because all the descendants have a lower weight. The weight assignment also satisfies the second condition because the weight of each path forms a geometric sequence. Using the multi-splay access lemma, our running time is bounded by

$$
O(\sum_{i=1}^{m} \log(n/w(\sigma_i)) + (\log 2) * (IB(P, \sigma) + m)) = O(m \log n + OPT(\sigma)) = O(m \log n).
$$

□

Theorem 7. *For a sequences of m access, multi-Splaying is $O(\min(\lg \lg n * OPT(\sigma), m \log n))$*

Proof: We simply pick a balanced initial tree P. Since the whole algorithm is independent of $w(x)$ and f, we can simply combine the above theorems to obtain the desired bound. □

5 Observations and Open Questions

In this paper we showed that multi-splay trees achieve $O(\log \log n)$ competitiveness, and simultaneously achieve the natural $O(\log n)$ amortized bound for access in a BST. We also proved the access lemma for multi-splay trees – a powerful paramaterizable theorem for analyzing multi-splay tree access sequences.

The multi-splay tree access operation is similar to splaying, but differs in a few important ways. Consider modifying the algorithm so that it did not splay z. 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) on nodes on v's path to the root. The pattern is that starting at an ancestor of v, we splay for a while, then move to an ancestor, then splay for a while, then move to an ancestor, etc. Finally we splay v to the root.

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 a tighter bound on the running time, compared to what we can prove about 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.

Empirical evidence suggests that multi-splay trees satisfy the sequential access lemma. That is, that accessing all the keys in the tree in sorted order takes $O(n)$ time. Is this true? Our experiments indicate that the number of rotations for a sequential access on $n < 2^{23}$ nodes is bounded by $3.8n$.

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 interleave bound is insufficient, because it's known to be off by a factor of $\log \log n$ for some sequences.

Another area for research is to generalize the algorithm and analysis to allow insertions and deletions.

Appendix – Selected Portions of Multi-Splay Code

The implementation below makes the assumption that the reference tree P is a perfectly balanced tree of $2^k - 1$ nodes. We include the key components, but omit repetitious code, initialization, and standard algorithms (like splaying and rotation).

```
struct cNode {
  cNode *parent, *left, *right;
  int key;
  int  mindepth; // minimum depth of all the nodes in its splay subtree
  int  treesize; // number of nodes in its splay subtrees, including itself
  bool isroot;   // is this node the root of its splay tree
  int  refdepth; // the depth in the initial reference tree
  int  refheight;// refdepth + refheight = lg N
};

// initialize the multi-splay trees with a reference tree.
// the reference tree is already a valid multi-splay, we only
// need to compute all the necessary fields.
cNode* Init(cNode* reference);

// nodes with positive and negative infinity(INFTY) key, and refdepth 0
cNode *nodeNegInfty, *nodePosInfty;
```

```
// return the largest node in its splay subtree with refdepth less than depth
cNode* Rightmost(cNode* node, const int depth)
{
  if (node==NULL || node->isroot==true || node->mindepth>=depth) return NULL;
  cNode* right = Rightmost(node->right, depth);
  if (right != NULL) return right;
  if (node->refdepth < depth) return node;
  return Rightmost(node->left, depth);
}

// return the smallest node in its splay subtree with refdepth less than depth
// this is analogous to Rightmost()
cNode* Leftmost(cNode* node, const int depth);

// splay the node to the root using the standard splaying algorithm
// any version (top down, bottom up, etc) of splay will work here.
// we implemented the bottom up version.
cNode* Splay(cNode* node);

// splay the node to this ancestor.  node ends up being a child of ancestor.
cNode* Splay(cNode* node, cNode* ancestor);

// update mindepth and treesize fields for a specific node using its
// left and right children
void Recompute(cNode* node);

// switch y's prefer child from left to right
void Left2Right(cNode* y, cNode* upper){
  Splay(y);
  cNode* z;
  if (y->left->treesize == y->refheight) z = NULL;
  else z = Rightmost(y->left, y->refdepth);
  if (z != NULL) {
    Splay(z, y->left);
    z->right->isroot = true;
    Recompute(z);
  }
  else y->left->isroot = true;
  if (upper->key != INFTY) {
    Splay(upper, y->right);
    upper->left->isroot = false;
    Recompute(upper);
  }
  else y->right->isroot = false;
  Recompute(y);
}
```

```
// switch y's prefer child from right to left
// this is analogous to Left2Right code
void Right2Left(cNode* y, cNode* lower);

// return the pointer to the root of multi-splay tree
cNode* GetRoot();

// recursive helper function for Access
cNode* Query(int key, cNode* cp, cNode *lower, cNode* upper){
  if (cp->isroot) {lower = nodeNegInfty; upper = nodePosInfty;}
  if (cp->key == key) return cp;
  if (key > cp->key)      {lower = cp; cp = cp->right;}
  else if (key < cp->key) {upper = cp; cp = cp->left;}
  cNode* result = Query(key, cp, lower, upper);
  if (cp->isroot){
    if (lower->refdepth > upper->refdepth) Left2Right(lower, upper);
    if (upper->refdepth > lower->refdepth) Right2Left(upper, lower);
  }
  return result;
}

// access the node with key value
cNode* Access(const int key){
  cNode* result = Query(key, GetRoot(), NULL, NULL);
  return Splay(result);
}
```

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.

[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.

[IW82] K. Culik II and D. Wood. A note on some tree similarity measures. *Inform. Process. Lett.*, pages 39–42, 1982.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 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, 1992.

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

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