Multi-Splay Trees

Chengwen Chris Wang

CMU-CS-06-140
July 31, 2006

School of Computer Science
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Daniel Sleator, Chair
Manuel Blum
Gary Miller
Robert Tarjan, Princeton University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

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

Keywords: Binary Search Tree, Competitive Algorithm, Dynamic Finger, Deque, Dynamic Optimality, Splay Tree, Tango

Abstract

In this thesis, we introduce a new binary search tree data structure called multi-splay tree and prove that multi-splay trees have most of the useful properties different binary search trees (BSTs) have. First, we demonstrate a close variant of the splay tree access lemma [ST85] for multi-splay trees, a lemma that implies multi-splay trees have the $O(\log n)$ runtime property, the static finger property, and the static optimality property. Then, we extend the access lemma by showing the remassing lemma, which is similar to the reweighting lemma for splay trees [Geo04]. The remassing lemma shows that multi-splay trees satisfy the working set property and key-independent optimality, and multi-splay trees are competitive to parametrically balanced trees, as defined in [Geo04]. Furthermore, we also prove that multi-splay trees achieve the $O(\log \log n)$-competitiveness and that sequential access in multi-splay trees costs $O(n)$.

Then we naturally extend the static model to allow insertions and deletions and show how to carry out these operations in multi-splay trees to achieve $O(\log \log n)$-competitiveness, a result no other BST scheme has been proved to have. In addition, we prove that multi-splay trees satisfy the deque property, which is still an open problem for splay trees since it was conjectured in 1985 [Tar85]. While it is easy to construct a BST that satisfies the deque property trivially, no other BST scheme satisfying other useful properties has been proved to have deque property. In summary, these results show that multi-splay trees have most of the important properties satisfied by different binary search trees.

Acknowledgments

I am pleased to acknowledge the people who made this research possible. First and foremost, I am deeply indebted to Professor Daniel Sleator for his unwavering support and valuable suggestions. I am grateful to Professor Gary Miller for many useful suggestions and helpful discussions. I owe a debt of gratitude to Jonathan Derryberry for extensive discussions that stimulated my thinking and helped to clarify many of the ideas presented. Lastly, many thanks to Maverick Woo for his helpful suggestions and constructive criticisms.

Contents

1 Introduction **1**

1.1 Related Works 2

1.1.1 Splay Trees 2

1.1.2 Tango 5

1.1.3 Chain Splay 5

2 Binary Search Trees (BSTs) **7**

2.1 BST Model 7

2.2 Competitive Analysis on BST 8

2.3 Interleave Lower Bound 8

2.4 Properties of an $O(1)$-competitive BST 9

2.4.1 Implications between the Properties 13

3 Multi-Splay Trees **17**

3.1 The Multi-Splay Tree Data Structure 17

3.1.1 Simplified Drawing of a Multi-Splay Tree 18

3.1.2 Recursive Definition of a Multi-Splay Tree 20

3.2 The Multi-Splaying Algorithm 20

3.3 Details of Multi-Splaying Algorithm 24

3.3.1 Determining the Direction of a Switch 24

3.3.2 Switch on a Node with Missing Children 24

3.3.3 Switch without *refLeftParent* or *refRightParent* 25
3.4 Running Time Analysis 26
3.4.1 Multi-Splay Tree Satisfies $O(\log \log n)$-Competitive Property . . 26
3.4.2 Multi-Splay Tree Satisfies Access Property 28
3.4.3 Multi-Splay Tree Satisfies Reweight Property 35
3.4.4 Multi-Splay Tree Satisfies Scanning Property 37
3.5 Comment on the Fields of the Multi-Splay Tree Nodes 40

4 Dynamic Binary Search Trees **43**
4.1 Dynamic BST Model 43
4.2 Competitive Analysis on Dyanmic BST 44
4.3 Dynamic Interleave Lower Bound 44
4.3.1 Proof of the Dynamic Interleave Bound 45
4.4 Properties of an $O(1)$-dynamic-competitive BST 48

5 Dynamic Multi-Splay Trees **51**
5.1 Making Multi-Splay Tree Dynamic 51
5.2 Simulating Pointer Traversal in the Reference Tree – Virtual Traversal . . 52
5.2.1 Locating Child in the Reference Tree 52
5.2.2 Locating Parent in the Reference Tree 53
5.3 Simulating Rotations in the Reference Tree – Virtual Rotation 54
5.4 Implementing Insertion 56
5.5 Implementing Deletion 58
5.6 Running Time Analysis 60
5.6.1 Proof of $O(\log n)$ amortized time per operation 60
5.6.2 Proof of $O(\log \log n)$-dynamic-competitiveness 61
5.6.3 Proof of Deque Theorem 62

6 Conclusion **67**
6.1 Comparisons between Multi-Splay Tree and Splay Tree 68

6.2 Lower Bounds 68
6.3 More Open Problems 69

A Lists of Notations and Symbols **71**

B Table of Constants **75**

Bibliography **77**

List of Figures

2.1 Construction and example of a static BST for a fix finger. 14
2.2 Implication relationships of BST properties. 16

3.1 The fields of a node in a multi-splay tree. 18
3.2 A multi-splay tree with fields and its simplified version. 19
3.3 An example of a multi-splay tree and its corresponding reference tree. . . 21
3.4 Graphical representations of a single left-to-right switch. 23
3.5 Graphical representations of a single left-to-right switch with missing children. 25
3.6 Graphical representations of a single left-to-right switch with missing parent. 27
3.7 Notations in reference tree 30

5.1 Preprocessing of a rotation in the reference tree. 54
5.2 A rotation in the reference tree. 55
5.3 After querying x or z during an insertion. 57
5.4 An example of a multi-splay tree during deletion. 59

List of Tables

A.1 The symbols that are used throughout the thesis. 71
A.2 The notation used throughout the thesis. 72
A.3 The terminology used throughout the thesis. 73
A.4 The notation used in Section 3.4.2 and 3.4.3. 73
A.5 The notation used in Section 3.4.4. 74
A.6 The notation used for Section 4.3. 74

B.1 Table of constants. 75

Chapter 1

Introduction

Efficiently maintaining and manipulating totally ordered sets 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). Various types of BSTs were independently developed by a number of researchers in the early 1960s [Knu73]. Over the years, many types BSTs achieved the theoretical minimum number of $O(\log n)$ key comparisons needed per operation. Hence, many BST algorithms are optimal (up to a constant factor) using worst-case analysis.

However, for many sequences σ of m operations, the optimal cost for executing the sequence is $o(m \log n)$, lower than the theoretical minimum that uses worst-case, per-operation bounds. To exploit the patterns of query sequences from specific applications, such as randomly and independently drawn queries from a fixed distribution[1], finger search[2], and sequential queries[3], researchers have designed specialized BST algorithms that efficiently support various types of access patterns.

In 1985, Sleator and Tarjan [ST85, Tar85] showed that it is possible to efficiently handle all of the query patterns mentioned above (and many more) in a single BST data structure called a *splay tree*. A splay tree is a self-adjusting form of BST such that each time a node in the tree is accessed, that node is moved to the root according to an algorithm called *splaying*. Splay trees have a number of remarkable properties including the Balance Theo-

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

[2]See [BY76, GMPR77, Tsa86, TvW88, HL79, Har80, HM82, Fle93, SA96, BLM$^+$03, Pug89, Pug90, Iac01b, Bro05]

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

rem [ST85], the Static Optimality Theorem [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], the Key Independence properties [Iac02], and competitiveness to parametrically balanced trees [Geo04]. Because splay trees satisfied so many such properties, they were conjectured to be *dynamically optimal* by Sleator and Tarjan [ST85], meaning that splay trees were conjectured to be $O(1)$-competitive to the optimal off-line BST. After more than 20 years, the Dynamic Optimality Conjecture remains an open problem.

Since no one has shown that any 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 data structure proved to achieve a nontrivial competitive factor. Unfortunately, tango does not satisfy many of the necessary conditions of a constant competitive BST, including some that splay trees are known to satisfy. For example, it does not satisfy the Scanning Theorem.

In this thesis, we introduce a new data structure called multi-splay trees. We prove that multi-splay trees can efficiently execute most (if not all) query sequences proven to execute efficiently on other binary search trees. In Chapter 2, we define the static BST model and explain competitive analysis on BST. Then we describe a lower bound and enumerate many of the necessary properties of a constant competitive BST. In Chapter 3, we proceed to prove that multi-splay trees have almost all of the properties mentioned in Chapter 2. In Chapter 4, we generalize the BST model to support insertions and deletions. Since this is a new model, we prove a new lower bound and describe a few desirable properties in our new model. In Chapter 5, we prove that multi-splay trees are $O(\log \log n)$ dynamic competitive and satisfy deque property. No other dynamic binary search trees are proven to be $O(\log \log n)$ competitive. Moreover, the deque property is a long-standing unproven conjecture for splay trees.

1.1 Related Works

1.1.1 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.

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

Theorem 2 (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 $c_s * \lg(s(a)/s(v)) + c_{sa}$.*

The main difference between this and the original access lemma is that we are allowed to stop at any ancestor a. In other words, splay v in the subtree rooted at 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.

Theorem 3 (Reweighting Lemma)**.** *[Geo04] For any sequence of interleaving splays and reweights, the amortized time to splay a node v in a tree currently rooted at r is at most $c_s * \lg(s(r)/w(v)) + c_{sa}$, and [4] the amortized time to reweight a node v from $w(v)$ to $w'(v)$ is $max(0, c_r * \log(w'(v)/w(v)))$, where $c_r = c_s + 1 = 4$.*

While this theorem is proved in [Geo04], the theorem is not very well-known. For completeness, here is an informal proof.

Proof. Consider an extended version of the Splay Tree Access Lemma, modified as follows. Using the same potential function as used in the access lemma proof, change the amortized cost of a splay to $c_s * \lg(s(r)/w(v)) + c_{sa}$ (changing the denominator inside the log from $s(v)$ to $w(v)$). Because $s(v) \geq w(v)$ for all v, the Splay Tree Access Lemma shows this expression is a valid upper-bound on the amortized cost of a splay.

Additionally, allow two operations, DecW$(w(x), w'(x))$ and IncRootW$(w(r), w'(r))$. The operation DecW$(w(x), w'(x))$, which decreases the weight of node x from $w(x)$ to $w'(x)$ can only decrease the potential, so its amortized cost is at most 0. The operation

[4] The denominator inside the log is $w(v)$, which is different from the access lemma.

IncRootW$(w(r), w'(r))$, which increases the weight of the root r from $w(r)$ to $w'(r)$ only changes the potential of the root, which it increases. To account for this increase in potential, it suffices to pay the following amortized cost:

$$\begin{aligned}\text{Change in potential} &= \log s'(r) - \log s(r)\\ &= \log \frac{s(r) + w'(r) - w(r)}{s(r)}\\ &\leq \log \frac{w'(r)}{w(r)}.\end{aligned}$$

Note that reweighting an element does not change the actual cost, and the amortized cost derived from the extended access lemma is an upper bound on the actual cost. Thus, if the total cost computed using the reweighting lemma is at least as much as the total cost computed using the extended access lemma, then the total cost from the reweighting lemma is an upper bound on the actual cost. Hence, we only need to assign a cost to each of the reweight operations so that the cost according to the reweighting lemma is always at least the cost according to the extended access lemma.

To match the cost of reweighting lemma to that of the extended access lemma, we use the same potential function as the access lemma, and use the same splay cost and decrease weight cost bounds as used in the extended access lemma. That is, Splay(x) costs $c_s * \lg(s(r)/w(v)) + c_{sa}$, and DecW$(w(x), w'(x))$ costs 0.

As for the IncW$(w(x), w'(x))$ operation, (which increases the weight of x to $w'(x)$), the reweighting lemma will reweight x immediately instead of waiting until x is splayed to the root, which the extended access lemma has to do since the weight increase can only be performed at the root. Note that since decreases in weight always occur immediately, while increases in weight are delayed in the extended access lemma, $s(r)$ in the extended access lemma is always less than or equal to $s(r)$ in the reweighting lemma. To avoid confusion, we use $s(r)$ to denote the total weight in extended access lemma, and $s(r)^+$ to denote the total weight in the reweighting lemma.

Moreover, for a particular node v, there is no reason to increase the weight of v unless we are about to splay it because if we are not planning to splay v, increasing its weight only increases the total weight without making any other operations cheaper. Once we splay v, its weight will be the same in both the extended access lemma and the reweighting lemma. Thus, for a fixed node v, the reweighting lemma pays less than the extended access lemma pays *only* on the *first* splay of v after v's weight increases. Thus, we can figure out the cost of IncW$(w(v), w'(v))$ so reweight always pay more than the extended access lemma as follows:

$$
\begin{aligned}
\text{(Amortized cost of extended access lemma)} &\leq \text{(Total cost of reweighting lemma)}\\
\text{Splay}_{access}(v) + \text{IncRootW}(w(v), w'(v)) &\leq \text{IncW}(w(v), w'(v)) + \text{Splay}_{reweight}(v)\\
c_s \lg \frac{s(r)}{w(v)} + c_{sa} + \lg \frac{w'(v)}{w(v)} &\leq \text{IncW}(w(v), w'(v)) + c_s \lg \frac{s(r)^+}{w'(v)} + c_{sa}\\
c_s \lg \frac{s(r) * w'(v)}{w(v) * s(r)^+} + \lg \frac{w'(v)}{w(v)} &\leq \text{IncW}(w(v), w'(v))\\
(c_s + 1) * \lg \frac{w'(v)}{w(v)} &\leq \text{IncW}(w(v), w'(v)).
\end{aligned}
$$

Thus, if we assign a cost of $(c_s + 1) * \lg(w'(v)/w(v))$ to IncW$(w(v), w'(v))$, the total cost computed using the reweighting lemma will always be at least the cost using the extended access lemma and, hence, it is at least the cost computed using the original Splay Tree Access Lemma. □

1.1.2 Tango

Proposed by Demain *et al*, Tango was the first $O(\log \log n)$-competitive BST. Currently, the best and the only non-trivial competitive factor for BST is $O(\log \log n)$. Unfortunately, tango does not satisfy many of the necessary conditions of a constant competitive BST, including some that splay trees are known to satisfy. Inspired by tango, we invented multi-splay trees in attempt to alleviate some of the theoretical shortcomings of tango while maintaining its $O(\log \log n)$-competitiveness property. A multi-splay tree is essentially the same as tango, except the multi-splay tree is a collection of splay trees while tango is a collection of red-black trees. Another minor difference is that tango searches for different nodes during a simulated switch of the reference tree. This differences are elucidated in the description of multi-splaying algorithm in Section 3.2.

1.1.3 Chain Splay

Based on tango, Georgakopoulos [Geo05] modified the splay algorithm [ST85] to achieve $O(\log \log n)$-competitiveness and $O(\log n)$ amortized running time. His algorithm, called chain splaying, exhibits none of the other necessary conditions of a constant competitive BST. Although it is quite similar to multi-splay tree (without insertion and deletion), Georgakopoulos independently discovered his data structure. These two data structures only differs on the locations of the partial splays during a series of the switches. Because of those differences, Georgakopoulos managed to use $\lg \lg n$ less bits per node. Unfortunately, these small differences make it so that most of the techniques we developed to analyze multi-splay tree do not apply to chain splaying. The effect of these small differences is discussed in Section 3.5.

Chapter 2

Binary Search Trees (BSTs)

2.1 BST Model

In order to discuss the optimality of BST algorithms, we need to give a precise definition of this class of algorithms and their costs. The model we use is 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 $o(\log n)$ bits of additional information but no additional pointers.

A BST algorithm is required to process a sequence of queries $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$. Each access σ_i is a query to a key $\hat{\sigma_i}$ in the tree[1], 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 in any node or rotate any edges that it touches along the way[2]. The cost of the algorithm to execute a query sequence is defined to be the number of nodes touched plus the number of rotations.

Finally, we do not allow any information to be preserved from one access to the next, other than 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.

[1]WLOG, this model is only concerned with successful searches [AW98].
[2]A definition of rotation can be found in [CSRL01]

2.2 Competitive Analysis on BST

Given any initial tree T_0 and any m-element access sequence σ, for any BST algorithm satisfying these requests, the cost can be defined using the model in Section 2.1. Thus, we can define OPT(T_0, σ) to be the minimum cost of any BST algorithm for satisfying these requests starting with initial tree T_0. Furthermore, since the number of rotations needed to change any binary search tree of n nodes into another one is at most $O(n)$ [Cra72, CW82, STT86, Mäk88, LP89], it follows that OPT(T_0, σ) differs from OPT(T_0', σ) by at most $O(n)$. Thus, as long as $m = \Omega(n)$, the initial tree is irrelevant. We denote the off-line optimal cost starting from the *best* possible initial tree as OPT(σ). Similarly, for any on-line binary search algorithm A, A(σ) denotes the on-line cost to execute σ starting from the *worst* initial tree. Because the initial tree of a BST algorithm could be a very unbalanced binary search tree, we assume the number of operations, m, is greater than $n \log n$ to avoid unfairly penalizing the on-line BST algorithm.

An on-line binary search tree algorithm A is T-competitive if

$$\forall \sigma \text{A}(\sigma) < T * \text{OPT}(\sigma) + O(m).$$

This framework in which the $O(\log \log n)$-competitive bounds for the best competitive on-line binary search trees [DHIP04, SW04, Geo05, WDS06] are proven does not allow for insertions or deletions. We generalize this framework to include these update operations, and extend the lower bound appropriately in Chapter 4. We also show how to modify the multi-splay tree data structure to handle insertions and deletions, and prove that it remains $O(\log \log n)$-competitive.

2.3 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. Wilber [Wil89] derived a lower bound on OPT(T_0, σ) which was later modified and name the *interleave bound* by Demaine *et al*. [DHIP04].

Let IB(P, σ) 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 IB$(P, \sigma) = \sum_{v \in P}$ IB(P, σ, v), where for each node v, IB(P, σ, 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, IB(P, σ, v) is the number of times the labels switch.

Theorem 4 (Interleave Lower Bound)**.** *[Wil89, DHIP04, DSW05]*

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

Since the number of rotations needed to change any binary tree of n nodes into another one is at most $2n - 6$ [Cra72, CW82, STT86, Mäk88, LP89]. It follows that OPT(T_0, σ) differs from OPT(T_0', σ) by at most $2n - 6$. Thus, as long as $m = \Omega(n)$, the initial tree is irrelevant.

Using the Interleave Lower Bound, the smallest competitive ratio proved for on-line binary search trees [DHIP04, SW04, Geo05, WDS06] are $O(\log \log n)$-competitive. Currently, we still do not know if it is possible to have a smaller competitive ratio. In particular, we do not know if it is possible to have an $O(1)$-competitive BST, but we know an extensive list of properties that any $O(1)$-competitive BST must satisfy.

2.4 Properties of an $O(1)$-competitive BST

Before we move on to discuss the properties identified as necessary for an $O(1)$-competitive BST, let us first discuss the assumptions of this section. In this section, all sequences of operations are assumed to involve only queries. We call a sequence without insertions and deletions a *query sequence*. Since the set of keys do not change, we can assume WLOG that there are n keys numbered from $1, 2, \ldots, n$.

Now we are ready for a complete list of the useful binary search tree properties.

Property. A binary search tree structure has the $O(\log n)$ *runtime* property if it executes every σ in time $O(m \log n)$

In the worst case, some query sequences will need $\Omega(m \log n)$ time [Wil89]. Thus, having this property implies the data structure is theoretically optimal under worst-case analysis. Almost every binary search tree has the $O(\log n)$ runtime property.

Property. [ST85] A binary search tree structure has the *static finger* property if it executes every σ in time $O(m + \sum_{i=1}^{m} \log(|f - \sigma_i| + 1))$ for every integer $1 \leq f \leq n$, where f is called a finger.

There exists a specialized data structure [GMPR77, Bro98] which is tuned for a specific value of f, and has this property for that specific finger. However, for a data structure to have the static finger property, it must have the finger search running time for all possible fingers f.

Property. [AW98] A binary search tree structure A is $O(1)$-*distribution-competitive* if for all n, all distributions D on n elements and all initial trees T_0, the expected cost for A to serve a request is less than a constant times the optimal static tree for distribution D.

An example of a binary search tree that satisfies *only* the $O(1)$-distribution-competitive property is the *move-to-root* binary search tree [AM78]. This binary search tree always rotates the queried node x repeatedly until x become the root. Because the optimal static tree for a fixed distribution D is a static tree, the $O(1)$-distribution-competitive property is implied by the *static optimality* property described below.

Property. [ST85] A binary search tree structure has the *static optimality* property if the time to execute σ is $O(m+\sum_{i=1}^{m} f(i)\log(m/f(i)))$, where $f(i)$ is the number of times key i is queried.

Because $\Omega(\sum_{i=1}^{m} f(i)\log(m/f(i)))$ [Abr63] is an informational theoretical lower bound on a static BST for a sequence of queries with frequency $f(i)$, the binary search trees with static optimality is constant competitive to any static binary search tree, including the optimal static tree for distribution D. Several data structures [Knu71, Fre75, Meh75, Meh79, GW77, HT71, HKT79, Unt79, Hu82, Kor81, KV81, BST85] have the static optimality, but they need to know $f(i)$ during initialization. On the other hand, splay trees have the static optimality property without knowing the frequency $f(i)$ in advance. Any data structure with the static optimality property also has the static finger property [Iac01a].

Property. [ST85] A binary search tree structure has the *working set* property if the time to execute σ is $O(m + \sum_{i=1}^{m} \log d(l(i), i))$, where $d(i, j)$ is the number of distinct keys accessed in the subsequence $\sigma_i, \sigma_{i+1} \ldots \sigma_j$, and $l(i)$ is the index of the last access to σ_i in the subsequence $\sigma_1, \sigma_2, \ldots \sigma_{i-1}$. ($l(i) = 1$ if σ_i does not appear in the subsequence.)

The working set property implies both static finger and static optimality. It also implies that if all queries are in a small subset of keys of size k, then the query sequence can be executed in $O(m \log k)$. In many applications, such as compression [Jon88, GRVW95], a recently queried element is likely to be queried again. These recent queries are exactly the element with low amortized cost in the working set property.

Property. [Iac02] A binary search tree structure has the *key-independent* property if the time to execute σ is $O(E[OPT(b(\sigma))])$, where b is random bijection of the keys from n to n.

Iacono [Iac02] introduced the key-independent optimality as another necessary condition for an $O(1)$-competitive binary search tree, and he proved that the key independent property is equivalent to the working set property up to a multiplicative constant factor.

Property. [ST85] A binary search tree structure has the *access* property if for any positive weight assignment $w(x)$ for each element, the time to execute σ is $O(m + \sum_{i=1}^{m} \log \frac{W}{s(\sigma_i)})$, where $W = \sum_{i=1}^{n} w(i)$, $s(\sigma_i)$ must be greater than $w(\sigma_i)$ (and $s(x)$ can depend on the structure of the binary search tree).

With a simple weight assignment, this property implies the static finger and the static optimality properties [ST85]. Because of the flexibility in assigning weights, the access property can be used to combine and generalize properties proved with weight assignment. For instance, the access property implies that for any constant number of finger $f_1, f_2 ... f_k$, the amortized cost to execute a sequence is summation of the log of the distance to the closest finger. That is, $O(m + \sum_{i=1}^{m} \min_j \log(|f_j - a_i| + 1))$.

Property. [Geo04] Let $w_i(x)$ be any positive weight assignment of x right before i^{th} query. A binary search tree structure has the *reweight* property if the time to execute σ is $O(m + \sum_{i=1}^{m} \log \frac{W_i}{w_i(\sigma_i)} + \sum_{i=2}^{m} \sum_{j=1}^{n} \log \max(0, \frac{w_i(j)}{w_{i-1}(j)}))$, where $W_i = \sum_{j=1}^{n} w_i(j)$.

This property is almost the same as the access property with an additional reweight operation, and the cost to increase the weight of element from *old* to *new* is roughly $O(\log \frac{new}{old})$. The reweight operation is not an operation in the data structure, it is merely used in the analysis. The reweight operation enables the analysis to adapt to the query patterns and prove tighter bounds [Geo04].

Property. A binary search tree structure has the *dynamic finger* property if the time to execute σ is $O(m + \sum_{i=2}^{m} \log(d(i) + 1))$, where $d(i)$ is the difference in rank between the ith query and the $(i-1)$th query.

Brodal [Bro05] wrote a chapter on finger search trees and some of the common data structures with the dynamic finger property. Several search trees [BY76, GMPR77, Tsa86, TvW88, HL79, Har80, HM82, Fle93, SA96, BLM$^+$03] have this property, but many violate the definition of Binary Search Tree. For instance, the level linked (2, 4)-tree of Huddleston and Mehlhorn [HM82] and unified data structure of Iacono [Iac01b, BD04] use extra pointers that are not valid in the BST model; randomized skip lists of Pugh [Pug89, Pug90] duplicates the same key multiple times, which is a violation of the BST model; or the auxiliary *hand* data structure of Blelloch, Maggs and Woo [BMW03] maintains extra pointers into a degree balanced binary search tree. Splay tree [ST85] is one of the few data structure that satisfies the binary search tree model and has the dynamic finger property [CMSS00, Col00].

Property. [Tar85, ST85] A binary search tree structure has the *scanning* property if the time to execute $\sigma = 1, 2, 3, \ldots, n$ is $O(n)$ starting at any valid initial tree.

Property. [ST85] A binary search tree data structure has the *traversal* property if given any initial tree T_0 and a input tree T_i, the cost of sequentially querying elements in the order they appear in preorder of T_i is $O(n)$.

When T_i is a right path, the elements in preorder of T_i is $1, 2, \ldots n$, which is exactly the query sequence in the scanning property. Thus, any data structure that satisfies this property also satisfies the scanning property. While any $O(1)$-competitive binary search tree must have the traversal property, no binary search tree is known to have the traversal property. However, special case of the traversal property (when $T_0 = T_i$ [CH93], or when T_i is a right path [Sun89a, Tar85, Sun92, Elm04]) was proved for splay trees.

Property. A binary search tree data structure has the $O(\log \log n)$*-competitive* property if it executes σ in time $O(\log \log n) * OPT(\sigma)$.

The $O(\log \log n)$-competitive property is currently the best competitive (and only nontrivial) bound proved for a binary search tree [DHIP04, SW04, Geo05, WDS06].

Property. A binary search tree structure is *competitive to parametrically balanced trees* if the data structure is $O(1)$-competitive to parametrically balanced trees.

Parametrically balanced trees is a large class of balance search trees introduced by Georgakopoulos [Geo04]. The class includes most balanced trees, such as BB(α)-trees [NR73, BM80], AVL-trees [AVL62], half-balanced trees [Oli82, Ove83], B-tree [RB72]. These parametrically balanced trees are allowed to restructure based on future queries and pay a small cost proportional to the number of local changes in the structure. Since Georgakopoulos [Geo04] has a detailed description on this class of balanced trees, we omit the details here.

2.4.1 Implications between the Properties

In this section, we show or cite the proof for each implication. All the implications are shown in Figure 2.2.

Lemma 1. *[ST85] If a BST satisfies the access property, then it also satisfies the static optimality property.*

Lemma 2. *[ST85] If a BST satisfies the access property, then it also satisfies the static finger property.*

Lemma 3. *[Iac02] A BST satisfies the working set property if and only if it also satisfies the key-independent property.*

Lemma 4. *[Geo04] If a BST satisfies the reweight property, then is also satisfies the working set property.*

Lemma 5. *[Geo04] If a BST satisfies the reweight property, then it is competitive to parametrically balanced trees.*

Lemma 6. *[Iac00] If a BST satisfies the working set property, then it also satisfies the static optimality property.*

Lemma 7. *If a BST satisfies the static optimality property, then it also satisfies the $O(1)$-distribution-competitive property.*

Proof. Since the optimal BST for a fix distribution is defined as a static tree, and a statically optimal BST is $O(1)$-competitive to every static tree, a BST with static optimality property also satisfies $O(1)$-distribution-competitive property. □

Lemma 8. *If a BST satisfies the static optimality property, then it also satisfies the static finger property.*

Lemma 9. *If a BST satisfies the static finger property, then it also satisfies the $O(\log n)$ runtime property.*

Proof. This is trivially true because the distance of a node to a finger can be at most n. □

Proof. As shown in Figure 2.1, for every fix finger f, we can create a static tree T whose left spine and right spine are $f, f-1, f-2, f-4, \ldots, f-2^j$ and $f, f+1, f+2, f+4, \ldots, f+2^k$, respectively, where $j = \max\{i | f - 2^i \geq 1\}$ and $k = \min\{i | f + 2^i \leq n\}$. Then we construct a balanced tree for each set of nodes hanging off the nodes on the

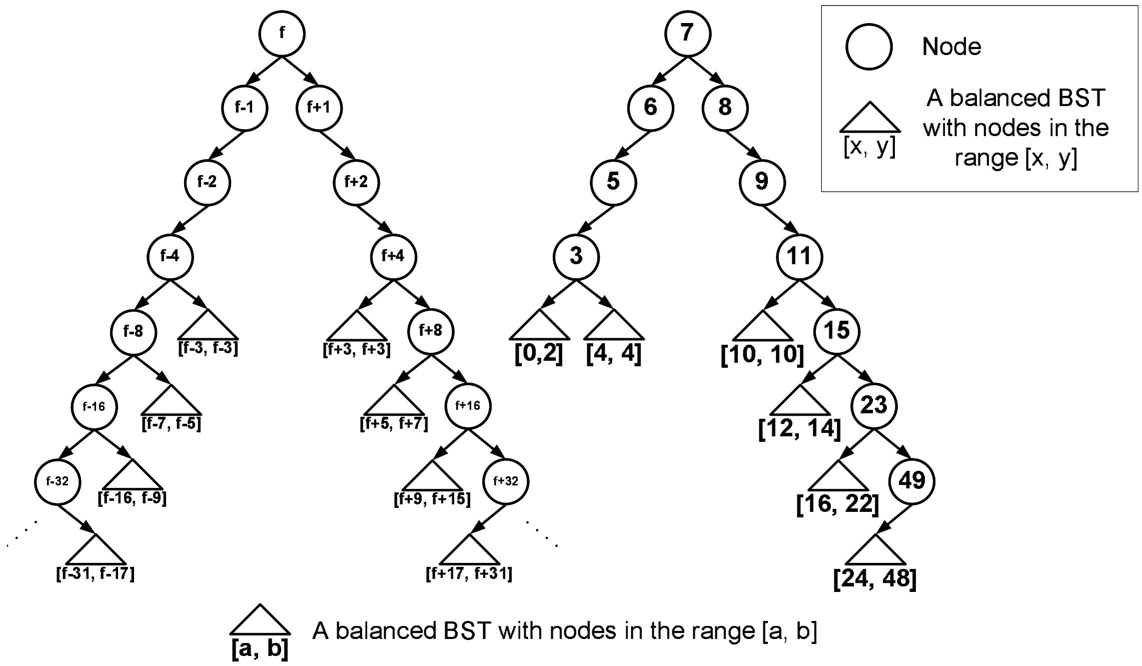

Figure 2.1: The construction of a static BST for a fix finger f is shown on the left. An example of a 50 nodes BST with finger at 7 is shown on the right.

left and right spines. Base on our construction, the depth of a node with value v has $O(\log|f-v|)$ depth. Since we can construct a static tree for each finger, and statically optimal tree is $O(1)$-competitive to all static tree, a BST with static optimality property also satisfies static finger property.

□

Lemma 10. *[Geo04] If a BST is competitive to parametrically balanced trees, then it also satisfies the static finger property.*

Proof. For every fix finger f, the construction shown in Figure 2.1 is a parametrically balanced tree.[3] Thus, the same argument in Lemma 8 applies. □

Lemma 11 (private conversation with Jonathan Derryberry and Marverick Woo)**.** *If a BST satisfies the dynamic finger property, then it also satisfies the static finger property.*

[3]For those who are familiar with [Geo04], every static tree is a parametrically balanced tree because we can set $b(x) = 1/3^{d(x)}$. Moreover, if the static tree has $O(\log n)$ height, then the difference between initial and final potential is bounded by $O(n \log n)$.

Proof. When the last the last query is x, and the next query is y, it suffices to show that if the amortized dynamic finger cost to query y is $c\lg(|x-y|+1)$, then the amortized cost is also at most $2c\lg(|f-x|+1)$ for any finger. Let the potential function be $c\lg(|f-x|+1)$, then the amortized dynamic finger cost is,

(log of the distance to the last query) + (initial potential) - (final potential).

For any query, the last query and the next query is either on the same side of the finger, or difference sides (case 1). When the queries are on the same side, it either moves closer to the finger (case 2) or further from the finger (case 3). Using the property that for all $a \geq 1$ and $b \geq 1$, $\lg(a+b) \leq \lg a + \lg b$, we bound the amortized cost of $c * \lg(|x-y|+1) + c * \lg(|f-y|+1) - c * \lg(|f-x|+1)$ for each case as follow:

$$
\begin{aligned}
\text{1) If} \quad x \leq f \leq y, \quad & \lg(y-x+1) + \lg(y-f+1) - \lg(f-x+1) \\
& \leq \lg(y-f+f-x+2) + \lg(y-f+1) - \lg(f-x+1) \\
& \leq \lg(y-f+1) + \lg(f-x+1) + \lg(y-f+1) - \lg(f-x+1) \\
& \leq 2 * \lg(y-f+1)
\end{aligned}
$$

$$
\begin{aligned}
\text{2) If} \quad x \leq y \leq f, \quad & \lg(y-x+1) + \lg(f-y+1) - \lg(f-x+1) \\
& = (\lg(y-x+1) - \lg(f-x+1)) + \lg(f-y+1) \\
& \leq \lg(f-y+1)
\end{aligned}
$$

$$
\begin{aligned}
\text{3) If} \quad y \leq x \leq f, \quad & \lg(x-y+1) + \lg(f-y+1) - \lg(f-x+1) \\
& \leq \lg(x-y+1) + \lg(f-y+1) \\
& \leq 2 * \lg(f-y+1)
\end{aligned}
$$

Since we did not make any assumption on the location of the finger f, this proof applies for all possible fingers. Moreover, this proof is tight when the queries are $f, x, f, x, f, x, \ldots$. □

Lemma 12. *[ST85] If a BST satisfies the traversal property, then it also satisfies the scanning property.*

Lemma 13. *[ST85] If a BST satisfies the dynamic finger property, then it also satisfies the scanning property.*

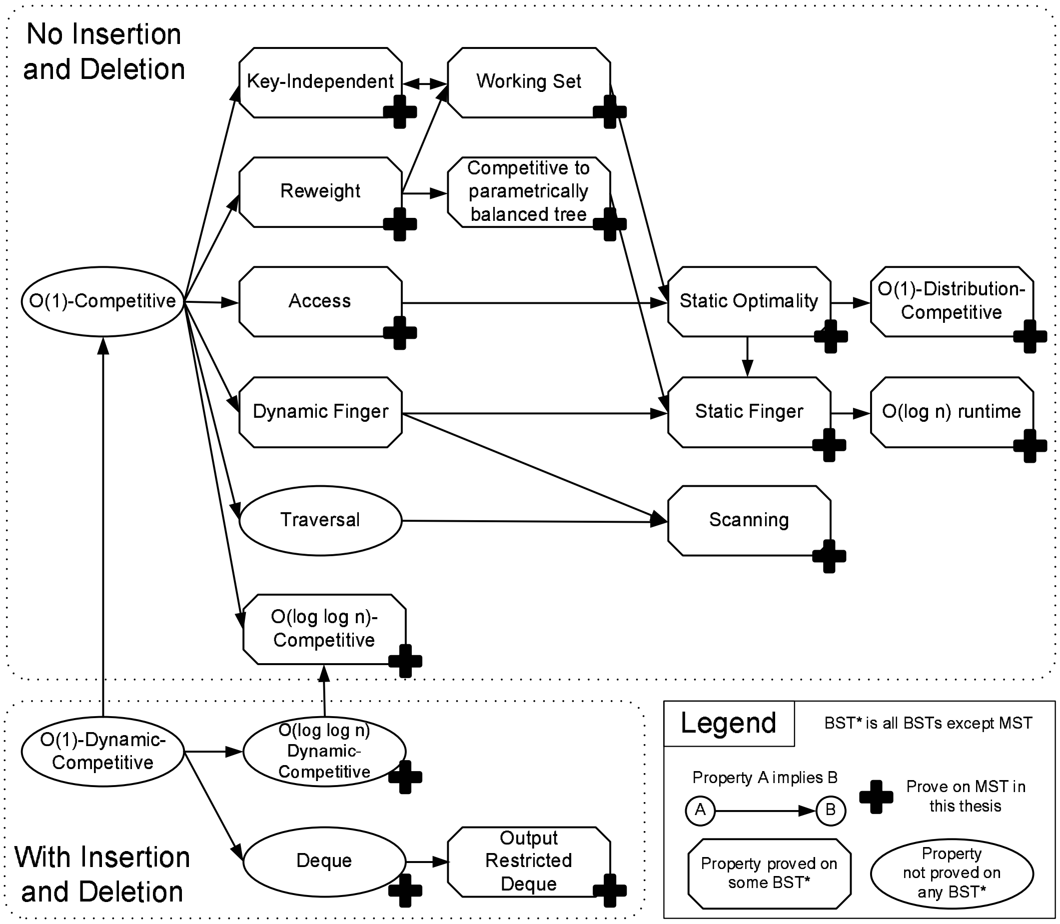

Figure 2.2: This figure shows the implication relationships for the list of properties in Section 2.4 and Section 4.4. The minimum set of edges are shown so that the transitive closure of the above graph includes all the implications. (MST stands for multi-splay tree in the legend.)

Chapter 3

Multi-Splay Trees

3.1 The Multi-Splay Tree Data Structure

Consider a *balanced*[1] 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 Chapter 5), 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.

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*. Al-

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

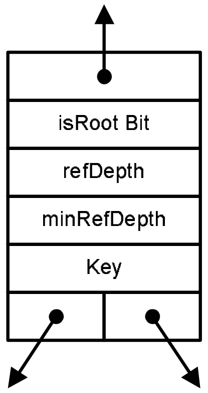

Figure 3.1: The fields of a node in a multi-splay tree.

though 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.[2] 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 $minRefDepth$ 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. Because the reference tree is a balanced tree [3], we only use $O(\log \log n)$ extra bits per node to store the additional informations.

3.1.1 Simplified Drawing of a Multi-Splay Tree

Throughout this thesis, we show many figures of multi-splay trees. For the sake of clarity, rather than showing the entire multi-splay trees with all the fields, we will simplify the drawing as shown in Figure 3.2. First, we will draw each multi-splay tree with its corresponding reference tree. We will always draw rectangular shapes for nodes of a multi-splay tree and circular shapes for nodes of a reference tree. Second, we will label each node with its key value, and mark each whose $isRoot$ bit is true with a thicker border. Third, because it is easy to derive the $refDepth$, and the $minRefDepth$ fields from the reference tree, we ignore those fields in the simplified drawing. Fourth, we will only draw the left and right

[2]Note that this quantity is static in our initial description of multi-splay trees, but becomes dynamic in Chapter 5 when we extend multi-splay trees to support insert and delete.

[3]In the Chapter 5, we generalize multi-splay tree to support insertions and deletions. We use red black tree as a reference tree, so we will also need an additional bit to store if a node is red or black.

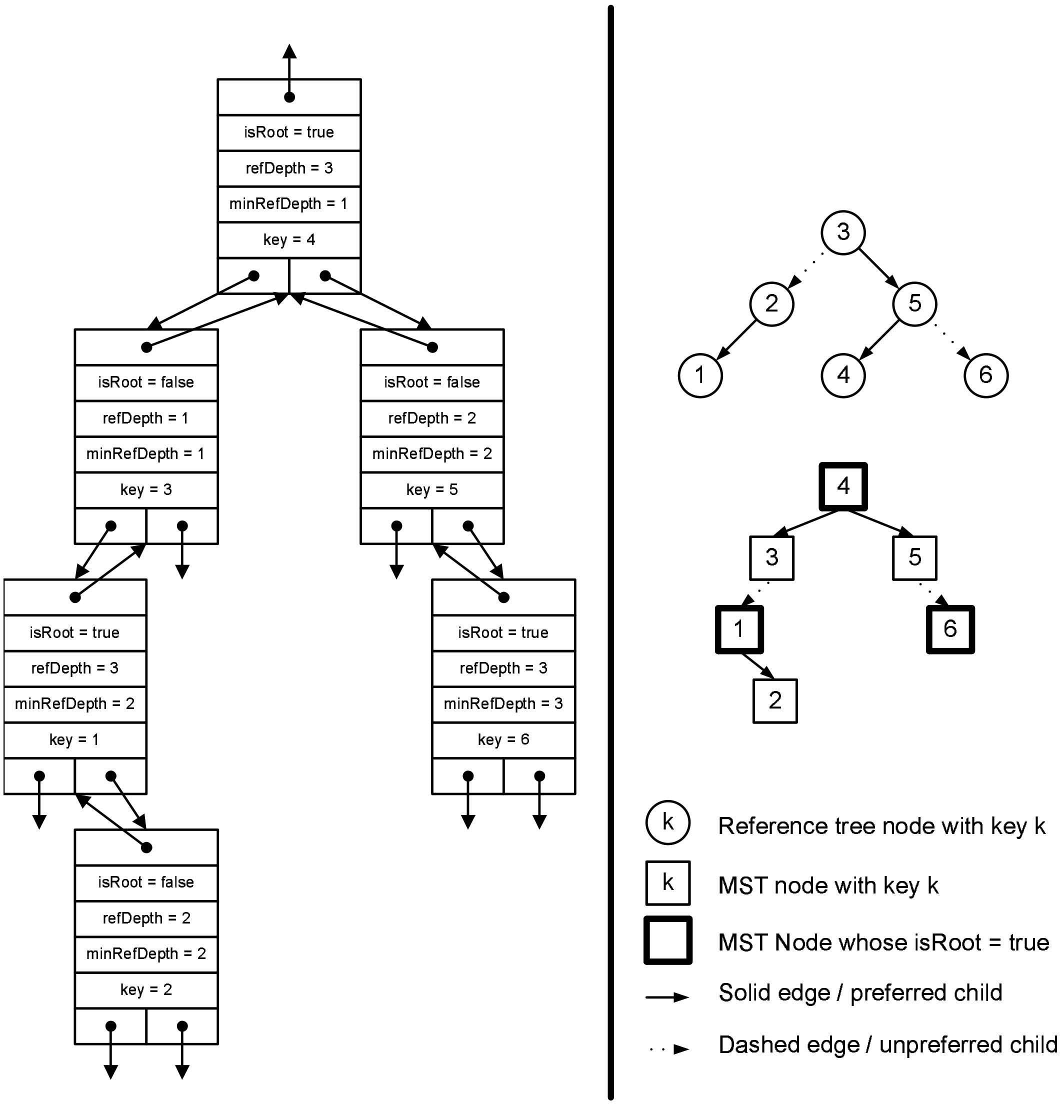

Figure 3.2: This figure shows a multi-splay tree with fields (left), and its simplified version (right). The simplification steps are described in Section 3.1.1

children pointers, and we ignore parent and nil pointers. Lastly, we use dashed arrows for pointers connecting nodes in two different splay trees, and solid arrows for the other pointers, even thought the types of pointers are already deducible from the $isRoot$ bit.

While it is easy to derive if a pointer connects nodes in two different splay trees using the $isRoot$ bit, for clarity, we use dashed arrows for pointers connecting nodes in two different splay trees, and solid arrows for other pointers. Using the simplified drawing, it is easy to derive the actual multi-splay tree. In this thesis, we will always use the simplified drawing.

3.1.2 Recursive Definition of a Multi-Splay Tree

This is an alternative recursive definition of multi-splay trees. First, we start with a fixed balanced tree P called the *reference tree*. Second, we pick a root-to-leaf path and call it a *preferred path*. Each node on the path has a *preferred child* defined by the preferred path, and we call the other child the *non-preferred child*. Third, we rotate the nodes on this preferred path to create a splay tree, S, and we mark the topmost node in S as a root. Fourth, we recurse the first step on the tree rooted at each non-preferred child of the preferred path. Each recursion returns a multi-splay tree. Fifth, to produce a single multi-splay tree, we connect the multi-splay trees to S's leaves in the symmetric order. Since each missing child of a binary search tree corresponds to an interval of keys, and the splay tree S has the same set of keys as the preferred path, each missing child of S corresponds to a non-preferred child's subtree.

3.2 The Multi-Splaying Algorithm

Like splay tree, there is a self-adjusting update algorithm that rotates a key to the root. This algorithm is called 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

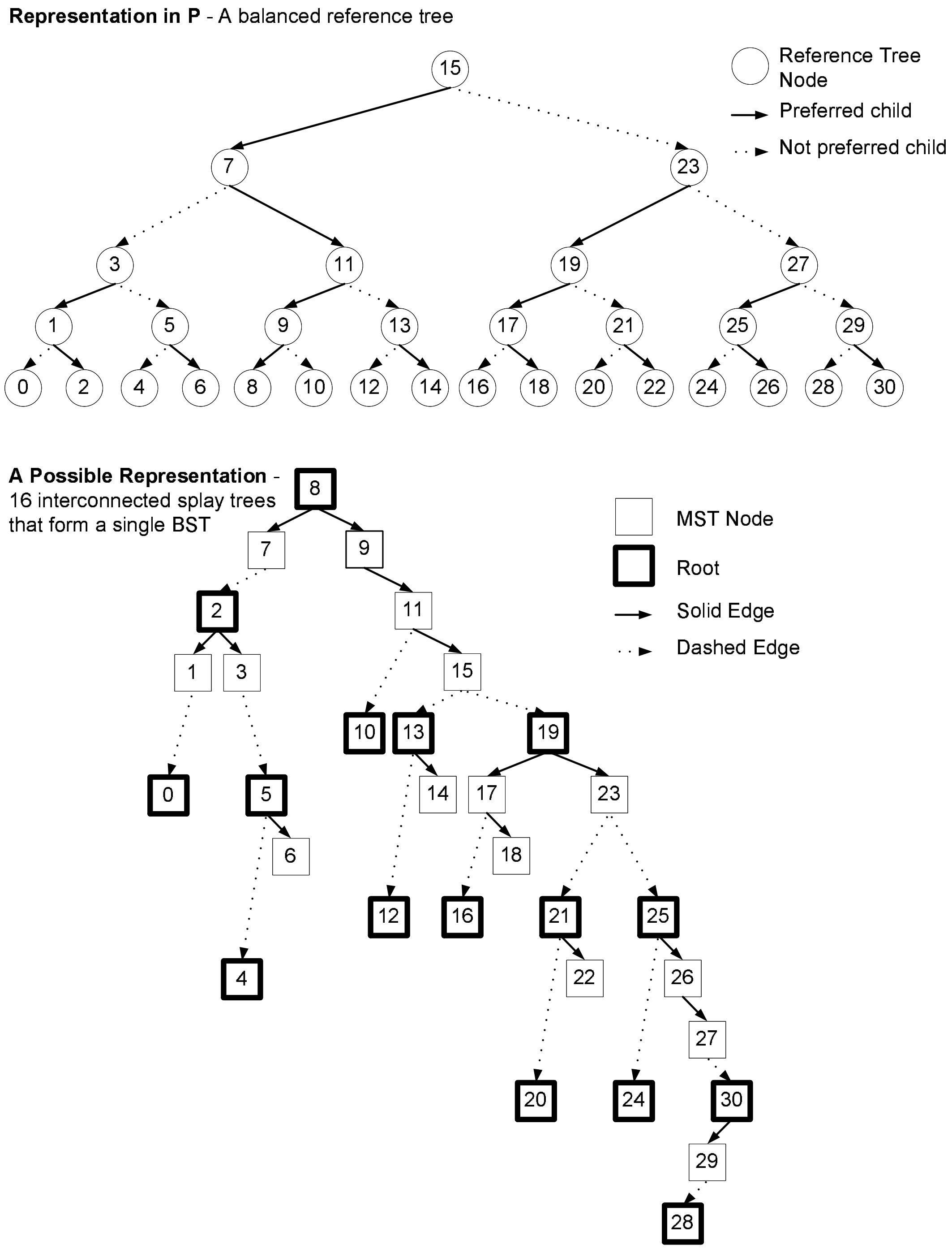

Figure 3.3: An example of a multi-splay tree and its corresponding reference tree.

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-splaying 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, the reference tree P is not our representation, the multi-splay tree T is. To achieve $O(\log \log n)$-competitiveness, we can only afford to spend $O(\log \log n)$ amortized time per switch. To simulate the series of switches in the reference tree P, we first traverse the multi-splay tree to find the queried node. While we traverse, we remember all the switches we need to perform. Then we perform those switches from bottom up. 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 regions of keyspace, separated by y (See Figure 3.4).

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 splay y. Then we first find x, the predecessor of L in S, using the $minRefDepth$ field. (Note that x is the largest node less than y with depth less than y, and x must be a member of U. Thus, to find x, we start from y's left child and) Then, we splay x until x becomes the left child of y. This ensures that the set of nodes in the right subtree of x is L. Thus, we mark the right child of x 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 z) in U until z becomes the right child of y, so that unmarking the $isRoot$ bit of the left child of z joins in R. As a detail, to prove multi-splay trees 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].

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

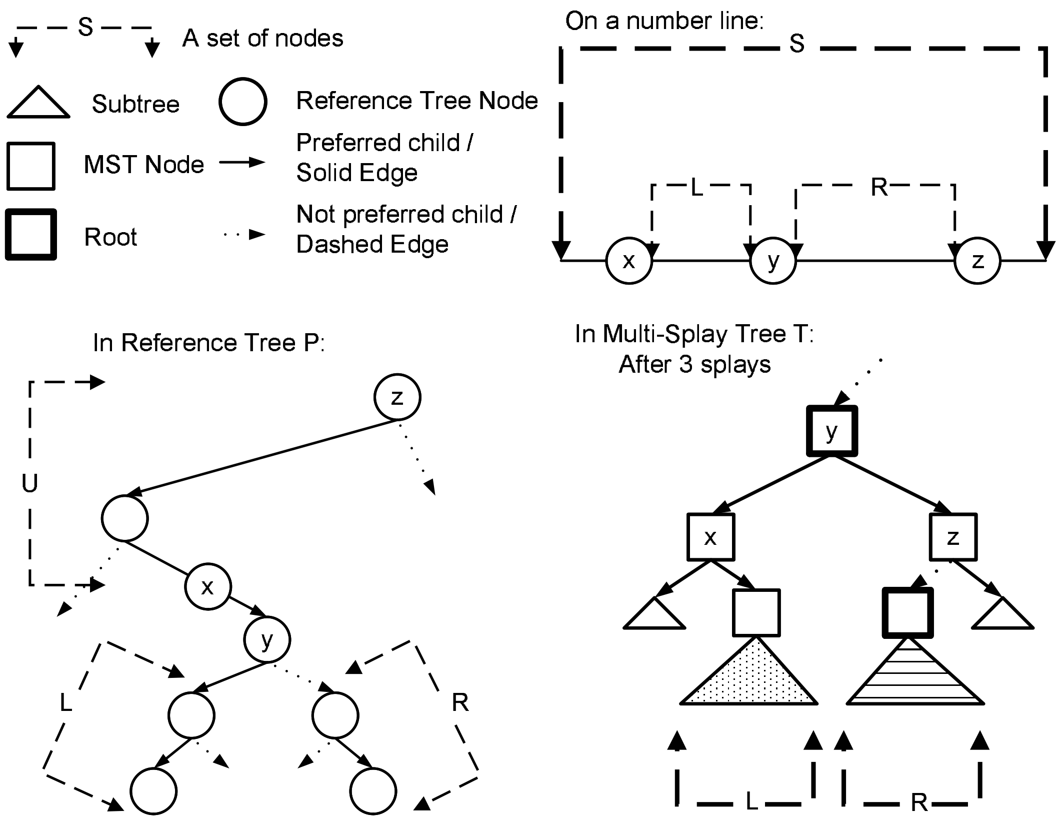

Figure 3.4: During a single left-to-right switch on y, this figure shows the graphical representations of S, U, L, R, x, y, and z immediately after the 3 splays.

node. Notice that this final switch brings the accessed node to 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. In addition, we have not discussed the boundary cases such as when x or z does not exist.

3.3 Details of Multi-Splaying Algorithm

3.3.1 Determining the Direction of a Switch

When serving a query σ_i for key $\hat{\sigma_i}$, we traverse the multi-splay tree T to find $\hat{\sigma_i}$. After we traverse, we perform a switch for each node r whose $isRoot$ bit is true (from bottom up). Because the reference tree is static, the switches on the same node must alternate between left-to-right switch and right-to-left switch. To determine the direction of a switch, we can store an extra bit to encode the direction of the last switch for each r (whose $isRoot$ bit is true). Thus, when we touch r, we can deduce the direction of a switch from the extra bit.

However, this extra bit is unnecessary, because we can also deduce the direction of a switch as follows. As we traverse down the multi-splay tree T to find $\hat{\sigma_i}$, we maintain $v_j =$ *pred*$(\hat{\sigma_i})$ and $w_j =$ *succ*$(\hat{\sigma_i})$ for the jth 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 jth 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 jth 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.

3.3.2 Switch on a Node with Missing Children

In this section, we describe how to switch a node in the reference tree with zero or one child. This type of switch only occurs at the final switch on the accessed node. Specifically, if the accessed node $\hat{\sigma_i}$ has zero or one child in the reference tree, then the final switch on $\hat{\sigma_i}$ still induces the corresponding splays, but no root marking will occur. In more detail, let $A(y)$ be the set of proper ancestors of y in the reference tree P. Let $refLeftParent(y)$ be the predecessor of y in $A(y)$, and define $refRightParent(y)$ analogously. When we perform a left-to-right (right-to-left) switch on a node y with missing children in the reference tree, we splay the $refLeftParent(y)$, y, and $refRightParent(y)$ as usual. (During a switch, we

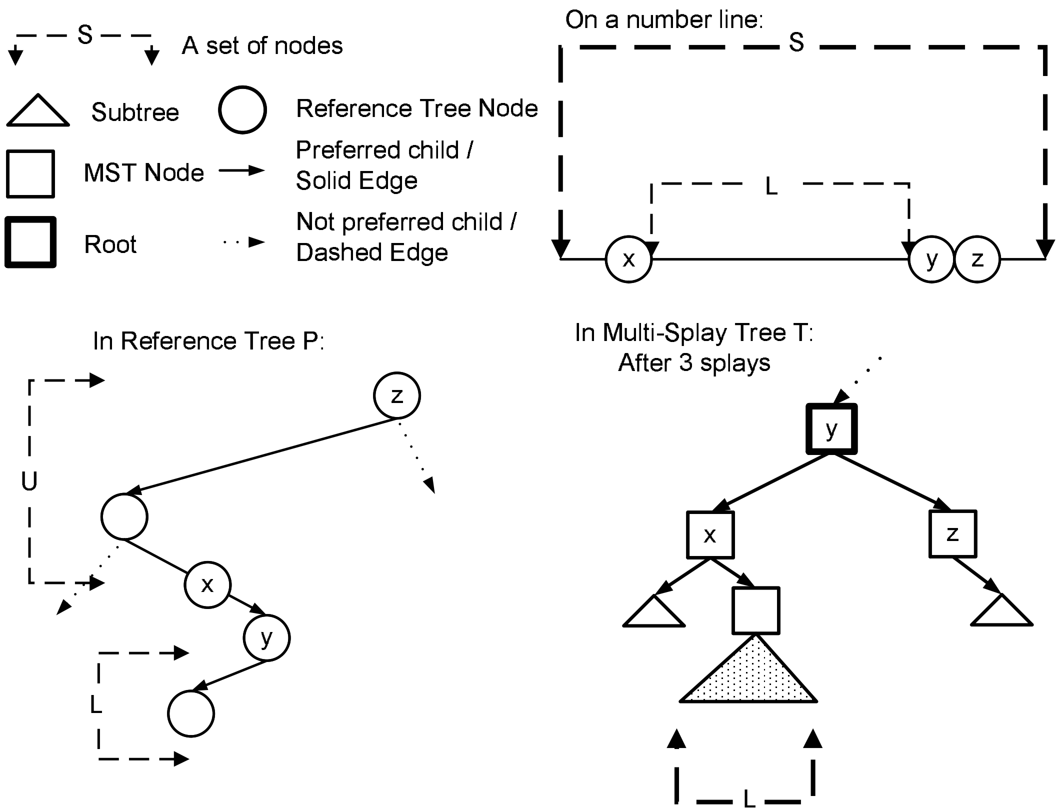

Figure 3.5: During a single left-to-right switch on y with a missing right child, this figure shows the graphical representations of S, U, L, x, y, and z immediately after the 3 splays.

typically use x to denote the $refLeftParent(y)$, and z to denote the $refRightParent(y)$.) If y has no left child in the reference tree, then the right child of the $refLeftParent(y)$ does not exist in y's splay tree. So we skip the marking (unmarking) of the $isRoot$ bit on that node. Similarly, if y has no right child in the reference tree, then the left child of the $refRightParent(y)$ does not exist. So we skip the unmarking (marking) of the $isRoot$ bit on that node. An example of a left-to-right switch with missing children is shown in Figure 3.5.

3.3.3 Switch without $refLeftParent$ or $refRightParent$

Since y's $refLeftParent$ and $refRightParent$ is not necessarily in y's splay tree, when we switch on a node y, we might not be able to find $refLeftParent(y)$ or $refRightParent(y)$.

Let x be $refLeftParent(y)$ and z be $refRightParent(y)$. For the proof of the multi-splay access lemma in Section 3.4.2, we can not afford to search for x and z if they do not exist. Specifically, we pay all the pointer traversal with rotations, so if we traversed to search for x or z and fail, we must splay the last node we touched as we traversed. We can not afford to pay the amortized cost of this splay in the analysis of the multi-splay access lemma. Fortunately, we can deduce if x and z exist by using the $minRefDepth$ field after we splay y. Since $refDepth(x) < refDepth(y)$, if $minRefDepth(\textbf{\textit{leftChild}}(y)) > refDepth(y)$, then x does not exist in y's splay tree. Similarly, because $refDepth(z) < refDepth(y)$, if $minRefDepth(\textbf{\textit{rightChild}}(y)) > refDepth(y)$, then z does not exist in y's splay tree.

If both x and z exist, then we proceed to mark and unmark as described in 3.2. If x does not exist, then the left $splaySubtree$ of y after splaying y is exactly L. (L and R are defined in Section 3.2.) Thus, during a left-to-right (right-to-left) switch, we mark (unmark) ***leftChild***(y)'s $isRoot$ bit. If z does not exist, then the right $splaySubtree$ of y after splaying y is exactly R. Thus, during a left-to-right (right-to-left) switch, we unmark (mark) ***rightChild***(y)'s $isRoot$ bit. An example of a left-to-right switch with missing children is shown in Figure 3.6.

3.4 Running Time Analysis

Theorem 5. *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 $O(\log^2 n)$. □

3.4.1 Multi-Splay Tree Satisfies $O(\log \log n)$-Competitive Property

For the purpose of this analysis, we define the potential of a multi-splay tree T as follows. If each node v has an arbitrary positive *weight* $w(v) = 1$, 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 $\sum_{v \in T} \lg s(v)$. In other words, the weight of each node in each splay tree is 1, and the potential of the multi-splay tree is the sum of the potentials of the splay trees.

Theorem 6. *For an arbitrary access sequence* $\sigma = \sigma_1 \cdots \sigma_m$ *in a multi-splay tree with* n *elements, the cost of* σ *is* $O(\textbf{\textit{OPT}}(\sigma) * \log \log n)$.

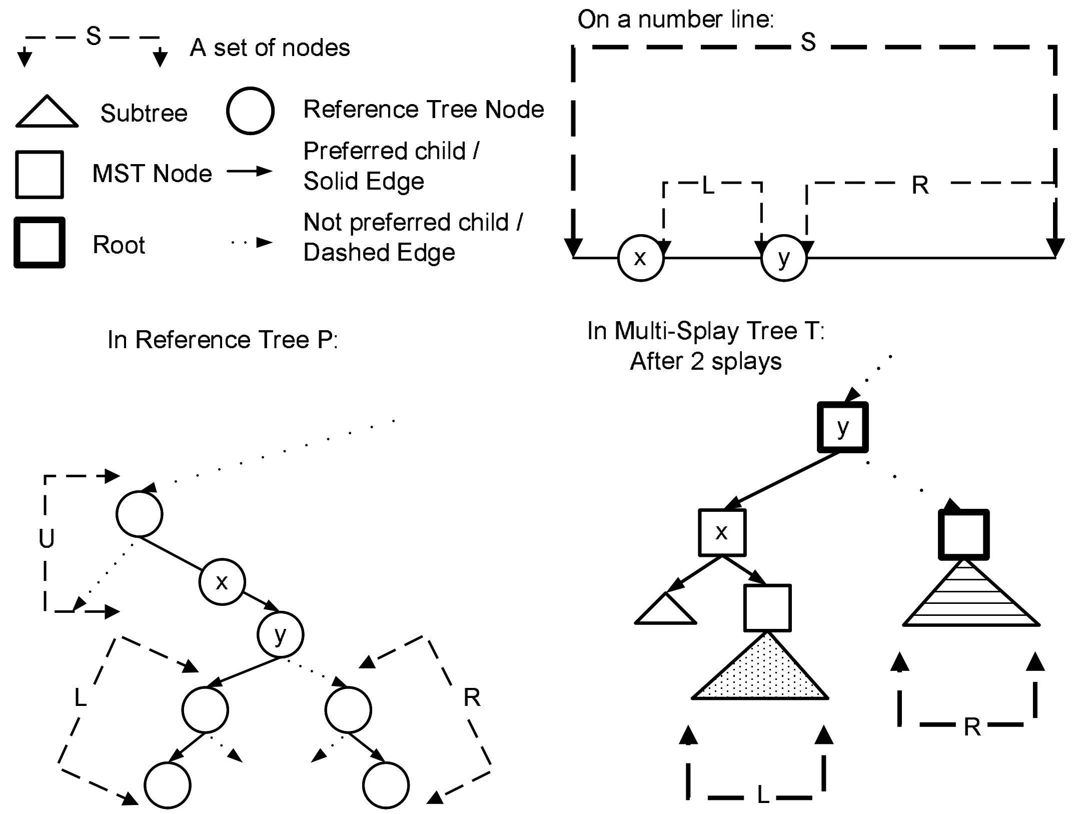

Figure 3.6: During a single left-to-right switch on y with a missing $refRightParent$, this figure shows the graphical representations of S, U, L, R, x, and y immediately after the two splays.

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 $O(\log \log n)$.

In this proof, we use y_i to denote the i^{th} node we switch, 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. Each switch at an arbitrary node y consists of up to 3 splays followed 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 $O(\log s(r_i))$, where r_i is the root of the splay tree corresponding to y_i's preferred path in the reference tree. Because $s(r_i) = O(\log n)$, the amortized cost of the 3 splays is $O(\log \log n)$.

The amortized cost of marking *child*(z) 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:

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

□

3.4.2 Multi-Splay Tree Satisfies Access Property

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

Theorem 7 (Multi-Splay Tree Access Lemma)**.** *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_i is $c_{ms}(\lg \frac{W}{\hat{w}(\sigma_i)}) + c_{msa}$, where $W = \sum_y mass(y)$, c_{ms} and c_{msa} are constants.*

Our overall approach to proving Theorem 7 will be to assign a set of weights to the elements of the BST T roughly based on mass assignment and repeatedly use the Reweighting Lemma of [Geo04]. The amortized cost of each switch will be bounded by the cost of the 3 splays according to the access lemma for splay trees, plus the change in potential due to adding/removing weight from the tree due to root markings (for the purposes 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:

$$\begin{aligned}
U(x) &= refSubtree(uchild(x)) \\
\hat{w}(x) &= mass(x) + \sum_{y \in U(x)} mass(y) \\
\Diamond(x) &= lip(x) \cup rip(x) \cup \{x\} \\
w(x) &= \max_{y \in \Diamond(x)} \hat{w}(y).
\end{aligned} \tag{3.1}$$

We assign a weight of $w(x)$ to each element in a multi-splay tree for the purposes of our analysis. The size, $s(x)$, of node x is equal to $\sum_{y \in splaySubtree(x)} w(x)$, where

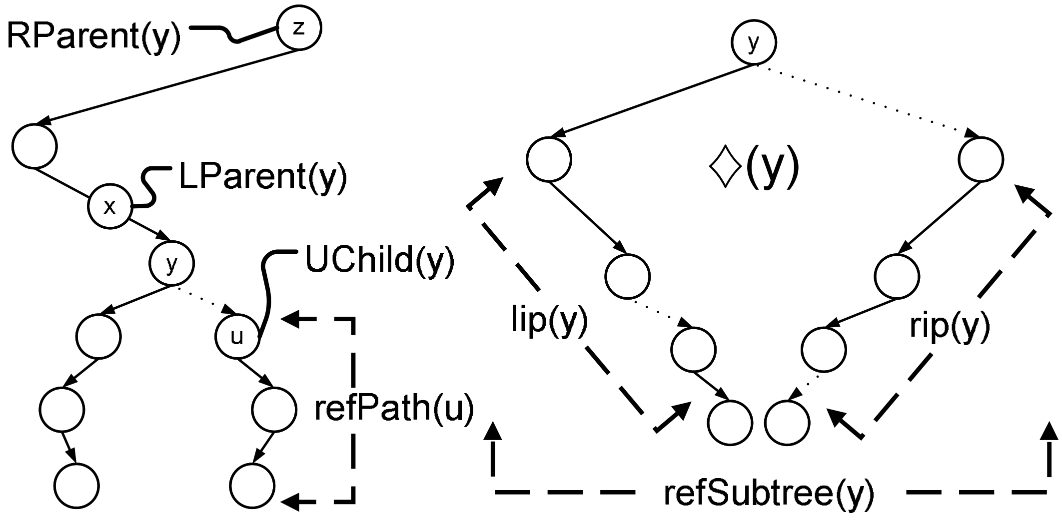

Figure 3.7: Notations in reference tree

$splaySubtree(x)$ is the subtree rooted at x of the splay tree containing x. The potential of T is $\sum_{x \in T} \log(s(x))$. If we were to use $\hat{w}(x)$ as the weight assignment, then this would essentially be the same weight assignment as used in link-cut tree analysis [ST85].

However, each time we switch x in a multi-splay tree, in addition to splaying x, we also splay $refLeftParent(x)$ and $refRightParent(x)$. On the other hand, in link-cut trees, we would only splay x. 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 $w(x)$ above gives us the following invariant.

Invariant 1. *For all nodes x, $w(refLeftParent(x)) \geq \hat{w}(x)$, $w(refRightParent(x)) \geq \hat{w}(x)$, and $w(x) \geq \hat{w}(x)$ whenever $refLeftParent(x)$ or $refRightParent(x)$ 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 of the switched nodes may change. Such a change in weight will be accounted for by using the Reweighting Lemma (Theorem 3).

Lemma 14. *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 $x \in P$,*

$$\sum_{y \in refPath(uchild(x))} w(y) \leq c_t * \sum_{u \in U(x)} mass(u) \leq c_t \hat{w}(x).$$

where $c_t = 3$.

Proof. First, it is clear that $3 * \sum_{u \in U(x)} mass(u) \leq 3\hat{w}(x)$ by the definition of $\hat{w}(x)$, so we only need to show that $\sum_{y \in refPath(uchild(x))} w(y) \leq 3 * \sum_{u \in U(x)} mass(u)$. In order to see this, we will show that $\sum_{y \in refPath(uchild(x))} (\hat{w}(y_L) + \hat{w}(y) + \hat{w}(y_R)) \leq 3 * \sum_{u \in U(x)} mass(u)$, where $y_L = \text{argmax}_{x \in lip(y)} \hat{w}(x)$ and y_R is defined analogously (y_L and/or y_R may not exist, in which case we assume $\hat{w}(y_L) = 0$ and/or $\hat{w}(y_R) = 0$).

Notice that $\sum_{y \in refPath(uchild(x))} \hat{w}(y) = \sum_{u \in U(x)} mass(u)$ 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 demonstrate this, for $y \in refPath(uchild(x))$, let $a_L = \text{argmax}_{z \in refPath(uchild(x)) \cap (A(y_L) \cup y_L)} refDepth(z)$, where $refDepth(z)$ is depth of z in P. Notice that $a_L \in lip(y)$ if it exists so that $refRightParent(a_L) = y$. Thus, each a_L that exists is distinct so $\sum_{y \in refPath(uchild(x))} \hat{w}(a_L) \leq \sum_{y \in refPath(uchild(x))} \hat{w}(y)$. Furthermore, $\hat{w}(y_L) \leq \hat{w}(a_L)$ because either $y_L = a_L$ or $y_L \in U(a_L)$. Thus, we have

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

□

With Lemma 14 in hand we are ready to prove the Multi-Splay 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 3 splays, and at most 2 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 Reweighting Lemma (Theorem 3), the amortized cost of the three splays is at most,

$$c_s * \left(\lg\left(\frac{s(r_i)}{w(y_i)}\right) + \lg\left(\frac{s(r_i)}{w(x_i)}\right) + \lg\left(\frac{s(r_i)}{w(z_i)}\right)\right) + 3c_{sa} \leq 3c_s * \lg\left(\frac{s(r_i)}{\hat{w}(y_i)}\right) + 3c_{sa}.$$

Second, it is free to mark the LCA of L_i as a root because this decreases potential of T. As for unmarking the LCA of R_i, by Lemma 14, the increase in $s(y_i)$ and $s(z_i)$ is bounded by $c_t * \hat{w}(y_i)$ 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 $2\lg(c_t + 1)$.

Third, after a switch on y_i, $\hat{w}(y_i)$ could 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,

$$\begin{aligned} w(x_i) &\geq \hat{w}(y_i) \\ w(y_i) &\geq \hat{w}(y_i) \\ w(z_i) &\geq \hat{w}(y_i) \end{aligned} \tag{3.2}$$

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 $w'(v) > w(v)$

$$w'(v) \leq s(r_i). \tag{3.3}$$

Thus, by the Reweighting Lemma, Equation 3.2, and Equation 3.3, the amortized cost of reweighting is at most $3c_r \lg(s(r_i)/\hat{w}(y_i))$, 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 14, $s(r_i) \leq c_t\hat{w}(y_{i+1})$, the series of k switches costs at most

$$\begin{aligned} \sum_{i=1}^{k}\left(c_{sw} * \lg\left(\frac{s(r_i)}{\hat{w}(y_i)}\right) + c_{swa}\right) &\leq c_{sw} * \sum_{i=1}^{k}\lg\left(\frac{c_t * s(r_i)}{s(r_{i-1})}\right) + c_{swa}k \\ &\leq c_{sw} * \lg\left(\frac{s(r_k)}{w(y_0)}\right) + (c_{sw}\lg c_t + c_{swa})k, \end{aligned}$$

where

$$\begin{aligned} c_{sw} &= 3c_s + 3c_r \\ c_{swa} &= 3c_{sa} + 2\lg(c_t + 1) \end{aligned}$$

Since k is smaller than the number of rotations needed for splaying y_0 the final switch, $(c_{sw} \lg c_t + c_{swa})$ can be charged to the last switch if each splaying step pays for an additional $(c_{sw} \lg c_t + c_{swa})$ units of work. The amortized cost of the final switch is

$$\begin{aligned} & \left(c_{sw} * \lg\left(\frac{s(splayRoot)}{\hat{w}(y_0)}\right) + cSwitchAdd\right) \\ & +(c_{sw} * \lg c_t + c_{swa})\left(c_s * \lg\left(\frac{s(splayRoot)}{\hat{w}(y_0)}\right) + c_{sa}\right) \\ = \; & (c_{sw} + c_s c_{sw} \lg c_t + c_s c_{swa}) \lg\left(\frac{s(r_k)}{\hat{w}(y_0)}\right) + (c_{swa} + c_{sa} c_{sw} \lg c_t + c_{sa} c_{swa}) \\ = \; & c_f \lg\left(\frac{s(splayRoot)}{\hat{w}(y_0)}\right) + c_{fa} \end{aligned}$$

where

$$\begin{aligned} c_f &= c_{sw} + c_s c_{sw} \lg c_t + c_s c_{swa} \\ c_{fa} &= c_{swa} + c_{sa} c_{sw} \lg c_t + c_{sa} c_{swa}. \end{aligned}$$

A multi-splay operation consists of a sequences of switches and a final switch. Thus, the total amortized cost of a multi-splay on σ_i is

$$\begin{aligned} c_{sw} \lg\left(\frac{s(r_k)}{w(\sigma_i)}\right) + c_{swa} \quad + \quad & c_f \lg\left(\frac{s(r_k)}{\hat{w}(\sigma_i)}\right) + c_{fa} \\ \leq \quad & (c_{sw} + c_f) * \lg\left(\frac{c_t W}{w(\sigma_i)}\right) + c_{fa} + c_{swa} \\ = \quad & (c_{sw} + c_f) * \lg\left(\frac{W}{w(\sigma_i)}\right) + (c_{sw} + c_f) \lg c_t + c_{fa} + c_{swa} \\ = \quad & c_{ms} \lg\left(\frac{W}{w(\sigma_i)}\right) + c_{msa} \end{aligned}$$

where

$$\begin{aligned} c_{ms} &= c_{sw} + c_f \\ c_{msa} &= (c_{sw} + c_f) \lg c_t + c_{fa} + c_{swa} \\ W &= \sum_y mass(y) \end{aligned}$$

□

We did not make any assumption on 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 and the minimum masses 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 initial and final potential is bounded by $O(n \log n)$.

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

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

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

Proof. The access lemma implies 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 Working Set Theorem for splay trees. 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 $mass(v)$ to $1/p(v)^2$. Note that $M = \sum_v mass(v) = O(1)$.

Whenever we query v, the cost of the query is $O(\log(M/mass(v)) + 1) = O(p(v) + 1)$ 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, $w(refLeftParent(v))$, $w(v)$, $w(refRightParent(v))$. 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 $O(1/mass(v)) = O(p(v))$. □

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

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

3.4.3 Multi-Splay Tree Satisfies Reweight Property

In this section, we extend the multi-splay tree access lemma to allow nodes to be "remassed" arbitrarily to any positive number, giving a result that is similar to the reweighting lemma for splay trees [Geo04]. As a simple corollary, it will follow that multi-splay trees satisfy the working set theorem.

Theorem 8 (Remassing Lemma)**.** *In a multi-splay tree T (where T also denotes the set of keys stored in the tree), let $mass_0, \ldots, mass_m$ be a sequence of mass functions where $mass_0 = mass_1$ such that $mass_i : T \to \mathbb{R}_{>0}$ for each $i \in \{1, \ldots, m\}$. Let $\sigma = \sigma_1, \ldots, \sigma_m$ be a sequence of accesses in T. The cost of accessing σ is at most*

$$\Phi_0 - \Phi_m + \sum_{i=1}^{m} \left(\log\left(\frac{W_i}{mass_i(\sigma_i)}\right) + \sum_{v \in T} \max\left(0, \log\left(\frac{mass_i(v)}{mass_{i-1}(v)}\right)\right)\right),$$

where $W_i = \sum_{v \in T} mass_i(v)$ and Φ_i represents the potential of T, as described in Section 3.4.2, after the i^{th} access (Φ_0 is the potential before the first access).

Before proving the remassing lemma, we need to extend the multi-splay tree access lemma to prove a "lazy" version of the remassing lemma. By lazy, we mean that the changes in weight between $mass_i$ and $mass_{i+1}$ are not applied for each node whose mass *increases* until that node is a member of $refPath(refRoot)$ (decreases in mass are still applied immediately, just as in Theorem 8), even though we charge a price of $\max(0, \log(\frac{mass'(v)}{mass(v)}))$ immediately whenever v is tagged to be remassed to $mass'(v)$ at a later time when its current mass is $mass(v)$. (Note that when v's actual remassing finally occurs when it becomes a member of $refPath(refRoot)$, say at time j, we remass it to $mass_j(v)$, its most recently assigned mass.) Let $mass'_i(v)$ denote the mass of node v during the i^{th} access using lazy remassing.

Informally, we can prove this lazy version of the remassing lemma simply by showing that the increase in T's potential due to an increase in mass at any one particular node v when v's remassing is finally applied after the j^{th} access is upper-bounded by $\log(\frac{mass'_{j+1}(v)}{mass'_j(v)})$. This will suffice to prove the lazy remassing lemma because we have collected $\sum_{k=i+1}^{j} \max(0, \log(\frac{mass_{k+1}(v)}{mass'_k(v)}))$ units of potential from each individual step since time i, the last time v was a member of $refPath(refRoot)$, and $\sum_{k=i+1}^{j} \max(0, \log(\frac{mass_{k+1}(v)}{mass'_k(v)})) \geq \log(\frac{mass'_{j+1}(v)}{mass'_j(v)})$ by the concavity of the log function.

We can show that the increase in the potential of T after the j^{th} access is upper-bounded by $3c_r * \log(\frac{mass'_{j+1}(v)}{mass'_j(v)})$ as follows. After v becomes a member of $refPath(refRoot)$ at time j, at most 3 nodes' weights will increase when v's mass increases from $mass'_j(v)$ to $mass'_{j+1}(v)$ (recall the definition of weight in Equation 3.1), and the ratio of the new weights of each of these nodes to its old weight is easily bounded by $\frac{mass'_{j+1}(v)}{mass'_j(v)}$. Hence, by the eager version Georgakopoulos's reweighting lemma, the cost of remassing v (accounted for by 3 eager splay tree reweighting operations performed in the splay tree $refPath(refRoot)$) is $3c_r * \log(\frac{mass'_{j+1}(v)}{mass'_j(v)})$), as suffices.

Note that we have ignored the initial and final potential of the tree, however this does not matter because the potential of the tree with eager remassing is always higher than the potential of the tree with lazy remassing. Also, for notational convenience, we do not perform lazy remassing after the m^{th} access on nodes who join T_r as a result of the m^{th} access.

Finally, we can prove the eager version of the multi-splay tree remassing lemma, as stated in Theorem 8, by arguing that the cost of eager remassing is at least as high as the cost of lazy remassing.

Proof. Rather than concern ourselves with all nodes whose mass change at a particular point in time, let us restrict our attention to a single node v whose mass changes from $mass_i(v)$ to $mass_{i+1}(v)$ after the i^{th} access. To prove Theorem 8, it suffices to show that the amortized cost of this remassing operation, $O(\max(0, \log(\frac{mass_{i+1}(v)}{mass_i})))$, is enough to pay for the cost of lazy reweighting. Then, we can simply apply this analysis to each node whose mass changes.

If $mass_{i+1}(v) \leq mass_i(v)$, then the lazy and eager versions of the remassing lemma coincide – both immediately remass v. On the other hand, consider the case in which $mass_{i+1}(v) > mass_i(v)$. Let W denote the total weight of T defined as $\sum_{v' \in T} w(v)$ assuming we eagerly remass nodes as per the theorem statement. Let W' denote the total weight of T if remassing is performed lazily. Notice that $W \geq W'$ at all times, so the amortized cost of each splay at a node other than v has not decreased using the analysis from the access lemma. On the other hand, when a splay is induced on v at time j, the cost according to the access lemma could be lower with eager remassing than with lazy remassing, but after this access is complete, v will be a member of $refPath(refRoot)$ so the lazy version of remassing resynchronizes with the eager version so that $mass_{j+1}(v) =$

$mass'_{j+1}(v)$.

Thus, it suffices to show that we can pay for this discrepancy merely by increasing the constant in front of $\log(\frac{mass_i(v)}{mass_{i-1}})$ in the amortized cost per remass with eager remassing. In particular, we show that paying $(3c_r + c_{ms}) * \lg(\frac{mass_i}{mass_{i-1}(v)})$ is enough.

In the lazy version of the remass lemma, the cost of splaying v and increasing the mass of v is $(c_{ms} \lg(W'_i/mass_(v)) + c_{msa} + 3c_r \lg(m'/mass(v)))$. In the eager version, the cost of splaying v and increasing the mass of v is $(c_{ms} \lg((W - mass(v) + m')/m') + c_{msa} + (3c_r + c_{ms}) * \lg(m'/mass(v)))$. It is easy to see that we always pay at least as much as the lazy version for each remass operation. Hence, $(3c_r + c_{ms}) * \lg(m'/mass(v))$ suffice to pay for each remass operation. □

Corollary 5. *[Geo04] Multi-splay trees are $O(1)$-competitive to parametrically balanced binary search trees.*

Proof. The remass property implies that multi-splay trees are $O(1)$-competitive to parametrically balanced binary search trees. □

3.4.4 Multi-Splay Tree Satisfies Scanning Property

We begin with several simple lemmas.

Lemma 15. *(Worst Switch Cost Lemma) The cost of a switch is $O(\log n)$* worst-case*, not amortized.*

Proof. Each switch consists of 3 splays and up to 2 root markings/unmarkings. Because the size of each splay tree is $O(height(P)) = O(\log n)$, the worst-case cost of the splays is $O(\log n)$, and clearly the root markings cost $O(1)$ worst-case. □

Lemma 16. *During a sequential access of all nodes of T, when a node with a left child (in P) is accessed, exactly one switch occurs.*

Proof. Within a sequential access, a query to a node v with a left child immediately follows a query to a node in its left ref-subtree, so the preferred path from the root includes v. The one switch occurs because the multi-splaying algorithm always switches the node that is accessed. □

Lemma 17. *(Touch Lemma) 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.)*

Proof. Observe that if all the rotations are performed on nodes in S, then the set of nodes S will always be a connected set of nodes that includes the root of T_S. A splay operation on $v \in S$ will rotate nodes on the path from v to the root. Because S consists of a connected set of nodes, all of these rotated nodes must be in S. Thus, the invariant that S is a connected set and $r \in S$ is maintained. □

Lemma 18. *During a sequential access sequence, when accessing nodes from the right ref-subtree R of y, the multi-splaying algorithm touches at most 2 nodes outside of R.*

Proof. After y is accessed, y is the root of the multi-splay tree, its right child z is the successor of R, and all the nodes of R are in z's left splay subtree (See Figure 3.4). The following splays induced by querying R can only touch y, R, and z by lemma 17. □

Lemma 19. *In a red-black tree T_{RB} of n nodes, $\sum_{v \in T_{RB}} \lg |subtree(v)| = O(n)$.*

Proof. Suppose we merge all the red nodes with their parents. For instance, if a black node originally has two red children and each red child has two black children, then we are left with a black node with 4 black children after the merge. (Essentially, we are converting the red-black tree into its corresponding 2-3-4 tree.)

Since every root-to-leaf path in a red black tree has the same number of black nodes, each black node can have at most two red children, and each red node has two black children, the merge process reduces the number of nodes in the subtree of every black node by at most a factor of 3.

Define $bh(v)$ to be the number of black nodes from v to a leaf, excluding v. Observe that the number of black nodes at $bh(v)$ is at most $\frac{n}{2^{bh(v)}}$. Also, note that the number of nodes in a black node v's subtree is at most $4^{bh(v)}$.

Hence,

$$
\begin{aligned}
\sum_{v \in T_{RB}} \lg |subtree(v)| &\leq 3 * \sum_{\text{black } v, v \in T_{RB}} \lg |subtree(v)| \\
&\leq 3 * \sum_{\text{black } v, v \in T_{RB}} \lg 4^{bh(v)} \\
&\leq 6 * \sum_{\text{black } v, v \in T_{RB}} bh(v) \\
&\leq 6 * \sum_{i=1}^{\lceil \lg n \rceil} i * \frac{n}{2^i} \\
&\leq 12n.
\end{aligned}
$$

$\square$

Theorem 9. *In any multi-splay tree T of n nodes, the cost of the access sequence $\sigma = \sigma_1, \cdots, \sigma_n$, where $\sigma_i < \sigma_{i+1}$ 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 a multi-splay tree T in terms of the cost of sequentially accessing subtrees of P. More specifically, we recursively account for the cost as follows:

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

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 $A(\textit{root}(t))$. Hence, we define

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

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 16, 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 $O(2+A(\textit{root}(t))+\log|t|) = O(A(\textit{root}(t))+\log|t|)$. The $O(A(\textit{root}(t))+\log|t|)$ bound is true because at most 2 nodes higher in P than *root*(t)'s right ascending path are touched as seen by Lemma 18, and the number of nodes in *root*(t)'s splay tree including *root*(t)'s right ascending path and below is $A(\textit{root}(t)) + height(t)$, which is $O(A(\textit{root}(t)) + \log|t|)$.

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)[4], 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 $O(a) = O(A(\textit{root}(t)))$ 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 $\textit{rightParent}^k(l)$ 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 19. □

3.5 Comment on the Fields of the Multi-Splay Tree Nodes

To proof that multi-splay trees have the $O(\log\log n)$-competitive property, we only need to store the *isRoot* field and the *refDepth* field for each multi-splay tree node. In the description of multi-splay tree algorithm in Section 3.2 and Section 3.3.3, we only use the

[4]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.

$minRefDepth$ field to find the $refLeftParent(y)$ ($refRightParent(y)$) during a left-to-right (right-to-left) switch on y. After we splay y during a left-to-right (right-to-left) switch on y, we can find the $refLeftParent(y)$ ($refRightParent(y)$) using Georgakopoulos's observation for the chain splaying algorithm in [Geo05]. Let x denotes $refLeftParent(y)$ and z denotes $refRightParent(y)$. Observe that for all the nodes u in y's left (right) subtree, if $refDepth(u) < refDepth(y)$, then $u \leq x$ ($u \geq z$), and if $refDepth(u) > refDepth(y)$, then $u > x$ ($u < z$). When we search for x (z) starting in y's left (right) child u, we set u to the right (left) child of u if $refDepth(u) < refDepth(y)$, and we set u to the left (right) child of u if $refDepth(u) > refDepth(y)$. We stop just before we set u to nil or leave y's splay tree, and u must be either x (z) or *succ*(x) (*pred*(z)). If $refDepth(u) > refDepth(y)$, then u is x (z). If $refDepth(u) < refDepth(y)$, then we splay u and *succ*(u) (*pred*(u)) to find x (z). Thus, we can find the $refLeftParent(y)$ ($refRightParent(y)$) without using the $minRefDepth$ field.

However, the above modification breaks our analysis of multi-splay tree access lemma. To proof the multi-splay tree access lemma in Section 3.4.2, we can only afford to splay nodes u such that $refDepth(u) \geq refDepth(y)$ during a switch on y. Because shallower nodes in the reference tree generally have larger weight, we charge the cost of all the splays during a switch on y to the cost of splaying y. If we splay *succ*$(refLeftParent(y))$ or *pred*$(refRightParent(y))$ during a switch on y, then the charging argument breaks because *succ*$(refLeftParent(y))$ and *pred*$(refRightParent(y))$ are deeper than y.

Chapter 4

Dynamic Binary Search Trees

4.1 Dynamic BST Model

Before we can reason about the properties and competitiveness of dynamic binary search 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 start from an empty tree and 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 pay for inserting the node at a leaf in addition to 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 point, the node can be removed).

During (or after) each operation, a BST algorithm may perform any rotations it wishes at the cost of one per rotation. The cost of an operation is simply the total number of nodes touched, plus the number of rotations. Without insertions and deletions, this definition would be identical to the one in Section 2.1.

In this model, we do not allow BSTs to swap nodes and contract edges during deletion. This implies that it must also pay for accessing both *pred*$(\hat{\sigma}_i)$ and *succ*$(\hat{\sigma}_i)$ while deleting

$\hat{\sigma}_i$. Because of this restriction, small modifications are necessary to include many standard binary search trees in this model.

4.2 Competitive Analysis on Dyanmic BST

In the standard BST model defined in Section 2.1, BST algorithms can not change the set of elements, so algorithms must start with a non-empty initial tree. Since there are many possible initial trees, we defined OPT(σ) to start with the best possible initial tree. However, the definition of optimal dynamic BST model is simpler, because every dynamic binary search tree algorithm starts with an empty tree. Since there is only one choice of initial tree, we use DOPT(σ) to refer to the cost of an optimal off-line dynamic BST algorithm executing σ.

An on-line binary search tree algorithm A is T-dynamic-competitive if

$$\forall \sigma A(\sigma) < T * DOPT(\sigma) + O(m)$$

Before we make multi-splay tree dynamic and prove that dynamic multi-splay tree is $O(\log \log n)$-dynamic-competitive in Chapter 5, we first prove a lower bound on the dynamic BST model.

4.3 Dynamic Interleave Lower Bound

With our new definitions, we must prove a new lower bound for DOPT(σ). Fortunately, techniques similar to those in [Wil89] suffice. Our new lower bound is a generalization of the one in [DHIP04], which is a variant of Wilber's first lower bound. Our lower bound generalize the interleave lower bound by allowing rotations in the reference tree. Allowing rotations is essential, without rotations, it is impossible to delete some of the nodes from the reference tree.

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 $\mathit{refSubtree}(v)$ 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, DIB(ρ, σ), by one. (ρ is a sequence of changes to the reference tree, and it is carefully defined in the proof.) 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. With rotations, this is a generalization of the original interleave bound even in the static BST model.

We will proof the theorem below.

Theorem 10 (Dynamic Interleave Bound)**.** *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 $\Omega(\mathit{DIB}(\rho, \sigma)/2 - n - 2k + cm)$, 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 ρ_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.[1]

The operations ρ_i are the changes to P that occur between successive operations of σ. (For multi-splay trees, 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.

4.3.1 Proof of the Dynamic Interleave Bound

Here we present an extended version of Wilber's first lower bound [Wil89]. Our presentation is similar to Demaine *et al.*'s, with modifications to permit the lower bound tree to be dynamic.

In our description of the bound, there are two trees, P and T, which are both dynamic BSTs over the same keys. The tree P is a *reference tree* that the lower bound will use (P does not really exist), and each internal node always has exactly one preferred child (like the reference tree for a multi-splay tree). The tree T refers to the tree maintained by an arbitrary BST algorithm A adhering to the model described in Section 4.1.

Let $\sigma = \sigma_1, \ldots, \sigma_m$ be a sequence of operations on T for which each σ_i is either a query, an insert, or a delete, and A is responsible for executing these operations in order.

[1] Although our model for BST deletion does not allow such swapping/splicing, multi-splay trees will only be *simulating* them while adhering to our dynamic BST model.

Because both P and T are dynamic, we often refer to them by their time index. By P_i and T_i, we mean the state of P and T right before σ_i is executed. For notational simplicity, both P and T are assumed to be empty initially (i.e., P_0 and T_0 are empty).

Further, because P is dynamic, we need a way to describe changes to it. Let $\rho = \rho_1, \ldots, \rho_m$ be a sequence of changes to P, where each ρ_i contains a *sequence* of rotations to be performed on P. Insertions and deletions in the reference tree correspond to the operations in σ and follow the standard BST insert and delete procedures. That is, an insertion occurs at the relevant leaf, and a deletion typically swaps the node v to be deleted with *pred*(v) and splices out v. The change in ρ_i is performed immediately before σ_i is executed by A (i.e., after σ_{i-1} is executed for $i > 1$). Note that ρ_1 and ρ_2 are always empty because there is at most one node in P prior to σ_2. Whenever a node in P is involved in a rotation (i.e., it is either v or p for a rotation of v over p), its preferred child is set to its leftmost child, if it has a child. This child setting is *not* considered a switch for accounting purposes (e.g., in $\text{DIB}(\rho, \sigma)$ as described below).

If σ_i queries $\hat{\sigma_i}$, P_i *switches* its nodes' preferred children as necessary so as to create a path consisting only of preferred child edges to $\hat{\sigma_i}$ starting from the root. In the case of insert, the switches connect both the predecessor and successor of $\hat{\sigma_i}$ to the root in succession. For delete, *pred*$(\hat{\sigma_i})$, *succ*$(\hat{\sigma_i})$, and $\hat{\sigma_i}$ are connected to the root in arbitrary order (note that the order only affects the lower bound by a constant additive term per deletion). Let $\text{DIB}_i(\rho, \sigma, v)$ be the number of switches of node v's preferred child that are made in P_i to accommodate σ_i. Let $\text{DIB}(\rho, \sigma, v) = \sum_{i=1}^{m} \text{DIB}_i(\rho, \sigma, v)$, and let $\text{DIB}(\rho, \sigma) = \sum_{v \in V} \text{DIB}(\rho, \sigma, v)$, where V is the set of all nodes that are inserted into P (and T) at some point.

Our lower-bound proof runs parallel to the proof for a static reference tree in [DHIP04], with some changes to allow P to be dynamic. We define $L^y = \textit{leftRefSubtree}(y) \cup y$ and $R^y = \textit{rightRefSubtree}(y)$ (L^y and R^y can be indexed by time as well). For a node y, define the *transition point* of y to be the highest node z in T such that the path from z to the root contains at least one node from both L^y and R^y. Observe that z is either the lowest common ancestor of L^y or of R^y.

We restate a few useful lemmas from [DHIP04]. Lemma 21 has been modified to account for P's being dynamic. The proofs of Lemmas 20 and 22 are the same as in [DHIP04] because these lemmas refer to a snapshot of P.

Lemma 20. *[DHIP04] The transition point z in T_i for a node y in P_i is unique.*

Lemma 21. *Suppose a BST access algorithm does not touch a node z in T for the time interval $i \in [j, k]$, and z is the transition point in T_j for a node y in P_j. Further, suppose that y is not rotated in the reference tree by the execution of $\rho_{j+1}, \ldots \rho_k$ (i.e., there is no* *rotation in $\rho_{j+1}, \ldots \rho_k$ of v over its parent p where $y = v$ or $y = p$). It follows that z remains the transition point of y for the entire time interval $[j, k]$.*

Proof. Suppose, without loss of generality, that $z \in R_j^y$. Notice that all of R_j^y is in the subtree rooted at z in T_j because z is the lowest common ancestor of R_j^y in T_j. Because z is not touched, z remains the lowest common ancestor of R_i^y for all $i \in [j, k]$.[2] Moreover, at time j the predecessor a of the nodes in the set $subtree(z) \cap (L_j^y \cup R_j^y)$ is in L^y because $L^y \cup R^y$ forms a contiguous region of keyspace. Notice that a is the deepest left-ancestor of z in T.[3] Thus, no rotation in $\rho_{j+1}, \ldots, \rho_k$ changes the fact that a is the deepest left-ancestor of z, and a cannot be deleted from T during $[j, k]$ because it has a right child. □

Lemma 22. *[DHIP04] At any time i, no node in T_i is the transition point for multiple nodes in P_i.*

The following theorem relates DIB(ρ, σ) to a lower bound on DOPT(σ):

Theorem 10. *(Dynamic Interleave Bound) 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 $\Omega(DIB(\rho, \sigma)/2 - n - 2k + cm)$, 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 ρ_i)).*

Proof. First, note that the cm term in the lower bound appears because each operation costs at least a constant c.

Following [DHIP04], suppose every time a node y in P is switched from left to right the lower bound places a marble on the transition point of y in T. Moreover, whenever the lower bound rotates v over p in P, it removes any marbles from the transition point of v and of p in T. On the other hand, whenever A touches a node, it discards all of the marbles at that node, and when A deletes a node y the lower bound removes the marble from y's transition point z if z exists and still has a marble after A deletes y. Clearly, if the number of marbles sitting on a node never exceeds 1 then the number of marbles removed is at most A's cost for σ.

Moreover, to prove the theorem it suffices to show that no node can ever have more than one marble. Because the number of marbles placed is at least half the number of

[2] Notice that z remains a member of R_i^y because if it needs to be swapped as a result of its successor's being deleted, our model dictates that the BST algorithm must access z in T, contradicting our assumption that the algorithm does not touch z in T.

[3] By "deepest left-ancestor of z", we mean the parent of the highest node in z's right ascending path.

total switches (because there are at least as many left-to-right switches as right-to-left switches[4]) and A must remove all of the marbles that are placed on T except those that either remain on T_m at the end (up to n) or are removed by the lower bound (up to $2k$ removed for rotations and up to m removed for deletions).

To see that no node can ever have more than one marble, notice that by Lemma 22 no two nodes in P_i ever have the same transition point in T_i. As argued in [DHIP04], when a left-to-right switch is made at y at times i and j ($i < j$), the transition point for y in T_i must be touched at some time during the interval $(i, j]$, assuming that the transition point remains constant during that interval. By Lemma 21, y's transition point z during this interval remains constant unless A touches z in T, in which case A removed its marbles, or the lower bound executed a rotation involving y, in which case the lower bound removed the marbles of z. □

4.4 Properties of an $O(1)$-dynamic-competitive BST

Partly because competitive analysis in standard BST model is already difficult, there are few results on the dynamic BST model. Below is a list of properties researchers have considered or mentioned. In this thesis, I will prove that multi-splay trees satisfy all of the following properties. Of the properties below, only multi-splay trees are known to satisfy both the $O(\log \log n)$-dynamic-competitive property and the deque property.

Property. A dynamic binary search tree has the $O(\log n)$ *dynamic runtime* property if a sequence of m operation is executed in time $O(m \log n)$.

In the worst case, some sequences will need $\Omega(m \log n)$ time using the sorting lower bound. Thus, having this property implies the data structure is theoretically optimal under worst-case analysis. Almost every dynamic binary search tree has the $O(\log n)$ runtime property.

Property. [Tar85] A dynamic binary search tree has the *deque* property if a sequence of m push, pop, inject and eject operation is executed in time $O(m + n)$.

Splay trees are conjectured to satisfy the deque property [Tar85]. Lucas [Luc88] showed that the total cost of a sequence of ejects and pops is $O(n\alpha(n, n))$ if the initial tree is a simple path of n nodes. Currently, the best bound is proved by Sundar

[4] This is true if we do not count the at most m right-to-left switches following the insertion of a node as a left child of a node that has a right child.

[Sun89a, Sun89b, Sun92]. He showed that splay trees can execute a sequence of m deque operations on n nodes in $O((m+n)\alpha(m+n, n+n))$.

Tarjan [Tar85] proved that splay tree satisfy a special case of the deque property - the output restricted deque property.

Property. A dynamic binary search tree has the *output restricted deque* property if a sequence of m push, pop and inject operation is execute in time $O(m+n)$

Property. A dynamic binary search tree has the $O(\log\log n)$-*dynamic-competitive* property if it execute every sequence σ of queries, inserts and deletes in time $O(\log\log n * DOPT(\sigma))$.

Multi-splay tree is the only data structure proved to satisfy this property. However, it may be possible to prove this property on other $O(\log\log n)$-competitive BSTs with some small modifications.

Chapter 5

Dynamic Multi-Splay Trees

5.1 Making Multi-Splay Tree Dynamic

With some modifications, our data structure can support insertions and deletions while maintaining the competitiveness and the runtime property. 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 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 on 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 a multi-splay tree so it is important that the corresponding reference tree requires only $O(m)$ reference tree traversals and reference tree rotations during a sequence of m operations. (Finding the *location* of the update does *not* involve reference tree traversals.) To emphasize that the reference tree is not explicitly maintained, we call each reference tree traversal a *virtual traversal*, and each reference tree rotation a *virtual rotation*. Red-black trees meet this requirement because they require only $O(1)$ amortized time to rebalance after an insert or delete [Tar83]. Because the reference tree is a red black tree, we also need an additional bit to store if a node is red or black.

5.2 Simulating Pointer Traversal in the Reference Tree – Virtual Traversal

To simulate a pointer traversal in the reference tree from node v in a multi-splay tree, we need to locate $refParent(v)$, $refLeftChild(v)$, and $refRightChild(v)$. In this section, we show how to find these nodes with a constant number of switches. [1]

5.2.1 Locating Child in the Reference Tree

To find the $refLeftChild(v)$, we perform four switches. First, we switch v so the set of nodes in the $refSubtree(refLeftChild(v))$ is identical to the set of nodes in $splaySubtree(\textbf{rightChild}(\textbf{leftChild}(v)))$, which we will refer to as T_l. If $refLeftParent(v))$ is not in v's splay tree, then $T_l = splaySubtree(\textbf{leftChild}(v))$. Then we search in the T_l for the node l with minimum $refDepth$ in T_l using the $minRefDepth$ field. The node l must be the $refLeftChild$ of v. Finally, we switch l twice and switch v again.

In our design, we switch l so the cost of searching for l is dominated by the switch cost, so we only need to account for the switch cost in our analysis. We switch l and v twice so the virtual traversal does not change the preferred path. Note that while the second switches on l and v are not necessary, they simplify some of the running time analysis.

Likewise, to find the $refRightChild(v)$ in four switches, we first switch v so the set of nodes in the $refSubtree(refRightChild(v))$ is identical to the set of nodes in $splaySubtree(\textbf{leftChild}(\textbf{rightChild}(v)))$, which we will call T_r. (If $refRightParent(v))$ is not in v's splay tree, then $T_r = splaySubtree(\textbf{rightChild}(v))$.) Then we search for the node r with minimum $refDepth$ in T_r using $minRefDepth$ field in T_r. The node r must be the $refRightChild$ of v. Finally, we switch r twice and switch v again. Thus, if $refLeftChild$ of v and $refRightChild$ of v do exist, then we can find them in four switches.

If $refLeftChild$ of v does not exist, then after we switch v, the left $splaySubtree$ of v will be empty. Similarly, if $refRightChild$ of v does not exist, then the right $splaySubtree$ of v will be empty after we switch v. Thus, we can determine if $refLeftChild$ of v or $refRightChild$ of v exist with a single switch.

In fact, the second switch on v, l, and r only consists of root marking, and the amount of potential change due to the marking and unmarking is the same as the first switch on v, l, and r. Thus, the second switches are free, and we only need to pay for two switches to

[1] In our original paper [WDS06], we added 3 new fields to store the values of the $refParent$, $refLeftChild$ and $refRightChild$ of each node. While it is a simpler solution, it uses $3 * \log n$ extra bits.

find either $refLeftChild$ or $refRightChild$.

5.2.2 Locating Parent in the Reference Tree

Observe that $refParent(v)$ must be either $refLeftParent(v)$ or $refRightParent(v)$, and $(refDepth(refParent(v)) = refDepth(v) - 1)$. To find $refParent$ of v, we first switch v twice. If $refParent(v)$ is in v's splay tree, then it must be splayed to either v's ***leftChild*** or ***rightChild***. Because each splay tree is a preferred path, there are at most one node of each $refDepth$ in a single splay tree. Thus, if the $refDepth$ of ***leftChild***(v) or ***rightChild***(v) equals to $refDepth(v) - 1$, then that node is v's $refParent$.

If $refParent(v)$ is not in v's splay tree, then v is $refParent(v)$'s non-preferred child. If we let T_p be the splay tree containing ***parent***(v), then $refParent(v)$ must be either the ***succ***(v) or ***pred***(v) in T_p. Now we could simply search for v's key in T_p, and stop when we find a node whose $refDepth$ equals to $refDepth(v) - 1$. In fact, when $refParent(v)$ is not in v's splay tree, we do not even have to search for it, because of the following lemma.

Lemma 23. *If v is a non-preferred child, then $refParent(v)$ must appear on the path from $splayRoot$ to v in multi-splay tree. Moreover, if we let r be the root of splay tree containing v, then $refParent(v)$ must be either **pred**$(subtree(r))$ or **succ**$(subtree(r))$.*

Proof. Since v is a non-preferred child, the set of nodes in $refSubtree(v)$ is identical to the set of nodes in $subtree(r)$. Hence,

$$\begin{aligned} refLeftParent(v) &= \textbf{\textit{pred}}(refSubtree(v)) = \textbf{\textit{pred}}(subtree(r)) \\ refRightParent(v) &= \textbf{\textit{succ}}(refSubtree(v)) = \textbf{\textit{succ}}(subtree(r)). \end{aligned}$$

Observe that for every node in every binary search tree, ***pred***$(subtree(x))$ and ***succ***$(subtree(x))$ must be ancestors of x. Thus, $refLeftParent(v)$ and $refRightParent(v)$ are ancestors of r which is an ancestor of v.

□

In this section, we have shown how to find the $refParent$, the $refLeftChild$, and the $refRightChild$ of any node. Moreover, all the pointer traversals in multi-splay tree can be paid by the cost of the switches, and we only need to pay for two switches per virtual traversal at most.

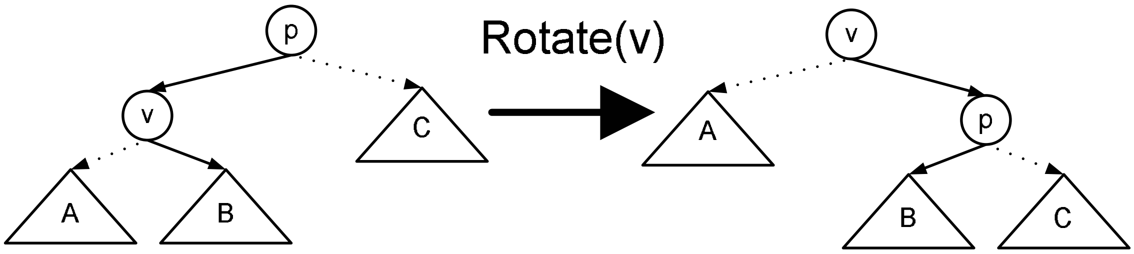

Figure 5.1: Before a right (left) rotation on v in the reference tree, we must make sure v's preferred child is right (left), and p's preferred child is v. Note that the partition of the nodes by the preferred paths remains the same before and after the rotation.

5.3 Simulating Rotations in the Reference Tree – Virtual Rotation

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 child by performing either 1 or 2 switches on v and v's parent. By meeting these requirements T ensures that the partition of preferred paths in the reference tree remains the same before and after the rotation,, as seen in Figure 5.1.

Similarly, to simulate a left rotation of a node v over its parent p in the reference tree, a multi-splay tree first ensures that v's preferred child is its left child, and v's parent's preferred child is its right by performing either 1 or 2 switches on v and v's parent. After the rotation, v's preferred child is set to the left child (which is p), and p's preferred child is set to right child. Thus, the set of nodes on each preferred path remains the same before and after the rotation.

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 $refDepth$ (the depth of v in the reference tree), and $minRefDepth$ (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 $refDepth(v) - refDepth(\textit{\textbf{parent}}(v))$, and $minRefDepth(v) - minRefDepth(\textit{\textbf{parent}}(v))$, except if v is the root of T, in which case it simply stores its $refDepth$, and $minRefDepth$. 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 multi-splay tree 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 $refDepth$

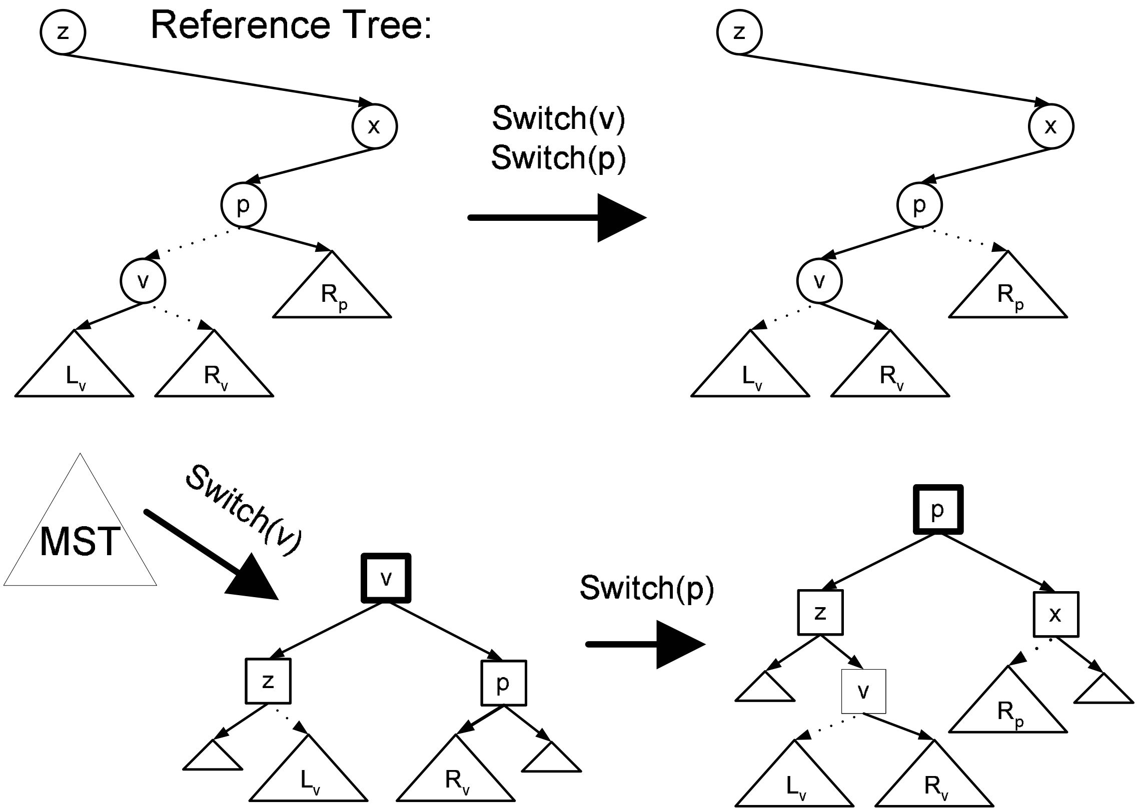

Figure 5.2: Observe that after we call *switch*(v) and *switch*(p), the sets of nodes in L_v and R_p form two subtrees in a multi-splay tree. A rotation of v over p in the reference tree decreases the depth value of each of the nodes in L_v by one, and increases the depth value of each of the nodes in R_p by one (Shown in Figure 5.1). Because L_v and R_p are grouped together by the switches, the updates in depth values cost $O(1)$ after performing the switches.

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), 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 5.2 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 $minRefDepth$ 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.

5.4 Implementing Insertion

To insert $\hat{\sigma_i}$, we query its successor or predecessor, then we perform a normal BST insert, and we set the appropriate fields of $\hat{\sigma_i}$ and its (constant number of) ancestors. We can find $refParent(\hat{\sigma_i})$ as we search for $\hat{\sigma_i}$ in the multi-splay tree, because $refParent(\hat{\sigma_i})$ is the node of maximum $refDepth$ on the access path. Then we rebalance the reference tree using amortized $O(1)$ simulated rotations and pointer traversals. Finally, we query $\hat{\sigma_i}$ again to bring it to the root of the multi-splay tree.

To elaborate on the above summary, let x be the ***pred***(σ_i) and z be the ***succ***(σ_i). Because $\hat{\sigma_i}$ is not in the multi-splay tree, when we search for $\hat{\sigma_i}$, we must touch both x and z. Then we query x or z depending on its $refDepth$. Note that because x is ***pred***(z) before the insertion, $refDepth(x) \neq refDepth(z)$. If $refDepth(x) > refDepth(z)$, we query x. (Because both x and z are splayed during this query, so the cost of pointer traversals are dominated by rotations.) After querying x, x becomes the root of the multi-splay tree and z becomes the right child of x as shown in Figure 5.3. Then we insert $\hat{\sigma_i}$ as the left child of z. On the other hand, if $refDepth(x) < refDepth(z)$, we query z instead. After querying z, z becomes the root of the multi-splay tree and x becomes the left child of z as shown in Figure 5.3. Then we insert $\hat{\sigma_i}$ as the right child of x.

Since $\hat{\sigma_i}$ must be the child of the deeper of x and z in the reference tree, we set the $refDepth(\hat{\sigma_i}) = \max(refDepth(x), refDepth(z)) + 1$. Since only x and z are the ancestors of $\hat{\sigma_i}$, only the fields of x and z can change after inserting $\hat{\sigma_i}$. Thus, we can update all the field changes due to this insertion in $O(1)$ time.

After the insertion, we might need to virtually rebalance the reference tree. Since the deeper of the two, x and z, is $refParent(\hat{\sigma_i})$, we already know the location of this

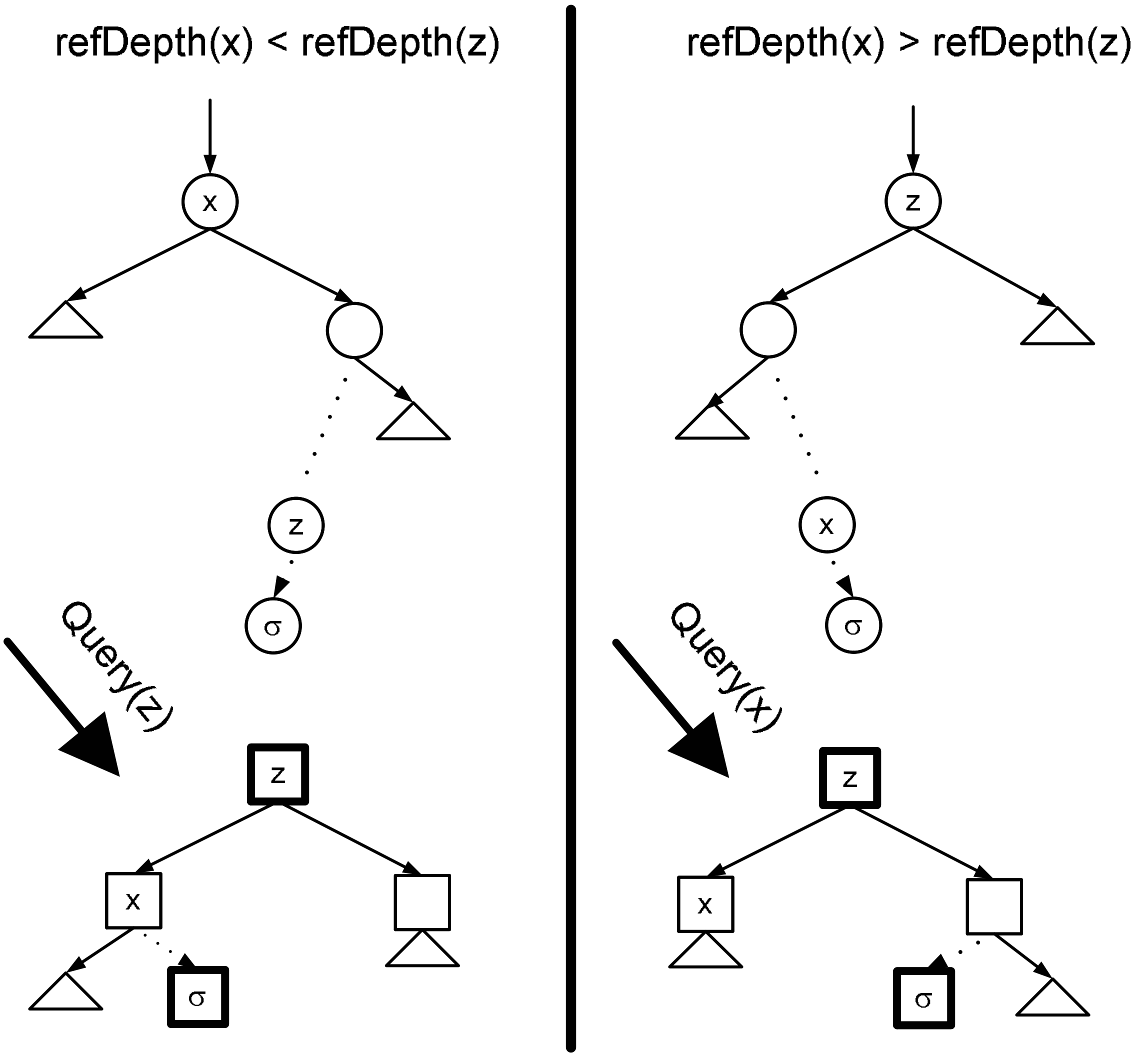

Figure 5.3: After querying x or z during an insertion.

insertion in the reference tree. In a red-black tree, we only need amortized $O(1)$ pointer traversal and rotations for rebalancing. We can perform each of those virtual traversals and virtual rotations in a constant number of switches as shown in Section 5.3 and Section 5.2. Finally, we query $refParent(\hat{\sigma_i})$ again to bring it to the root of multi-splay tree.

The above discussion omits a few trivial details, such as the first insertion, and the insertion of the smallest or the largest element. If σ_i is the first insertion, we can just create the node in $O(1)$ time. If $\hat{\sigma_i}$ is smaller than all the existing elements, then x does not exist. In this case, we query z and insert $\hat{\sigma_i}$ as the left child of z. Then we set $refDepth(\hat{\sigma_i})$ to $refDepth(z) + 1$ and proceed to virtually rebalance the reference tree as usual. Similarly, if $\hat{\sigma_i}$ is larger than all the existing elements, then z does not exist. We query x, insert $\hat{\sigma_i}$ as the right child of x, set $refDepth(\hat{\sigma_i})$ to $refDepth(x) + 1$, and proceed as usual.

5.5 Implementing Deletion

In the reference tree (or the standard red-black tree), a node $\hat{\sigma_i}$ has zero, one, or two children. To delete $\hat{\sigma_i}$ when it has no children, we can simply remove it because it is a leaf. When $\hat{\sigma_i}$ has one children, we can contract it in the red-black tree. If $\hat{\sigma_i}$ has two children, then we first swap $\hat{\sigma_i}$ with its predecessor, then contract $\hat{\sigma_i}$. In the following paragraphs, we describe how to simulate each of the above steps in the multi-splay trees.

To delete $\hat{\sigma_i}$ in the multi-splay tree, we first query $\hat{\sigma_i}$ If $\hat{\sigma_i}$ has no children in the reference tree, then we switch $refParent(\hat{\sigma_i})$ so $\hat{\sigma_i}$ becomes a leaf in multi-splay tree. Then we switch $refParent(\hat{\sigma_i})$ so $\hat{\sigma_i}$ is the only element in its splay tree. Once $\hat{\sigma_i}$ is a leaf and the only element in its tree, we could simply remove it without affecting the fields of any other nodes.

If $\hat{\sigma_i}$ has exactly one child in the reference tree, then we query $\hat{\sigma_i}$ and switch $refParent(\hat{\sigma_i})$ so that $\hat{\sigma_i}$ becomes the non-preferred child. Then we switch $\hat{\sigma_i}$ so that its only child is a non-preferred child, and $\hat{\sigma_i}$ becomes the only element in its splay tree. Let c be the (only) child of $\hat{\sigma_i}$ in the multi-splay tree. We proceed to rotate c so $\hat{\sigma_i}$ becomes a leaf, and remove $\hat{\sigma_i}$. (We have to rotate c before we delete $\hat{\sigma_i}$ because our dynamic BST model does not allow contraction. We only allow deletion at the leaf.) Due to the removal of $\hat{\sigma_i}$, for each node v in $refSubtree(\hat{\sigma_i})$, $refDepth(v)$ is reduced by one. Note that the set of nodes in $refSubtree(\hat{\sigma_i})$ are identical to $subtree(c)$. So we only need to update the fields of c to reflect all that changes in $refDepth$ and $minRefDepth$.

Finally, we consider the case in which $\hat{\sigma_i}$ has two children in the reference tree. Before rebalancing the reference tree using amortized $O(1)$ simulated rotations and pointer

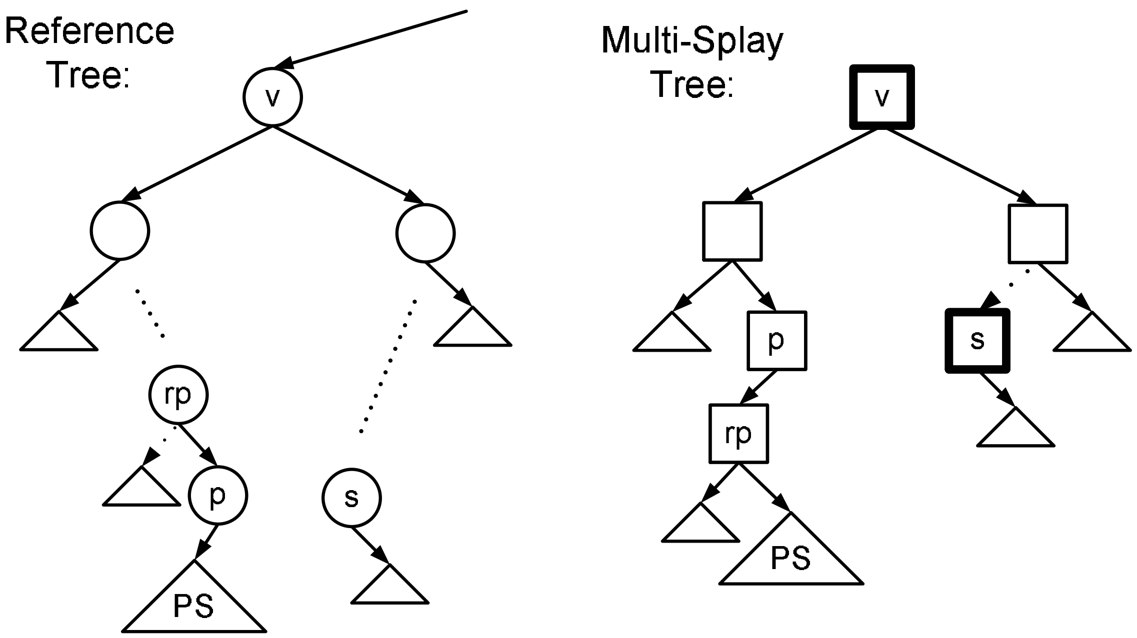

Figure 5.4: An example of what a multi-splay tree looks like during deletion of node $\hat{\sigma}_i$ with two children in the reference tree, where $v = \hat{\sigma}_i$, $p =$ *pred*$(\hat{\sigma}_i)$, $s =$ *succ*$(\hat{\sigma}_i)$, $rp = refParent($*pred*$(\hat{\sigma}_i))$ after the sequence *query*(p), *switch*(rp), *query*(p), *query*(s), and *query*(v). Here we show the case in which $rp < p$.

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 5.6. 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 a multi-splay tree that looks like the one in Figure 5.4.

There are two important aspects of this multi-splay tree. First, *pred*$(\hat{\sigma}_i)$, $\hat{\sigma}_i$, and *succ*$(\hat{\sigma}_i)$ are located close together. Second, *pred*$(\hat{\sigma}_i)$'s subtree is isolated in its own subtree of the multi-splay tree (the subtree PS in Figure 5.4). Thus, after we performed the sequence of queries and switches, we can delete $\hat{\sigma}_i$ as follows. We first swap $\hat{\sigma}_i$ and *pred*$(\hat{\sigma}_i)$, then contract $\hat{\sigma}_i$ to delete it. However, since we do not allow swap and contraction in our dynamic BST model, the operation is implemented by rotating $\hat{\sigma}_i$ to the leaf, removing it, and rotating *pred*$(\hat{\sigma}_i)$ to take $\hat{\sigma}_i$'s place. Because *pred*$(\hat{\sigma}_i)$, $\hat{\sigma}_i$, and *succ*$(\hat{\sigma}_i)$ are located close together, $O(1)$ rotations suffice. Then we change the field of the root of PS to decrement the $refDepth$ and $minRefDepth$ fields of every node in PS by one.

After the deletion, we might need to virtually rebalance the reference tree. In a red-black tree, we only need amortized $O(1)$ pointer traversal and rotations for rebalancing. We can perform each of those virtual traversals and virtual rotations in a constant number of switches as shown in Section 5.3 and Section 5.2.

5.6 Running Time Analysis

5.6.1 Proof of $O(\log n)$ amortized time per operation

Theorem 11. *The amortized cost of insertion, deletion or query on the multi-splay tree is $O(\log n)$, where n is the maximum number of elements exists in the multi-splay tree at any time.*

Proof. In this proof, we will use the multi-splay tree access lemma in Section 3.4.2 with $mass(v) = 1$ for all v. We first account for the cost of each virtual traversal and virtual rotation, then we bound the cost of each query, each insertion and each deletion.

Each virtual traversal and rotation is implemented with a constant number of switches. We apply the multi-splay tree access lemma to show that each switch has an amortized cost of at most $O(\log n)$.[2] In addition, because virtual rotation of v over p changes $\Diamond(v)$ and $\Diamond(p)$, we need to account for the difference in potential due to the virtual rotation. Since the $mass$ of every node is 1, the w of each node is proportional to its $refSubtree$. In a red-black tree, the size of v and p's $refSubtree$ can only change by a constant factor due to the rotation. Thus, the potential of v and p only changes by a constant after the rotation. Since both p and v's depths are less than or equal to 3 in their splay trees (as shown in Figure 5.2), the total potential changes per virtual rotation (excluding the switches) is bounded by $O(1)$. Hence, the total amortized cost of all the virtual traversal and virtual rotations is at most $O(m \log n)$.

Because we have already accounted for the cost of virtual traversal and rotation, we can assume there are no virtual traversal and rotation while bounding the cost of query, insertion and deletion.

By the multi-splay tree access lemma, the amortized cost of each query is $O(\log n)$.

[2]In the proof of multi-splay tree access lemma in Section 3.4.2, we bound the amortized cost of a query by upper bounding the amortized cost of each switch. Thus, while the amortized cost of a query might be *smaller* than the amortized cost of a switch, our *upper bound* on the amortized cost of a query is always *larger* than the amortized cost of a switch.

When we insert an element at the leaf, there are at most $O(\log n)$ elements in the inserted splay tree. The potential of each node in that splay tree can increase by at most 1, and the potential of the inserted node is at most $O(\log n)$. Thus, an insertion can only increases the potential by at most $O(\log n)$. The final query on the inserted element also has an amortized cost of $O(\log n)$, and this final query pays for the time to traverse to the location of insertion. Thus, the amortized cost of an insertion is $O(\log n)$.

When we delete an element v, we first perform a constant number of queries and switches so v, *pred*(v) and *succ*(v) have constant depths in multi-splay tree. By the multi-splay tree access lemma, each query and each switch have an amortized cost of $O(\log n)$. Then we swap v with *pred*(v) (through a constant number of rotations) and delete v. This swap and delete only increase the potential of *pred*(v), the potential for all other nodes decreases. Since v was the root of multi-splay tree, the potential of *pred*(v) is no more than the potential of v before the deletion. Hence, the increase in potential due to *pred*(v) is accounted for by the reduction in potential when v is removed. Thus, the amortized cost of a deletion is also $O(\log n)$. ☐

Corollary 6. *Multi-splay trees satisfy the $O(\log n)$ dynamic runtime property.*

Proof. Since the initial potential is 0, and the final potential is greater than or equal to 0, by Theorem 11, a sequence of m operations uses at most $O(m \log n)$, where n is the maximum number of elements exists in the multi-splay tree at any time. ☐

5.6.2 Proof of $O(\log \log n)$-dynamic-competitiveness

For the purpose of this analysis, we define the potential of a multi-splay tree T as follows. If each node v has an arbitrary positive *weight* $w(v) = 1$, 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 multi-splay tree to be $\sum_{v \in T} \lg s(v)$. In other words, the weight of each node in each splay tree is 1, and the potential of the multi-splay tree is the sum of the potentials of the splay trees.

Because we used the same potential function to prove that multi-splay tree is $O(\log \log n)$-competitive, by the analysis in Theorem 6, the amortized cost for each switch is $O(\log \log n)$.

Theorem 12. *For an arbitrary sequence $\sigma = \sigma_1 \cdots \sigma_m$ in a multi-splay tree, the cost of σ is $O(DOPT(\sigma) * \log \log n)$, where n is the maximum number of elements in the multi-splay tree.*

Proof. For each query, the number of switches is exactly the increase in the Dynamic Interleave Bound plus $O(1)$. Each switch costs amortized $O(\log \log n)$ in the multi-splay tree.

For each *insert*($\hat{\sigma_i}$), the number of switches caused by queries, which each cost amortized $O(\log \log n)$, performed during the insert's query is equal to the increase in the Dynamic Interleave Bound. After we accounts for the switches from queries, there are only $O(1)$ switches left, and each switch cost $O(\log \log n)$. (The $O(1)$ switches consist of the extra switches on $\hat{\sigma_i}$ and $refParent(\hat{\sigma_i})$ and the amortized $O(1)$ switches from virtual rebalancing.[3]) We charge these unaccounted $O(\log \log n)$ amortized cost 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 $refParent(\textbf{\textit{pred}}(\hat{\sigma_i}))$). The constant number of extra switches and the rest of the additional $O(\log \log n)$ amortized cost (consisting of the field updates, the virtual traversals, the virtual rotations, the extra rotations to move the deleted element to the leaf, 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 multi-splay trees are $O(\log \log n)$-competitive.

□

5.6.3 Proof of 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) on a multi-splay tree T with reference tree P, where P is a red-black tree. Because of the similarity between push and inject, and between pop and eject, this description will focus only on push and pop.

To perform $push(x)$, we first do a standard BST insert into T, and then query x in T (we do not need the fields of x to perform this operation). Next, we virtually insert x into P by setting the $refDepth$ and $minRefDepth$. Nodes on the current access path of x may

[3]Because our weight assignments do not depend on the non-preferred subtree, and the set of nodes in each splay tree is identical before and after the rotation, virtual rotation does *not* change the potential.

have their $minRefDepth$ values affected, but there are only a constant number of these, so updating these takes only constant time. Finally, P is virtually rebalanced, which includes performing a series of virtual pointer traversals followed by a constant number of virtual rotations and a switch on x.

During a pop, we query the smallest element x of T, then the successor of x, then we virtually delete x, which is now a leaf of T, from T and from P, which includes performing a constant number of field updates and performing virtual rebalancing similarly to what is done in push. Thus, we have the following invariant.

Invariant 2. *After a push or pop (inject or eject), the left (right) path of P is a preferred path, and the right (left) path of P, excluding $refRoot$, is a preferred path.*

The following property of red-black trees will be useful in our proof.

Lemma 24. *During any sequence of m rebalancing operations following an insertion or a deletion at a node 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]. □

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 on 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 each node on the outer shell with the exception of the root, which is given weight 1. All other nodes are assigned weight 0. The size of node x, denoted by $s(x)$ is defined as usual as the sum of the weights of nodes in x's subtree in T (ignoring the root markings that partition T). The rank of node x, denoted by $r(x)$ and defined only for nodes on the outer shell of P, is equal to $\lg s(x)$. The potential of a multi-splay tree is the sum of the ranks of nodes on the outer shell of P. Because we will always be performing splay steps on nodes with strictly positive weight during a sequence of deque operations with using this weighting scheme, so we can apply the Reweighting Lemma for splay trees [Geo04] to all splays or partial splays on nodes of the outer shell.

Notice that we will need to change the weights of some nodes as their black heights change so as to conform to our weighting scheme. This is why we use the Reweighting Lemma instead of the Access Lemma. Also, we will need to assign and remove weights to and from nodes as they enter and leave the outer shell of P.

Invariant 2 implies that the outer shell of P forms a connected component in T and all children of leaves of this component are marked as roots, which implies that splay steps executed on nodes outside the outer shell of the multi-splay tree will not change the potential of T so their costs can be accounted for separately. However, whenever a node enters or leaves the outer shell of P due to a virtual rotation in P, we need to account for the change in potential.

Lemma 25. *In a sequence of deque operations, the amortized cost of a switch at height h in P is $O(h)$ and each switch of $\mathit{refRoot}$ costs $O(1)$.*

Proof. Only nodes on the outer shell of P, or children of the nodes of the outer shell are switched during a sequence of deque operations. When we switch a node x at height h not on the outer shell of P, the number of nodes in x's splay tree is at most h by Invariant 2. Thus, the *worst-case* cost of such a switch is $O(h)$. Moreover, since x is not in a splay tree that contains any nodes on the outer shell, such switch does not change the potential of T.

A switch of a node x at height h on the outer shell consists of two splay operations in T (on x and $\mathit{refParent}(x)$) in addition to a constant number of field updates. 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.) Finally, the amortized cost of switching $\mathit{refRoot}$ is $O(1)$ because switching $\mathit{refRoot}$ consists only of performing a constant amount of field updates in addition to one splay of $\mathit{refRoot}$, whose weight is 1, a constant fraction of the total weight of T. $\square$

Lemma 26. *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 a reweighting due to this operation is $O(\mathit{refHeight}(x))$.*

Proof. The black height of a node x in a red-black tree 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' black height can change because only a constant number of nodes of any particular black height are touched during rebalancing.

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

For each virtual rotation in P, a node may join or leave the outer shell of P. When a node leaves the outer shell, the potential of T decreases when its weight is removed. On the other hand, when a node x joins outer shell at height h in P, x has at most 2 ancestors in

T because the virtual rotation that placed x in the outer shell switched x and $\mathit{refParent}(x)$, which are in the same splay tree as $\mathit{refRoot}$. Thus, updating all of the necessary fields has a worst-case cost of $O(1)$. Additionally, x's newly added weight only increases the potential of a constant number of nodes. Finally, notice that for each ancestor a of x, it is the case that $\mathit{refHeight}(a) \leq \mathit{refHeight}(x) + 1$. Therefore, the increase in rank at each ancestor is bounded naïvely by $\lg 3$ so that the total increase in potential caused by a virtual rotation of x onto the outer shell of P is $O(1)$. ☐

Theorem 13 (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. In addition, to the work to perform the actual insertion or deletion, which can be paid for by the query executed during a deque operation, the cost of each operation can be broken down into two parts: the cost of the initial query that is performed, the cost of virtual rebalancing, and the cost of reweighting the nodes of T as a result of virtual rebalancing.

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 25.

During virtual rebalancing, the nodes that we virtually traverse and rotate in P are the same as nodes we would actually traverse in P if it existed, and each such virtual traversal or operation consists of, in addition a constant amount of bookkeeping, a constant number of switches to nodes whose black heights are at most one larger than the highest node touched during that particular node or rotation. This, along with Lemma 25, shows that the amortized cost of traversing or rotating a node x is $O(bh(x))$. Similarly, by Lemma 26 the cost of reweighting due to a virtual traversal or rotation of node x is $O(bh(x))$. Thus, by Lemma 24, the total cost of switches due to virtual rebalancing 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. ☐

Intuitively, it is easier to argue that multi-splay trees support efficient deque operations than to argue 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 $\mathit{refRoot}$ 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 insure that restructuring due to pushes and pops does not interfere with restructuring due to injects and ejects.

Chapter 6

Conclusion

In this thesis, we have introduced multi-splay trees, and proved several results demonstrating that multi-splay trees have many desirable properties. First, we proved a close variant of the splay tree access lemma [ST85] for multi-splay trees that is sufficient to show that multi-splay trees have the $O(\log n)$ runtime property, the static finger property, and static optimality. Then, we extended the access lemma by proving the remassing lemma, which is similar to the reweighting lemma for splay trees [Geo04]. The remassing lemma shows that multi-splay trees satisfy the working set property, key-independent optimality, and are competitive to parametrically balanced trees, as defined in [Geo04]. We also proved that multi-splay trees achieve $O(\log \log n)$-competitiveness and we showed 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 runtime and competitiveness bounds for query-only case apply when insertions and deletions are also allowed. Then, we proved that multi-splay trees satisfy the deque property, which is still an open problem for splay trees since it was conjectured in 1985 [Tar85]. While it is easy to construct a BST that trivially satisfies the deque property, no other BST scheme satisfying other useful properties has been proved to have deque property.

6.1 Comparisons between Multi-Splay Tree and Splay Tree

The multi-splaying algorithm is similar to splaying, but differs in a few important ways. Consider modifying the algorithm so that it does not splay the left parent during a left-to-right switch and right parent during a right-to-left switch. 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. The pattern is that starting at an ancestor of v, we splay for a while, stop, then move to an ancestor, then splay for a while, then stop, then move to an ancestor, etc. Finally we switch v so that it moves to the root. These partial splays do not keep multi-splay trees balanced. However, with the additional splays (not on the path between the queried element and the root), multi-splay trees 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. In other words, the root markings allow us to temporarily split and join splay trees. Basically, if we do not expect future access in a subtree L of the splay tree T, we split off L. As a result, when we access elements in T, we do not have to pay for anything in L. But when we need to access L, we pay an extra cost to re-attach L into T. This technique allows us to prove tighter bounds on multi-splay trees. However, it is difficult to apply this technique to splay trees, partly because there are significantly less structure in 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. Proving this would be a major contribution toward proving the dynamic optimality of splay trees.

6.2 Lower Bounds

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. While the static and the dynamic interleave lower bounds are very similar, we do not know if the new dynamic interleave lower bound is tight.

Another open problem regarding the dynamic interleave lower bound is whether the best bound can be computed in polynomial time. If not, another interesting problem is whether it can be approximated to within a constant factor, or some factor that is $o(\log\log n)$.

Lower bounds sometimes lead to new algorithms. Examples of this are the development of new algorithms for binary search trees based on Wilber's first lower bound [Wil89, DHIP04, WDS06]. There is the possibility that our lower bound formulation could be used in this fashion.

6.3 More Open Problems

Returning to the original motivation for this research, the problem of finding a $o(\log\log n)$-competitive on-line BST remains open. Even in the off-line model, the problem of finding an $O(1)$-competitive BST is difficult. The best known off-line constant competitive algorithm use dynamical programming. The algorithm not only requires exponential time to compute what rotations to do, but also provides little insight.

Another problem is devising an on-line comparison-based data structure (that does not necessarily adhere to the BST model) that is within a factor of $o(\log\log n)$ of the optimal off-line BST. For example, Iacono devised a non-BST comparison-based data structure called the *unified structure* that exploits temporal and spatial locality of accesses with better bounds than have been proven for most BSTs [Iac01b], but his data structure is only $O(\log n)$-competitive.[1]

[1] Consider the sequence of n accesses $0, \sqrt{n}, 2\sqrt{n}, \ldots, (\sqrt{n}-1)\sqrt{n}, 0, \sqrt{n}, 2\sqrt{n}, \ldots, (\sqrt{n}-1)\sqrt{n}, \ldots$. The unified structure requires $\Omega(\log n)$ time per access while splay trees require only $O(1)$ time per access. To see that splay trees require only $O(1)$ time per access for this sequence, notice that this first round of $\sqrt{n}$ accesses costs $O(n)$ by the Dynamic Finger Theorem. After the first round, at most $2\sqrt{n}$ nodes remain on the left spine and the nodes $0, \sqrt{n}, 2\sqrt{n}, \ldots, (\sqrt{n}-1)\sqrt{n}$ are all among them. Thus, all following rounds will not touch any nodes that were not on the left spine at the end of first round. Applying the Dynamic Finger Theorem on this smaller tree with at most $2\sqrt{n}$ nodes shows that successive rounds cost only $O(\sqrt{n})$.

Appendix A

Lists of Notations and Symbols

Symbols	
n	the number of nodes in the multi-splay tree
m	the length of the sequence of requests
P	a reference tree
r_i	$\mathit{root}(t_i)$ = the root of the i^{th} splay tree
σ	the sequence of queries (we generalize σ to include insertions and deletions after Chapter 4)
σ_i	the i^{th} operation
$\hat{\sigma}_i$	the key or node of σ_i
T	multi-splay tree
$\lvert t \rvert$	the number of nodes in $subtree(t)$, including $\mathit{root}(t)$
t_i	the splay tree involved in i^{th} switch during an operation
T_i	the state of T when σ_i is executed
y	(usually) the node that the multi-splaying algorithm switches in T
y_i	the i^{th} node switched during an operation
x_i, z_i	the two additional nodes we splay during the i^{th} switch

Table A.1: The symbols that are used throughout the thesis.

Notation	
DOPT(σ)	the minimum cost BST to serve the dynamic sequence σ
IB	static interleave bound
$isRoot(v)$	a bit to store if node v in multi-splay tree is the root of a splay tree
leftChild(v)	the left child of v (this is independent of the preferred children relationships)
leftRefSubtree(v)	the left subtree of v in the reference tree
OPT(σ)	the minimum cost BST to serve the query sequence σ
pred(v)	the largest node smaller than v
$minRefDepth(v)$	the minimum $refDepth$ of all the nodes in $splaySubtree(v)$
$refDepth(v)$	the depth of node v in the reference tree (root has depth 1) in the reference tree
$refRoot$	the root of the reference tree
$refSubtree(v)$	the subtree rooted at v in the reference tree (this tree is the same regardless of the preferred children relationships)
rightChild(v)	the right child of v (this is independent of the preferred children relationships)
rightRefSubtree(v)	the right subtree of v in the reference tree
root(t)	the root of tree/subtree t (either a splay tree, a multi-splay tree, or a reference tree)
$splayRoot$	the root of the multi-splay tree
$splaySubtree(v)$	the subtree rooted at v in multi-splay tree restricted to v's splay tree
$subtree(v)$	all the descendants of v (this is independent of the preferred children relationships)
succ(v)	the smallest node larger than v in t
switch(v)	swaps which child is the preferred one in reference tree; (details on how to simulate a switch in multi-splay trees are in Section 3.2)
$s(v)$	size of v = $\sum_{u \in splaySubtree(v)} w(u)$
$w(v)$	weight of v (usually in a splay tree)

Table A.2: The notation used throughout the thesis.

Terminology	
BST	binary search tree
dashed edge	an edge that connects different splay trees
preferred child	the child that is more recently touched if we were to perform all the operations on reference tree
preferred path	a path formed by a maximal chain of preferred child relations in the reference tree. Specifically, if a node v is in a preferred path, then v's preferred child is also in the preferred path.
multi-splay	an algorithm (defined in Section 3.2 that moves a node to the root in multi-splay tree using a series of switches
solid edge	the edges inside a single splay tree

Table A.3: The terminology used throughout the thesis.

Notation for the Multi-Splay Tree Access Lemma and Remass Lemma	
$\Diamond(x)$	$\mathit{lip}(x) \cup \mathit{rip}(x) \cup \{x\}$
Φ_i	the potential of the multi-splay tree after the i^{th} access
$\mathit{lip}(x)$	the set of nodes in x's left inner path in the reference tree (i.e. the set of nodes reachable starting at x's left child and following right child pointers)
$\mathit{mass}(x)$	mass of x (analogous to the weight of a node in splay tree)
$\mathit{refLeftParent}(x)$	the predecessor of x in the proper ancestors of x in the reference tree
$\mathit{refRightParent}(x)$	the successor of x in the proper ancestors of x in the reference tree
$\mathit{rip}(x)$	the set of nodes in x's right inner path in the reference tree (i.e. the set of nodes reachable starting at x's right child and following left child pointers)
$\mathit{refPath}(x)$	the set of nodes in x's preferred path that are at least as deep as x
$\mathit{uchild}(x)$	the non-preferred child of x in the reference tree
$U(x)$	$\mathit{refSubtree}(\mathit{uchild}(x))$
$w(x)$	$\max_{y \in \Diamond(x)} \hat{w}(y)$
$\hat{w}(x)$	$\mathit{mass}(x) + \sum_{y \in U(x)} \mathit{mass}(y)$

Table A.4: The notation used in Section 3.4.2 and 3.4.3.

Notation for the Scanning Theorem	
$A(v)$	the size of the *right ascending path* of v
right ascending path of v	the set of nodes u, such that *rightParent*$^*(v) = u$
rightParent of a node v	p if p's left child is v
t_L	the left subtree of tree t
t_R	the right subtree of tree t
T_{RB}	a red black tree
T_S	a splay tree

Table A.5: The notation used in Section 3.4.4.

Notations for Dynamic Interleave Lower Bound	
DIB	new dynamic interleave bound
DIB_i	the number of switches that must be made in P_i (which is implicit from ρ)
L_i^y	*leftRefSubtree*$(v) \cup \{y\}$ during the execution of σ_i
P_i	the state of P after σ_i is executed
ρ	a sequences of changes to the reference tree P
ρ_i	the i^{th} change to the reference tree P
R_i^y	*rightRefSubtree*(y) during the execution of σ_i

Table A.6: The notation used for Section 4.3.

Appendix B

Table of Constants

Constant Name	Symbol	Definition	Value
multiplicative splay	c_s		$= 3$
addictive splay	c_{sa}		$= 1$
multiplicative reweight	c_r	$= c_s + 1$	$= 4$
multiplicative telescope	c_t		$= 3$
multiplicative switch	c_{sw}	$= 3c_s + 3c_r$	$= 21$
additive switch	c_{swa}	$= 3c_{sa} + 2\lg(c_t + 1)$	$= 7$
multiplicative final switch	c_f	$= c_{sw} + c_s c_{sw} \lg c_t + c_s c_{swa}$	$= 42 + 63\lg 3$
addictive final switch	c_{fa}	$= c_{swa} + c_{sa} c_{sw} \lg c_t + c_{sa} c_{swa}$	$= 14 + 21\lg 3$
multiplicative multi-splay	c_{ms}	$= c_{sw} + c_f$	$= 63 + 63\lg 3$
additive multi-splay	c_{msa}	$= (c_{sw} + c_f)\lg c_t + c_{fa} + c_{swa}$	≈ 312.4

Table B.1: Table of constants.

Bibliography

[Abr63] N. Abramson. *Information Theory and Coding*. McGraw-Hill, New York, 1963. 2.4

[AM78] Brian Allen and Ian Munro. Self-organizing binary search trees. *J. ACM*, 25(4):526–535, 1978. 2.4

[AVL62] G.M. Adel'son-Vel'ski and E.M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1263, 1962. 2.4

[AW98] Susanne Albers and Jeffery Westbrook. Self-organizing data structures. In *Developments from a June 1996 seminar on Online algorithms*, pages 13–51, London, UK, 1998. Springer-Verlag. 1, 2.4

[BD04] Mihai Bădoiu and Erik D. Demaine. A simplified and dynamic unified structure. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN 2004)*, volume 2976 of *Lecture Notes in Computer Science*, pages 466–473, Buenos Aires, Argentina, April 5–8 2004. 2.4

[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. 2, 2.4

[BM80] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight balanced trees. *Theoretical Computer Science*, 11:303–320, 1980. 2.4

[BMW03] G. Blelloch, B. Maggs, and M. Woo. Space-efficient finger search on degree-balanced search trees, 2003. 2.4

[Bro98] Gerth S. Brodal. Finger search trees with constant insertion time. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete* *algorithms*, pages 540–549, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. 2.4

[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. 2, 2.4

[BST85] S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14:545–568, 1985. 1, 2.4

[BY76] J.L. Bently and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. 2, 2.4

[CH93] R. Chaudhuri and H. Höft. Splaying a search tree in preorder takes linear time. *SIGACT News*, 24(2):88–93, 1993. 2.4

[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. 1, 2.4

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The Proof. *Siam J. Comput.*, 30:44–85, 2000. 1, 2.4

[Cra72] C.A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Dept. of COmputer Science, Stanford University, 1972. 2.2, 2.3

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. 2

[CW82] K. Culik, II and D. Wood. A note on some tree similarity measures. *Inform. Process. Lett.*, pages 39–42, 1982. 2.2, 2.3

[DHIP04] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic Optimality–Almost. *FOCS*, 2004. 1, 2.2, 2.3, 4, 2.3, 2.4, 3.2, 3.4.1, 4.3, 4.3.1, 20, 22, 4.3.1, 6.2

[DSW05] J. Derryberry, D. D. Sleator, and C. C. Wang. A lower bound framework for binary search trees with rotations. Technical Report CMU-CS-05-187, Carnegie Mellon University, 2005. 4

[Elm04] Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314:459–466, 2004. 3, 1, 2.4

[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. 2, 2.4

[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. 1, 2.4

[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. (document), 1, 3, 1.1.1, 2.4, 4, 5, 10, 3, 3.4.2, 3.4.3, 5, 5.6.3, 6

[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. 1.1.3, 2.2, 2.3, 2.4, 3.5

[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. 2, 2.4

[GRVW95] Dennis Grinberg, Sivaramakrishnan Rajagopalan, Ramarathnam Venkatesan, and Victor K. Wei. Splay trees for data compression. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 522–530, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics. 2.4

[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. 3.4.4

[GW77] A. M. Garsia and M. L. Wachs. A new algorithm for minimal binary search trees. *SIAM Journal on Computing*, 6:622–642, 1977. 1, 2.4

[Har80] D. Harel. Fast updates of balanced search trees with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980. 2, 2.4

[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. 1, 2.4

[HL79] D. Harel and G.S. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979. 2, 2.4

[HM82] S. Huddleston and K. Mehlhorm. A new data structure fore representing sorted lists. *Acta Informatica*, 17:157–184, 1982. 2, 2.4, 5.6.3

[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. 1, 2.4

[Hu82] T. C. Hu. *Combinatorial Algorithms*. Addison-Wesley, Reading, MA, 1982. 1, 2.4

[Iac00] John Iacono. New upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory (LNCS 1851)*, pages 32–45, 2000. 6

[Iac01a] J. Iacono. *Distribution Sensitive Data Structures*. PhD thesis, Rutgers, The State University of New Jersey, Graduate School, New Brunswick, 2001. 2.4

[Iac01b] 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. 2, 2.4, 6.3

[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. 1, 2.4, 3, 4, 3.4.2

[Jon88] D. W. Jones. Application of splay trees to data compression. *Commun. ACM*, 31(8):996–1007, 1988. 2.4

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971. 1, 2.4

[Knu73] D. E. Knuth. *The art of computer programming, vol. 3: Sorting and searching*. Addison-Wesley, Reading, MA, 1973. 1

[Kor81] J. F. Korsch. Greedy binary search trees are nearly optimal. *Inform. Proc. Letters*, 13:16–19, 1981. 1, 2.4

[KV81] H. P. Kriegel and V. K. Vaishnavi. Weighted multidimensional b-trees used as nearly optimal dynamic dictionaries. *Mathematical Foundations of Computer Science*, 1981. 1, 2.4

[LP89] F. Luccio and L. Palgi. On the upper bound on the rotation distance of binary trees. *Inf. Process. Lett.*, 31(2):57–60, 1989. 2.2, 2.3

[Luc88] J. M. Lucas. Arbitrary splitting in splay trees. Technical Report DCS-TR-234, Rutgers University, 1988. 4.4

[Mäk88] Erkki Mäkinen. On the rotation distance of binary trees. *Inf. Process. Lett.*, 26(5):271–272, 1988. 2.2, 2.3

[Meh75] K. Mehlhorn. Nearly optimal binary search trees. *Acta Inform.*, 5:287–295, 1975. 1, 2.4

[Meh79] K. Mehlhorn. Dynamic binary search. *SIAM Journal on Computing*, 8:175–198, 1979. 1, 2.4

[NR73] J. Nievergelt and E.M. Reingold. Binary search trees of bounded balanced. *SIAM J. on Computing*, 2:33–43, 1973. 2.4

[Oli82] H. J. Olivié. A new class of balanced search trees: half balanced search trees. *Theoret. Inform. Appl.*, 16:51–71, 1982. 2.4

[Ove83] M. H. Overmars. The design of dynamic data structures. In *Lecture Notes in Computer Science*, volume 156. Springer-Verlag, Heidelberg, 1983. 2.4

[Pug89] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Dept. of Computer Science, University of Maryland, 1989. 2, 2.4

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. 2, 2.4

[RB72] E. McCreight R. Bayer. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972. 2.4

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. 2, 2.4

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985. (document), 1, 1.1.1, 1, 1.1.3, 2.1, 2.4, 1, 2, 12, 13, 3.4.2, 1, 3.4.2, 2, 3.4.2, 5.3, 6

[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. 2.2, 2.3

[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. 3, 1, 2.4, 4.4

[Sun89b] R. Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. Technical Report 427, Courant Institue, New York University, 1989. 4.4

[Sun92] R. Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12:95–124, 1992. 3, 1, 2.4, 4.4

[SW04] D. D. Sleator and C. C. Wang. Dynamic optimality and multi-splay trees. Technical Report CMU-CS-04-171, Carnegie Mellon University, 2004. 2.2, 2.3, 2.4

[Tar83] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983. 4.3, 5.1

[Tar85] R. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985. (document), 1, 3, 2.4, 4.4, 6

[Tsa86] Athanasios K Tsakalidis. Avl-trees for localized search. *Inf. Control*, 67(1-3):173–194, 1986. 2, 2.4

[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. 2, 2.4

[Unt79] K. Unterauer. Dynamic weighted binary search trees. *Acta Inform.*, 11:341–362, 1979. 1, 2.4

[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. 2.2, 2.3, 2.4, 1, 6.2

[Wil89] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal of Computing*, 18(1):56–67, 1989. 2.1, 2.3, 4, 2.4, 4.3, 4.3.1, 6.2

Index

multi-splay tree, 17–41
  data structure, 17–20
    recursive, 20
  deletion, 58–62
  drawing, 18–20
  dynamic, 51–66
  insertion, 56–58, 60, 62
  multi-splay, 20–26
  node fields, 17, 40–41, 54
  property, *see* multi-splay tree property
  reference tree
    child, 52
    parent, 53
    rotation, 54–56, 60
    traversal, 52–53, 60
  running time, *see* multi-splay tree property
    worst case, 26
multi-splay tree property, 26–40, 60–66
  $O(\log\log n)$-competitive, 26–28
  $O(\log\log n)$-dynamic-competitiveness, 61–62
  $O(\log n)$ dynamic runtime, 60–61
  access, 28–35
  deque, 62–66
  reweight, 35–37
  scanning, 37–40

binary search tree, *see* BST
BST, 7–15
  algorithm, 7
  competitive analysis, 8
  dynamic, 43–49
    competitive analysis, 44
    lower bound, 44–48
    model, 43
    property, *see* BST property
  lower bound, 8
  model, 7
  property, *see* BST property
BST property, 9–12
  $O(1)$-distribution-competitive, 10, 13
  $O(\log\log n)$-competitive, 12
  $O(\log\log n)$-dynamic-competitive, 49
  $O(\log n)$ dynamic runtime, 48
  $O(\log n)$ runtime, 9, 13
  T-competitive, 8
  access, 11, 13
  competitive to parametrically balanced trees, 12–14
  deque, 48
  dynamic finger, 11, 14, 15
  implication, 13–16
  key-independent, 11, 13
  output restricted deque, 49
  reweight, 11, 13
  reweight property, 13
  scanning, 12, 15
  static finger, 9, 13, 14
  static optimality, 10, 13
  traversal, 12, 15
  working set, 10, 13

chain splay, 5, 40
constants, 75
  c_{fa}, 33, 34, 75
  c_f, 33, 34, 75
  c_{msa}, 29, 33, 34, 37, 75
  c_{ms}, 29, 33, 34, 37, 75
  c_r, 3, 32, 33, 36, 37, 75
  c_{sa}, 3–5, 32, 33, 75
  c_{swa}, 32–34, 75
  c_{sw}, 32–34, 75
  c_s, 3–5, 32, 33, 75
  c_t, 30–34, 75

dynamic interleave lower bound, 44–48

interleave lower bound, 8
  dynamic, 44–48

splay tree, 2–5
  access lemma, 3
  generalized access lemma, 3
  reweighting lemma, 3
  touch lemma, 38
switch, 9, 20–24
  direction, 24
  missing children, 24, 25
  missing parent, 25, 27
  worst-case, 37

tango, 2, 5