NEW PATHS FROM SPLAY TO DYNAMIC OPTIMALITY

CALEB CARSON LEVY

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE PROGRAM IN
APPLIED AND COMPUTATIONAL MATHEMATICS
ADVISER: PROFESSOR ROBERT TARJAN

JUNE 2019
Abstract

Consider the task of performing a sequence of searches in a binary search tree. After each search, an algorithm is allowed to arbitrarily restructure the tree, at a cost proportional to the amount of restructuring performed. The cost of an execution is the sum of the time spent searching and the time spent optimizing those searches with restructuring operations. This notion was introduced by Sleator and Tarjan in 1985 [54], along with an algorithm and a conjecture. The algorithm, Splay, is an elegant procedure for performing adjustments while moving searched items to the top of the tree. The conjecture, called dynamic optimality, is that the cost of splaying is always within a constant factor of the optimal algorithm for performing searches. The conjecture stands to this day. In this work, we attempt to lay the foundations for a proof of the dynamic optimality conjecture.

Central to our methods are simulation embeddings and approximate monotonicity. A simulation embedding maps each execution to a list of keys that induces a target algorithm to simulate the execution. Approximately monotone algorithms are those whose cost does not increase by more than a constant factor when keys are removed from the list. Approximately monotone algorithms with simulation embeddings are dynamically optimal. Building on these concepts, we present the following results:

- In Chapter 3, we build a simulation embedding for Splay by inducing Splay to perform arbitrary subtree transformations. Thus, if Splay is approximately monotone then it is dynamically optimal. We also show that approximate monotonicity is a necessary condition for dynamic optimality.

- In Chapter 4, we show that if Splay is dynamically optimal, then with respect to optimal costs, its additive overhead is at most linear in the sum of initial tree size and number of requests.
• In Chapter 5, we prove that a known lower bound on optimal execution cost by Wilber [64] is approximately monotone.

• In Chapter 6, we speculate about how one might establish dynamic optimality by adapting the proof of approximate monotonicity from the lower bound to Splay.

• In Chapter 7, we show that the related traversal [54] and deque [61] conjectures also follow if Splay is approximately monotone, and demonstrate that most of the results in this paper extend to a broader class of “path-based” algorithms.

• Finally, the appendices offer a “grab-bag” collection of results related to this problem. Appendix A generalizes the tree transformations used to build simulation embeddings; Appendix B offers new results about the behavior of Splay on “pattern-avoiding” permutations; Appendix C includes proofs of selected pieces of “folklore” that have appeared throughout the literature.
Acknowledgements

First and foremost, I thank Bob Tarjan for all of his help, support, and advice over the past four years. It has been a joy to work with him. Bob helped me build my career, waited patiently while I found my way in life, and helped me pull off more crazy shenanigans than any reasonable graduate student should expect to get away with. He is the greatest advisor and mentor I ever could have asked for, and a true friend.

I owe my success to Meng Wang, for his constant reassurance, encouragement, and the occasional well-placed kick to the butt to get done those things that are required to succeed in a Ph.D. On countless occasions, he led me away from the path of foolishness and toward the path of wisdom. I could not have done this without him.

I thank Allan and Wendy Levy for providing me with a life of opportunity, and for instilling in me the self-esteem to use, the work ethic to keep, and the perspective to appreciate those opportunities. They are the most accepting, loving, and supportive parents I know, and by some miracle they happen to be mine. I would not be surrounded by such great people and doing work that I love without the foundations that they raised me on.

I’ve made many wonderful friends at Princeton. I especially want to call out Amit Halevi, Daniel Cooney, Levon Avanesyan, Mark Goresky, Julia Martini and Philippe von Wurstemberger for helping with everything from navigating job markets to cheering me up during hard moments, and for informing, if needed, whenever I had absolutely no clue what I was talking about. Above all, I will remember many wonderful conversations, laughs and lame puns, hanging out, and having a great time. Special thanks to Julia for introducing me to Berlin and Philippe for showing me around Zurich.
I thank our department secretaries Tina Dwyer, Gina Holland, Mitra Kelly, Audrey Mainzer and Nicki Mahler. They have gone out their way to make my stay at Princeton as pleasant and productive as possible, guided me through Princeton’s bureaucracy, and given me tips for everything from good housing and veterinarians to resources for subletting apartments. I deeply appreciate their help.

Huge thanks to Chun-Nam Yu and Iraj Saniee for a wonderful summer experience at Bell Labs.

I had the opportunity to spend a good portion of my Ph.D. working at Intertrust Technologies. I thank Adam Beveridge, Prisdha Dharma, Prasanna Gajbhiye, Bill Horne, David Maher, Jeff McDow, Steve Mitchell, Jarl Nilsson, Vanishree Rao, Talal Shamoon, Nagendra Siravara, Mark Stein, Eric Swenson and Bob Tarjan for being part of that experience.

I thank Joe Hellerstein for providing the chance to embark on certain misadventures towards the end of my Ph.D. related to career development, database theory and distributed logic programming.

Of course, a Ph.D. requires actual research as well as personal growth. Here again, I am grateful to Bob Tarjan for countless critiques, suggestions, and insights. Bob taught me how to do great research, and more importantly, how to do great work. I spent many hours in front of the whiteboard with Bob, the majority of which comprised Bob pointing out flaws and mistakes in my proofs and ideas. These occasions will be some of the most fondly remembered from my time in graduate school. If I have any regrets about my time working with Bob, it would be that I did not get to spend even more time doing research with him.

I thank Siddhartha Sen for inviting me to Microsoft Research to learn a little bit about systems programming, and for extensive feedback and suggestions on my thesis work. I thank Lázló Kozma, John Iacono, Ian Munro, Thatchaphol Saranurak and Daniel Sleator for wonderful discussions about both Splay trees and data structures
generally. A special thanks to Kurt Mehlhorn for extensive suggestions and feedback on this work, and for serving as a reader for this thesis. I thank Bernard Chazelle and Robert Sedgewick for serving on my defense committee.

I spent many hours talking with Daniel Cooney about my work on Splay trees. Dan’s work is primarily in stochastic differential equations and evolutionary game theory. Our research areas are quite different, so communicating our ideas required talking about them at a higher level. I think a large portion of this work, and its presentation, benefitted from these discussions. I can only hope I managed to return the favor.

Finally, I must acknowledge the help of three cats throughout this process: Porkloaf, Agent Ginger, and Agent 26. They have been the perfect research assistants. They have never been judgmental, and they show their love (and hunger) every time I see them. They have consistently ensured that I do not wind up going in circles: if I spend too much contiguous time at my desk, they will eventually just sit on my research notebooks in order to demand attention. Porkloaf passed away during the second year of my Ph.D. after fourteen years of companionship. He is dearly missed by me and my family.

With much enthusiasm, I look forward to beginning my postdoctoral research at UC Santa Cruz with Seshadhri Comandur upon graduation.
In memory of my Grandma, Beatrice Kellerman.
Contents

Abstract ................................................................. iii
Acknowledgements ......................................................... v
List of Figures .............................................................. xii

1 Notes for Readers ......................................................... 1

2 The Dynamic Optimality Conjecture ...................................... 3
  2.1 Binary Search Trees ................................................ 4
  2.2 Rotation ............................................................ 6
  2.3 Splay .............................................................. 7
  2.4 Transition Tree Execution Model .................................... 9
  2.5 Dynamic Optimality ................................................ 13
  2.6 What is Known ..................................................... 14
  2.7 What is not Known ................................................ 16

3 Simulation Embeddings ................................................ 18
  3.1 Tree Transformations ................................................ 20
  3.2 Monotonicity Implies Optimality ................................... 25
  3.3 Optimality Requires Monotonicity .................................. 27
  3.4 Partial Proof of Monotonicity ....................................... 28
  3.5 Competitive Ratios .................................................. 31
  3.6 Related Work ....................................................... 32
4 Eliminating Additive Overhead 35
  4.1 Amplifying Transients 36
  4.2 Background 39

5 Wilber’s Crossing Lower Bound 41
  5.1 Treaps and Move-to-Root 41
  5.2 Crossing Cost and Bookkeeping 43
  5.3 Wilber’s Bound is Approximately Monotone 45

6 Proposal for Proving Optimality 48
  6.1 Move-to-Root and Splay 49
  6.2 Splay’s Crossing Cost 51
  6.3 Potentials for Heap-Order Violations 54
  6.4 Splay’s Bookkeeping Cost 55
  6.5 Similar Ideas 57

7 Extensions and Comments 59
  7.1 Insertion and Deletion 59
  7.2 The Deque Conjecture 60
  7.3 The Traversal Conjecture 62
  7.4 Path-Based Algorithms 63
  7.5 The Power of Simplicity 64

Concluding Remarks 66

A Generalized Transformations 67
  A.1 Universal Transformations 67
  A.2 Simultaneous Transformations 69
  A.3 Top-Down Splay 71
List of Figures

2.1 Rotation at node $x$ with parent $y$, and reversing the effect by rotating at $y$. Triangles denote subtrees. ................................. 6

2.2 A splaying step at node $x$. Symmetric variants not shown. .......... 7

2.3 Example of a subtree transformation of cost four. ...................... 10

2.4 Example of an execution for an instance $X = (1, 2, 6)$. The total cost is $|Q'_1| + |Q'_2| + |Q'_3| = 2 + 4 + 2 = 8$. ......................... 11

3.1 Applying restricted rotations to unwind a binary search tree $T$ into a flat tree $F$, as per the algorithm described in [41]. Repeatedly rotate an off-spine node to the spine, moving the spine appropriately. .... 21

3.2 A Hamiltonian cycle in $G_4$. Splay at nodes marked by $*$ to convert one tree into the next. ................................................. 23

3.3 Example of eliding three transition trees. ............................... 28

5.1 Execution of $X = (8, 2, 7, 4)$ starting from $T$ by Move-to-Root. There are 14 total crossing nodes. Hence, $Λ(X, T) = 14$. ............ 45

6.1 A global view of Splay trees. The transformation from the left to the middle illustrates Move-to-Root. The transformation from the left to the right illustrates Splay trees. (Figure and caption from [7]). ...... 50

6.2 Comparison of crossing nodes for Splay and Move-to-Root when $X = (8, 2, 7, 4)$................................................................. 51
6.3 The access paths for \( x \) and \( y \) contain both crossing nodes and bookkeeping nodes. Every splay step when splaying \( x \) is a zig-zig, and every splay step (except for the last) when splaying \( y \) is a zig-zag.

7.1 Schematic of a binary search tree representation for a deque, showing how the operations defined by [61] map to rotations in the tree. A push or inject makes the rotated nodes visible. A pop initiates a rebalance of this particular tree.

A.1 Splaying the sequence \( G(1, 3, 5) \) cleans up the path no matter what its state was when the splaying started.

A.2 Schematic of the top portion of the augmented trees \( \hat{T} \). To induce a rotation at \( u \), perform top-down splays at \( (a, u, a, z) \). To induce a rotation at \( v \), top-down splay the sequence \( (v, a, z) \). To induce a rotation at \( w \), top-down splay at \( (a, w, a, z) \). To induce a rotation at \( y \), top-down splay at \( (y, a, z) \).

B.1 Possible locations for the next sub-root \( x \) to be insertion splayed in \( \pi \setminus (2, 3, 1) \). The case on the left occurs when the next splayed node has left-depth 0, and the case on the right occurs when it has left-depth 1. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.

B.2 Possible locations for the next sub-root \( x \) to be insertion splayed in \( \pi \setminus (3, 1, 2) \). The case on the left occurs when the next splayed node is less than the root, and the case on the right occurs when the next sub-root is the new tree maximum. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.
C.1 Configuration of crossing and inside nodes in the proof of Theorem C.3.  

The node pairs $s_i$ and $t_i$, and $t_{i+1}$ and $x$, may or may not be distinct.

D.1 Depiction of the after-tree $S_i$ with zipped subtree $J_i$ and after-tree $T_i$, and unzipped subtree $K_i$. The top tree $I_i$ is identically arranged in both $S_i$ and $T_i$ by Lemma D.1. By Lemma D.3, we need merely compute crossing depths for nodes on the access path for $x$ and their immediate children. Horizontally striped nodes are those for which $A_{i} = 1$. (See Section D.3.) Vertically striped nodes are those for which $B_{i} = 1$. Crosshatched nodes are those for which $A_{i}B_{i} = 1$. For this tree, $E_{i} = 1$ when root$(J_i^+) = v_i$. The nodes $w_{i}^1, w_{i}^2, \ldots$ are the crossing nodes for $x$ in $J_i$.  

\[ \text{xiv} \]
Chapter 1

Notes for Readers

The nature of this thesis is perhaps unusual nowadays. It is not a “sandwich” of every-thing I have done during my Ph.D., but rather, a single, coherent story comprising the most important parts of my academic work. My SODA paper with Bob Tarjan, “A New Path From Splay to Dynamic Optimality,” [37] may essentially be viewed as an “extended abstract” for this thesis.

Chapters 2-7 should read front-to-back, in order. Examiners, and those already reasonably familiar with the conjecture, will be fairly comfortable reading at least one Chapter per day. Those less familiar might read one or two sub-chapters per day.

This thesis is light on background material. The most essential relevant references are indicated where appropriate. Readers who are looking for a comprehensive survey of the topic are directed to [35, Chapters 1 and 2]. The formal material in this thesis is fairly self-contained, and anyone with a basic theoretical understanding of data structures should be able to follow it. Kozma’s thesis will simply provide more context for the uninitiated.

Finally, a comment on the philosophy of this work. Folklore wisdom\(^1\) dictates it is sometimes \textit{easier} to induct on \textit{stronger} hypotheses, because they provide more exploitable structure. Accordingly, this investigation adopts a somewhat different tone.

\(^1\)A good source of which is [49]
from its companions by making no qualms about presuming that splay is constant-competitive and allowing this assumption to guide the intuition. I rather ask: “what is the simplest, most straightforward path to proving dynamic optimality?” I hope you will find my answers to be interesting.
Chapter 2

The Dynamic Optimality Conjecture

By an instance, we mean a list of tasks together with a specified initial configuration. An execution of an instance is a sequence of operations that accomplish the tasks. An algorithm maps an instance to an execution of that instance.

Operations are assigned a cost. The cost of an execution is the sum of the costs of its constituent operations, and the cost of an algorithm to execute an instance is the cost of the execution it produces for that instance. Importantly, certain kinds of instances admit a natural notion of an optimal execution: a collection of operations that accomplish the given instance with the lowest possible total cost.

In this paper, instances consist of an initial tree together with a sequence of keys to access (the tasks) in that tree. The operations are subtree transformations, and the executions are sequences of subtree transformations which bring each accessed key to the root. The main algorithm of interest is Splay [54]. The cost of a subtree transformation is the size of the transformed subtree, and the cost of an execution is the sum of sizes of the transformed subtrees.
It is of course desirable to obtain precise descriptions of optimal executions, but this seems to be exceedingly difficult. However, there is sometimes something almost as good: a constant-competitive algorithm whose execution cost never exceeds a fixed multiple\(^1\) of the optimum cost. The crux of the dynamic optimality conjecture is to determine whether Splay is constant-competitive for all instances comprising searches in a binary search tree.

### 2.1 Binary Search Trees

A binary tree \(T\) contains of a finite set of nodes, with one node designated to be the root. All nodes have a left and a right child pointer, each leading to a different node. Either or both children may be missing; a missing child is denoted by null. Every node in \(T\), save for the root, has a single parent node of which it is a child. (The root has no parent.) The size of \(T\) is the number of nodes it contains, and is denoted \(|T|\).

There is a unique path from root(\(T\)) to every other node \(x\) in \(T\), called the access path for \(x\) in \(T\). If \(x\) is on the access path for \(y\) then \(x\) is an ancestor of \(y\), and \(y\) is a descendent of \(x\). (Every node is an ancestor and a descendent of itself.) The subtree comprising \(x\) and all of its descendants is called the subtree rooted at node \(x\). Nodes thus have left and right subtrees rooted respectively at their left and right children. (Subtrees are empty for null children.) The depth of the node \(x\), denoted \(d_T(x)\), is the number of edges on its access path. Its right-depth is the number of right pointers followed, and its left-depth is the number of left pointers followed.

In a binary search tree, every node has a unique key, and the tree satisfies the symmetric order condition: every node’s key is greater than those in its left subtree and smaller than those in its right subtree. The binary search tree derives its name from how its structure enables finding keys. To find a key \(k\), initialize the current node to be the root. While the current node is not null and does not contain the

\(^{1}\)That is, the same constant applies to all instances.
given key, replace the current node by its left or right child depending on whether \( k \) is smaller or larger than the key in the current node, respectively. The search returns the last current node, which contains \( k \) if \( k \) is in the tree and otherwise \text{null}. The \textit{cost} of this search is set by convention to be the number of nodes, including \text{null}, encountered prior to termination (this is called the \textit{length} of the search path). To keep our presentation simple, we assume that a key and the node containing it can be used interchangeably in binary comparisons. (Binary search trees are \textit{endogenous} data structures \cite{59}.)

The \textit{lowest common ancestor} of \( x \) and \( y \) in \( T \), denoted \( \text{lca}_T(x, y) \), is the deepest node shared by the access paths of both \( x \) and \( y \). Note that \( \min\{x, y\} \leq \text{lca}_T(x, y) \leq \max\{x, y\} \). The \textit{left spine} of \( T \) is the access path to the smallest key in \( T \), and the \textit{right spine} of \( T \) is the access path to the largest key in \( T \). The left and right spines consist entirely of left and right pointers, respectively. If every node in \( T \) is on the left (respectively right) spine then \( T \) is called a \textit{left} (respectively \textit{right}) spine.

To \textit{insert} a new key \( k \) into a binary search tree \( T \), first do a search for \( k \) in \( T \). When the search reaches a missing node, replace this node with a new node containing the key \( k \). (Inserting into an empty tree makes \( k \) the root key.) If \( X = (x_1, \ldots, x_m) \) is a sequence of keys then the \textit{insertion tree for} \( X \), denoted \( \text{BST}(X) \), is the binary search tree obtained by starting from an empty tree and inserting keys in order of their first appearance in \( X \).

The symbol \( \oplus \) denotes sequence concatenation.\footnote{The notation \( \bigoplus_{i=1}^{m} X_i \) denotes the concatenation of the sequence of sequences \( X_1, \ldots, X_m \).} Let \( T \) be a binary search tree of \( n \) nodes with root \( r \), \( L = \text{left-subtree}(r) \), and \( R = \text{right-subtree}(r) \). The \textit{preorder} and \textit{postorder} of \( T \) are defined as follows: the preorder and postorder of the empty tree is the empty sequence, and \( \text{preorder}(T) = (r) \oplus \text{preorder}(L) \oplus \text{preorder}(R) \) and \( \text{postorder}(T) = \text{postorder}(L) \oplus \text{postorder}(R) \oplus (r) \). Such a sequence is referred to as a \textit{preorder} or a \textit{postorder}.
2.2 Rotation

Binary search trees are the canonical data structure for maintaining an ordered set of elements, and are building blocks in countless algorithms. Perhaps the most attractive feature of binary search trees is that the number of comparisons required to find a key in an \( n \)-node binary search tree is \( O(\log n) \), provided that the tree is properly arranged, which is good in theory and practice. However, without exercising care when inserting nodes, a binary search tree can easily become unbalanced (for example when inserting 1, 2, \ldots, n in order), leading to search costs as high as \( \Omega(n) \). Thus, binary search trees require some form of maintenance and restructuring for good performance.

We will employ a local restructuring primitive called rotation. A rotation at left child \( x \) with parent \( y \) makes \( y \) the right child of \( x \) while preserving symmetric order. A rotation at a right child is symmetric, and rotation at the root is undefined. (See Figure 2.1.) A rotation changes three child pointers in the tree.

Rotations were first employed in “balanced” search trees, which include AVL trees [1], red-black trees [26], weight-balanced trees [46], and more recently weak...
AVL trees [27]. These trees augment nodes with bits that provide rough information about how “balanced” each node’s subtree is. After an insertion or deletion, rotations are performed to restore invariants on the balance bits that ensure all search paths have $O(\log n)$ nodes. While balanced searched trees are not the focus of this work, they were progenitors for the main algorithm of interest.

## 2.3 Splay

The Splay algorithm [54] eschews keeping track of balance information, replacing it with an intriguing notion: instead of adjusting the search tree only after insertion and deletion, Splay modifies the tree after every search.

The algorithm begins with a binary search for a key in the tree. Let $x$ be the node returned by this search. If $x$ is not \texttt{null} then the algorithm repeatedly applies a “splay step” until $x$ becomes the root. A splay step applies a certain series of rotations based on the relationship between $x$, its parent, and its grandparent, as follows. If $x$
has no grandparent (i.e. x’s parent is the root), then rotate at x (this case is always terminal). Otherwise, if x is a left child and its parent is a right child, or vice-versa, rotate at x twice. Otherwise, rotate at x’s parent, and then rotate at x. Sleator and Tarjan [54] assigned the respective names zig, zig-zag and zig-zig to these three cases. The series of splay steps that bring x to the root are collectively called to as *splaying at x*, or simply *splaying x*. The three cases are depicted in Figure 2.2.

The cost of splaying a single key x in T is defined to be \(d_T(x) + 1\).\(^3\) If \(X = (x_1, \ldots, x_m)\) is a sequence of requested keys in T then the cost of splaying X starting from T is defined as \(m + \sum_{i=1}^{m} d_{T_{i-1}}(x_i)\), where \(T_0 = T\), and for \(1 \leq i \leq m\), \(T_i\) is the result of splaying \(x_i\) in \(T_{i-1}\). To perform *insertion splaying*, insert a key into the tree and then splay the newly created node. The cost of an insertion splay is the cost of the splay operation.

While an individual splay operation can involve every node in the tree, the total cost of splaying \(m\) requested keys in a tree of size \(n > 0\) is \(O((m + n)\log n)\). Hence, the worst case cost of a splay operation, *amortized* over all the requests, is the same as any balanced binary search tree [54]. This is perhaps surprising for an algorithm that keeps no record of balance information.

What makes Splay truly remarkable is how it takes advantage of “latent structure” in the request sequence, and provides more than simple “worst-case” guarantees. As just one example, if \(t_X(i)\) is the number of different keys seen between the last and current access to \(x_i\) (or since the beginning of the sequence if \(i\) is the first access to \(x_i\)), then the cost to splay X starting from T is \(O(n \log n + \sum_{j=1}^{m} \log(t_X(j) + 1))\).\(^4\) (This is called the “working set” property [54].) Thus, Splay exploits “temporal locality” in the access pattern.

Splay simultaneously exploits “spatial” locality, as shown by the following theorem (originally conjectured in [54]) that we will use later on:

\(^3\)The search cost can be absorbed into the rotations.
\(^4\)Note that \(O(\log n)\) amortized cost per splay is a corollary of this.
Theorem 2.1 (Dynamic Finger [14, 13]). Let the rank of $x$ in $T$, denoted $r_T(x)$, be the number of nodes in $T$ whose keys are less than or equal the key in $x$. The cost of splaying $X = (x_1, \ldots, x_m)$ starting from $T$ is $O(|T| + m + DF_T(X))$, where $DF_T(X) \equiv \sum_{i=2}^{m} \log_2(|r_T(x_i) - r_T(x_{i-1})| + 1)$.

In fact, the properties of Splay inspired the authors of [54] to speculate on a much stronger possibility: that Splay’s cost is always within a constant factor of the “optimal” way of executing the requests in an appropriate model of binary search tree executions.

2.4 Transition Tree Execution Model

The following model for binary search tree executions is based on “transition trees.” It is cost-equivalent (to within constant factors) to other models defined throughout the literature, including the rotation-based model (see Chapter C.1) and the geometric model (see Chapter 2.6). Our definition most closely resembles [35, Second Model].

The fundamental operation is subtree transformation. To perform a subtree transformation on tree $T$, first select an arbitrary connected subtree of $Q$ containing the root of $T$.\footnote{This is distinct from the subtree rooted at a node $x$; $Q$ need not contain all ancestors of the root. The overloading of terminology is unfortunate but traditional in the literature. It will hopefully always be clear from context which sense is being used.} Then reshape $Q$ into any other valid binary search tree $Q'$ containing the same set of keys. (We refer to $Q'$ as a transition tree.) To complete the operation, form the after-tree $T'$ by substituting $Q'$ for $Q$ in $T$, re-attaching the subtrees of $Q$ to $Q'$ in the manner uniquely prescribed by the symmetric order. The cost of this operation is the size (number of nodes) of $Q$. The process is depicted in Figure 2.3.

Definition 2.1 (BST Model). An instance of a binary search tree optimization problem comprises a sequence $X = (x_1, x_2, \ldots, x_m)$ of requested keys and an initial tree $T$ containing these keys.
Figure 2.3: Example of a subtree transformation of cost four.

An *execution* for this instance comprises a sequence of subtrees $Q_1, \ldots, Q_m$, a sequence of transition trees $Q'_1, \ldots, Q'_m$, and a sequence of after-trees $T_1, \ldots, T_m$, where $T_0 \equiv T$ and for $1 \leq i \leq m$:

- $Q_i$ is a connected subtree of $T_{i-1}$ that contains both $\text{root}(T_{i-1})$ and $x_i$;
- $Q'_i$ is a binary search tree with the same keys as $Q_i$ such that $x_i = \text{root}(Q'_i)$;\(^6\)
- $T_i$ is formed by substituting $Q'_i$ for $Q_i$ in $T_{i-1}$.

Each $Q_i$ in an execution is uniquely determined by $T_{i-1}$ and $Q'_i$, and each $T_i$ for $i > 0$ is uniquely determined by $Q'_i$ and $T_{i-1}$. Thus an execution is uniquely determined by the sequence of transition trees, and we shall occasionally denote the execution by this sequence. Figure 2.4 shows an example of an instance and corresponding execution.

The *cost* of execution $E = (Q'_1, \ldots, Q'_m)$ is $\sum_{i=1}^{m} |Q'_i|$, where $|Q'|$ denotes the number of nodes in $Q'$. At least one execution for $X$ starting from $T$ has minimum, or *optimum* value. Define

$$\text{OPT}(X, T) \equiv \min_{E \text{ for } X,T} \sum_{Q' \in E} |Q'|.$$\(^6\)

\(^6\)The requirement that the requested key be placed at the root of its corresponding transition tree is a matter of convenience, and may be dropped if needed in order for the model to accommodate other algorithms (such as Greedy, defined below, or Semi-Splay [54]). Dropping this requirement lowers the optimum cost for a given instance by at most a constant factor. (See C.1.)
Figure 2.4: Example of an execution for an instance $X = (1, 2, 6)$. The total cost is $|Q_1'| + |Q_2'| + |Q_3'| = 2 + 4 + 2 = 8$.

Defining the cost of an execution as the sum of the transition tree sizes captures the notion of paying for restructuring: fewer operations are required to substitute a smaller tree. Each transition tree contains the root and the node containing the requested key, and therefore the path connecting them (i.e. access path). This accounts for the cost of searching. We describe how to extend this model to encompass insertions and deletions in Chapter 7.1.

**Remark 2.1.** We will sometimes abuse notation and let $\text{OPT}(X, T)$ also refer to a sequence of transition trees that achieves the optimum cost. We will also abuse terminology by referring to the optimum execution for $X$ and $T$ instead of an optimum execution. Although there may be many executions achieving optimum cost, it will not matter for our purposes which is chosen, so we may assume one is chosen arbitrarily.

**Remark 2.2.** By [8, Theorem 43], we may assume without loss of generality that every node in the initial tree $T$ has an accessed descendant in $T$.\footnote{Essentially, if a node $y$ without children in $T$ is never accessed then every occurrence of $y$ in a transition tree can be replaced with an access to $y$’s predecessor or successor.} Hence, $T$ is the
smallest connected subtree of the root containing all of $X$’s keys. Since every node in $T$, at least initially, is on a path from the root to a node that will be accessed, even an optimal execution must visit every node in the tree at least once. Therefore, $\text{OPT}(X,T) \geq |T|$. Similarly, every execution produces at least one transition tree per request. Hence, $\text{OPT}(X,T) \geq |X|$, where $|X|$ is the number of requests in $X$.

**Remark 2.3.** Note that $|\text{OPT}(X,T) - \text{OPT}(X,T')| \leq |T|$ for any two binary search trees $T$ and $T'$ with the same set of keys. Thus, it makes little difference to $\text{OPT}$ whether the initial tree is left specified or unspecified, and many authors allow $\text{OPT}$ to choose the best initial tree for a given request sequence for free. However, the initial tree can (potentially) have a significant impact on Splay’s behavior. Thus, we require instances to specify an initial tree. This matter is discussed further in Chapter 4.

**Remark 2.4** (When are two trees the same?). There seem to be at least two subtly distinct notions of when two binary search trees are “equal.” The first view, which we will refer to as “node identity,” is that a tree is fundamentally a set of nodes. While restructuring may change the arrangement of these nodes, any tree comprising the same nodes is the same tree. Thus, $T$ and $T'$ may start out with the same keys in the same arrangement, but if the nodes themselves are different then they are different trees. Conversely, rotating a node in $T$ may change its configuration, but under the view of node identity it remains the same object. This view is perhaps more common (likely due to how trees are actually implemented on computers).

In the second view, which we will refer to as “configuration identity,” a binary search tree is essentially a map from keys to the binary encodings of their access paths.\(^8\) So, for example, rotating at a child of the root of $T$ changes every path in the

---

\(^8\)See Chapter 3.4 for more detail about path encodings.
tree, and the resultant tree $T'$ is an entirely different mapping. Similarly, performing the reverse rotation in $T'$ to form $T''$ means $T = T''$.\footnote{The symbols \textquotedblleft $T$\textquotedblright{} and \textquotedblleft $T''$\textquotedblright{} may be viewed as different names for the same underlying tree, while \textquotedblleft $T'$\textquotedblright{} is a name for the rotated version of $T$.}

The difference is basically semantic (and preference), and primarily influences notation and vocabulary. Node identity seems natural for describing how operations are implemented; configuration identity is suited for describing the outcomes of operations. Definition 2.1 and the remainder of this paper are presented in the language of configuration identity (unless otherwise noted) because it affords a greater level of abstraction and conciseness.

### 2.5 Dynamic Optimality

An algorithm $A$ maps an instance $(X, T)$ to a sequence of valid transition trees $Q'_1, \ldots, Q'_{|X|}$ that execute the request sequence $X$ with starting from tree $T$. The cost of this execution is denoted $\text{cost}_A(X, T)$. Note that we will be dealing almost exclusively with the cost of the Splay algorithm. Hence, $\text{cost}(X, T)$, without subscript, will always refer to the cost of splaying the keys of $X$ with initial tree $T$.

**Definition 2.2.** An algorithm $A$ is said to be dynamically optimal (or constant competitive) if there exists some constant $c \geq 1$ so that $\text{cost}_A(X, T) \leq c \text{OPT}(X, T)$ for all sequences of keys $X$ and all corresponding initial trees $T$. The dynamic optimality conjecture states that Splay is dynamically optimal.

The term "dynamically optimal" instead of "constant-competitive" is traditional in the literature on this problem, and so we adapt it in this paper.
2.6 What is Known

Remark 2.5. This section provides only a cursory overview of the background literature on dynamic optimality. Lázló Kozma provides a more systematic survey in his Ph.D. thesis [35, Chapters 1 and 2].

The most interesting binary search tree algorithms are on-line algorithms; i.e. those which choose their operations based on tasks that come before, but not after, the current one. These reflect the algorithms that tend to be useful in realistic scenarios, and Splay is one of the most famous on-line binary search tree algorithms.

Various authors have explored generalizations of Splay. Subramanian defined a class of algorithms that reshape a tree in small steps. A set of rules, called a “template,” determine which step to take based on the arrangement of nodes in the immediate vicinity of the currently selected node. Different templates give rise to different algorithms, and a number of these algorithms have many of the same properties as Splay [57]. Georgakopoulos and McClurkin [23] and later Chalermsook et al. [7] proved further results about template-based algorithms.

Iacono describes a “proof-of-concept” algorithm in [31] that uses a multiplicative weights update method where each “expert” is a member from a certain class of on-line BST algorithms, ensuring that if any member of this class is optimal then so is his algorithm.

Finally, “Tango trees” [19] are known to cost at most $O(\log \log n)$ times the optimum for an initial tree of size $n$, vs. the best-known $O(\log n)$ for Splay and Greedy (or any balanced-tree algorithm). Tango trees are not dynamically optimal: there exist request sequences on which Tango costs $\Theta(\log \log n)$ times the optimum [19].

Both Iacono’s algorithm and Tango trees are mainly of theoretical interest, due to their complexity and high constant factor overhead.
Geometry and Greedy

The other major candidate on-line optimal binary search tree algorithm is colloquially known as “Greedy.” (Variously, “GreedyFuture” and “Geometric Greedy.”) An off-line version of this algorithm, which min-heap orders nodes on the access path by soonest future access time of the keys, was conjectured to be constant-competitive for binary search trees, first by Lucas [40], and then independently by Munro [45]. The algorithm received renewed attention when Demaine et al. showed that it could be simulated by an on-line algorithm by using a representation of binary search tree executions as cartesian coordinate point-sets [18].

In the geometric BST model, a request sequence of integers $X = (x_1, \ldots, x_m)$ is mapped to the Cartesian coordinate point set $P = \{(x_1, 1), (x_2, 2), \ldots, (x_m, m)\}$, and a valid execution in this view comprises a “satisfied superset” of integer valued points $S$, where $P \subseteq S$. A point-set $S$ is satisfied if the axis-aligned rectangle defined by any pair of distinct points $a, b \in S$ with distinct horizontal and vertical coordinates contains at least one point in $S \setminus \{a, b\}$. The transition subtree for access $i$ comprises the set of points with vertical coordinate $i$, and can be reconstructed implicitly from the remainder of the execution [18]. Precursors of the geometric model by Derryberry et al. [21] appeared prior to [18].

The geometric version of Greedy algorithm is defined inductively: $S_1 = \{(x_1, 1)\}$ and $S_{i+1}$ is defined by adding the minimum set of points required to satisfy $S_i \cup \{(x_{i+1}, i + 1)\}$. In [18], Demaine et al. showed that the original and the geometric versions of Greedy are equivalent. Since then, many of the interesting properties that first drew attention to Splay have been proved for Greedy, including working-

---

10 Additional rules are needed for arranging nodes whose keys are not subsequently requested. See [35, Section 2.5.3] for details.

11 Standard versions of the geometric model do not require a specified initial tree, although this can be incorporated (see [6]).

12 If $a = (x_1, y_1)$ and $b = (x_2, y_2)$ then the axis-aligned rectangle is the set of points $\{(x, y) \mid \min\{x_1, x_2\} \leq x \leq \max\{x_1, x_2\} \text{ and } \min\{y_1, y_2\} \leq y \leq \max\{y_1, y_2\}\}$. 
set [22, 24] and dynamic finger [32], as well as some additional properties. See [35] for a thorough survey.

Many of the techniques developed in this work have either been (as with simulation embeddings) or could likely be (in the case of bounding additive overhead) applied to Greedy. However, Greedy is not the focus of this thesis. Adapting the machinery in this paper to the geometric model would likely constitute another investigation in its own right.

Lower Bounds

Wilber derived two lower bounds on the cost of executions for a given instance [64]. The first lower bound is computed by first choosing a fixed “reference tree” and then counting the number of “alternations” in the request sequence with respect to the reference tree.\(^{13}\) We examine Wilber’s second lower bound, which we call the crossing bound, at length in Chapter 5.

There is one other noteworthy lower bound, the “independent rectangle bound” (IRB). This bound, defined in the geometric model in [18], subsumes both of Wilber’s lower bounds. The similarity of this lower bound to Greedy serves as heuristic evidence that Greedy is dynamically optimal. We will not make use of the alternation or independent rectangle bounds in this thesis.

2.7 What is not Known

Currently, no on-line binary search tree algorithm is known to be constant competitive. Actually, our knowledge is much worse than this. There is no sub-exponential time algorithm whatsoever that is known to compute the cost of an optimum binary

\(^{13}\)The actual statement of the bound is somewhat intricate. See [35, Chapter 2.6.1] for a fairly clear description.
search tree execution for an instance to within a constant factor. None of the known lower bounds are known to be tight, although some are conjectured to be.

In fact, there is circumstantial evidence indicating that exactly computing the optimum cost of a binary search tree instance is NP-Complete: a slight generalization of the problem in which each task in the instance can request multiple keys is NP-Complete, as proved in [18]. The theoretical and practical difficulties we encountered when trying to reason about optimal binary search tree executions ultimately led us to the present approach, which consciously avoids directly comparing algorithms with optimal behavior.
Chapter 3

Simulation Embeddings

How can we prove that Splay’s cost never exceeds a constant multiple of the optimum without knowing what optimum executions “look like?” We answer this question by combining two concepts.

The first starts with a simple observation: in many situations, one intuitively expects that removing tasks from an instance should decrease the cost for the algorithm to execute it. This may not always be the case, but it is a reasonable idea to explore. Accordingly, we say that an algorithm is \emph{approximately monotone} if some fixed multiple of the time it requires to execute a list of tasks is an upper bound on the cost of executing any subsequence thereof.\footnote{In \cite{37}, we described approximate monotone algorithms as having the “subsequence property.” The benefit of hindsight motivated the change of terminology.} More formally:

\textbf{Definition 3.1} (Approximate Monotonicity). An algorithm $A$ is \emph{approximately monotone} (or simply \emph{monotone}) if there is some constant $b > 0$ so that $\cost_{A}(Y, T) \leq b \cost_{A}(X, T)$ for every request sequence $X$, subsequence $Y$ of $X$, and initial tree $T$. The constant $b$ is called the \emph{subsequence overhead}. If $b = 1$ then $A$ is called \emph{strictly monotone}.\footnote{Note that $b \geq 1$ since $X$ is a subsequence of itself.}
Approximate monotone functions are generalizations of monotone set functions: a real-valued set function \( f \) is monotone if \( f(A) \leq f(B) \) for all \( A \subseteq B \).\(^3\) Here, \( b > 1 \) is allowed. Note that a subsequence is not necessarily contiguous: a subsequence \( Y \) can result from any sequence of deletions of keys from \( X \). E.g. \((1, 3, 6)\) is a subsequence of \((1, 2, 3, 5, 6)\).

The second idea is to force an algorithm to simulate arbitrary executions by feeding it appropriately constructed “simulation-inducing” instances. A simulating instance must have two properties. First, the cost for the algorithm to execute the simulation should not exceed a fixed multiple of the simulated execution’s cost. Second, the simulation must contain the original instance corresponding to the simulated execution as a subsequence. We call a map from executions to simulation-inducing instances a simulation embedding.\(^4\) Formally:

**Definition 3.2** (Simulation Embedding). A simulation embedding \( S_A \) for an algorithm \( A \) is a map from executions to request sequences such that, for some \( c > 0 \),

- \( \text{cost}_A(S_A(E), T) \leq c|E| \), where \( |E| \) denotes the cost of \( E \), and

- \( X \) is a subsequence of \( S_A(E) \)

for every request sequence \( X \), initial tree \( T \), and execution \( E \) for the instance \((X, T)\).

Such a mapping can be used to simulate an optimal execution of a given instance just as well as any other execution. The cost for the algorithm to execute the simulation is, by construction, no more than a fixed multiple of the optimal cost for that instance.

Now comes the key point. The simulation of this optimal execution will, again by construction, contain the original instance as a subsequence. If the algorithm is also approximately monotone, then the cost of executing the original instance will

---

\(^3\)Technically, the algorithms we deal with are *sequence*-valued instead of *set*-valued, but the generalization is trivial.

\(^4\)As in, the original instance is “embedded” as a subsequence of the simulation sequence.
not exceed a fixed multiple of the simulation’s cost, and hence of the optimal cost. To conclude:

**Theorem 3.1.** Approximately monotone algorithms with simulation embeddings are constant-competitive.

We reserve the term “simulation embedding” exclusively for simulations constructed with the intention of reducing some question of constant competitiveness to one of proving that an algorithm is approximately monotone. (Many other uses of simulation-inducing instances exist.) In this work, we only build simulation embeddings for binary search tree algorithms. However, the concept itself seems much more general, and likely has applications outside of binary search trees.

The next section covers how to induce Splay to perform restricted rotations, as defined by [10], in constant time, and thereby transform a subtree arbitrarily in time proportional to the subtree’s size. Tree transformations are used in Chapter 3.2 to build a simulation embedding for Splay in the BST Model, and establish as a consequence that approximate monotonicity implies dynamic optimality. Tree transformations will also serve as the basis for several other results.

### 3.1 Tree Transformations

The term *restricted rotation* refers to rotating a node whose parent is either the root, or the root’s left child.

**Theorem 3.2 ([11] and [41]).** Any tree $T_1$ of size $n$ can be transformed into any other tree $T_2$ on the same set of keys through the application of at most $4n$ restricted rotations.
Cleary was the first to prove a linear bound on restricted rotation distance in [10]. Cleary and Taback improved this bound to $4n$ in [11]. Lucas presents Cleary’s result in terms of a standard BST model [41]. She describes an algorithm that uses restricted rotations to “unwrap” $T_1$ into a flattened tree $F$ that has no left child with a right child and no right child with a left child. $F$ is transformed into $T_2$ by applying the flattening algorithm in reverse. Figure 3.1 contains a sketch of this algorithm.

**Inducing Restricted Rotations**

The next theorem, despite its utter simplicity, forms the basis of our entire method. To begin, we view a splay operation as a function that takes a tree $T$ and a node

---

5Both papers proved these results in a manner far outside the description of binary search trees used by computer scientists. Cleary’s interest was in the generators of “Thompson’s group $F$.” Very roughly, this group corresponds, in a certain sense, to the class all maps which transform subtrees of roots of binary search trees. Apparently, this group is of some interest to algebraists due to a (correct) conjecture that it is an infinite simple group with a finite presentation. Readers intrigued by this digression may wish to consult [5].
$x \in T$ and returns a new tree $T'$. Note that we can only do this because of Splay’s very simple structure: the resultant tree $T'$ does not depend at all on the history of accesses. Define $G_n$, the transition digraph for Splay on binary search trees of $n$ nodes, as follows. Assign to every binary search tree with $n$ nodes and a fixed set of keys (say, integers 1 to $n$) a vertex in $G_n$. For each $T \in G_n$ and $x \in T$, draw an arc from $T$ to $\text{splay}(T, x)$.

**Theorem 3.3.** $G_4$ is strongly connected.\(^6\)

*Proof.* The reader may feel free to verify by hand the Hamiltonian cycle of $G_4$ depicted in Figure 3.2. \qed

**Remark 3.1.** We conjecture, but have not tried to prove, that $G_n$ has a Hamiltonian cycle for $n \geq 4$. Those interested in tackling this question may find it helpful to look at [39] and [42].

Theorem 3.3 opens the door to inducing restricted rotations in trees with four nodes. This is most easily seen by example. Suppose we start with tree XI in Figure 3.2, and that we wish to rotate at 2. The after-tree produced by this rotation corresponds to tree II in the diagram. Following the sequence of nodes marked by “∗,” we see that splaying the sequence of keys 1, 4, 2, 1 and then 4 produces the desired effect. Similarly, we can induce a rotation at 1 in tree III by splaying the sequence (1, 4, 1, 3). We can use an identical procedure for performing any other restricted rotation in a four-node tree: find the corresponding starting tree and final tree in the cycle of Figure 3.2 and splay at the indicated keys to transform accordingly.

While Figure 3.2 provides a striking visual depiction of this process, more pedestrian methods can improve the cost overhead of performing restricted rotations in this fashion.

\(^6\)Note that $G_3$ is not strongly connected.
Figure 3.2: A Hamiltonian cycle in $G_4$. Splay at nodes marked by * to convert one tree into the next.
Theorem 3.4. A sequence of at most five splay operations suffices to do a restricted rotation in a four-node binary tree.

Proof. By direct computation of the diameter of $G_4$. □

Remark 3.2. The manner in which a splay operation alters the search path is a function only of the path’s pointer structure. In particular, the structure of the subtrees hanging from the path have no bearing on the splay operation, save that they are reattached to the splayed path in accordance with the symmetric order. If we use splay operations to induce a restricted rotation in a four-node connected subtree of the root of a larger tree $T$, then we have performed the restricted rotation in $T$.

Transforming with Splay

Define a BST algorithm to have the transformation property if for every pair of trees $T$ and $T'$ on the same set of keys there exists a request sequence that induces the algorithm to transform $T$ into $T'$ in $O(n)$ time. The crucial consequence of Theorems 3.3 and Remark 3.2 is that Splay has the transformation property! Stated more precisely:

Theorem 3.5. Let $T$ and $T'$ be binary search trees of size $n \geq 4$ with the same keys. There exists a request sequence causing Splay to transform $T$ into $T'$ with cost at most $80n$.

Proof. By Theorem 3.2, it takes at most $4n$ restricted rotations to transform $T$ into $T'$. By Theorem 3.4, each restricted rotation can be done using no more than 5 splays. Each splay path has length 4 or less. Hence the total cost in the BST model is at most $4n \times 5 \times 4 = 80n$. □

The proof of Theorem 3.5 is designed for minimalism. A more careful analysis could reduce the constant factor. For example, half of the restricted rotations rotate
left or right at the root, and can be induced with cost 2. Additionally, our purposes are different from Cleary’s in [10], and relaxing the requirement that the rotated node’s parent is either the root or the root’s left child may enable further reduction of the constant.

**Remark 3.3.** In fact, Splay can transform subtrees in far more powerful ways. See *universal transformations* (Appendix A.1) and *simultaneous transformations* (Appendix A.2) for examples.

### 3.2 Monotonicity Implies Optimality

Applying the logic of Remark 3.2: the net effect of using splay operations to transform \( Q \) into \( Q' \), where \( Q' \) is a connected subtree of the root of another tree \( T \), is to substitute the subtree \( Q' \) for \( Q \) in \( T \). Hence Theorem 3.5 is a recipe for inducing Splay to perform *subtree transformations*. This can immediately be used to build a simulation embedding, \( S \), for Splay in the model of Definition 2.1, by stringing together the transformations that induce each subtree substitution of an execution. Now for the formalities.

In what follows, we may assume without loss of generality that all initial trees and transition trees in the BST model have at least four nodes. Given trees \( T \) and \( T' \) with the same keys, let \( \mathcal{T}(T, T') \) denote the sequence of keys generated by using the process described in Theorem 3.5 for transforming \( T \) into \( T'' \). \(^7\) (If \( T = T' \) then the sole term in \( \mathcal{T}(T, T') \) is the key in \( \text{root}(T) \).) We refer to \( \mathcal{T}(T, T') \) as the *transformation sequence* turning \( T \) into \( T' \). Additionally, let \( W \oplus Y \) denote the concatenation of sequences \( W = (w_1, \ldots, w_k) \) and \( Y = (y_1, \ldots, y_l) \) into \( (w_1, \ldots, w_k, y_1, \ldots, y_l) \). The input to \( S \) is

\(^7\)The freedom to choose different four-node subtrees for enacting restricted rotations with splaying means there can be many such sequences. This choice will not affect the following analysis, so we assume an arbitrary convention is imposed for choosing the subtrees.
an execution $E$ for $X = (x_1, \ldots, x_m)$ starting from $T$, comprising subtrees $Q_1, \ldots, Q_m$, transition trees $Q'_1, \ldots, Q'_m$, and after-trees $T_1, \ldots, T_m$, as described in Definition 2.1.

**Theorem 3.6.** The map $S(E) \equiv \mathbb{T}(Q_1, Q'_1) \oplus \cdots \oplus \mathbb{T}(Q_m, Q'_m)$ from executions to request sequences is a simulation embedding for Splay in the BST Model.

**Proof.** By construction, the execution of each block $\mathbb{T}(Q_i, Q'_i)$ for $1 \leq i \leq m$ substitutes $Q'_i$ for $Q_i$ in $T_{i-1}$, which brings the splayed tree’s shape in line with that of the execution’s. Next, notice that by Definition 2.1, $x_i$ is at the root of $Q'_i$. Note that the Splay algorithm always brings the splayed node to the root of the tree. Hence, the last key featured in $\mathbb{T}(Q_i, Q'_i)$ is always $x_i$, meaning $X$ is a subsequence of $S(E)$.

Finally,

$$\text{cost}(S(E), T) = \sum_{i=1}^{m} \text{cost} (\mathbb{T}(Q_i, Q'_i), T_{i-1}) \leq 80(|Q'_1| + \cdots + |Q'_m|).$$

The cost of the initial execution is $|Q'_1| + \cdots + |Q'_m|$. Hence, a constant multiple of the original execution’s cost is an upper bound on the cost for Splay to execute the simulation. We conclude that $S$ is a simulation embedding for Splay. \hfill \Box

**Theorem 3.7.** If Splay is approximately monotone then it is dynamically optimal.

**Proof.** By Theorems 3.1 and 3.6. \hfill \Box

Reducing dynamic optimality to approximate monotonicity is an example of a time-tested mathematical technique: to establish that a function (the Splay algorithm) has a certain property (its cost is $< c \cdot \text{OPT}$) at every point (an instance $(X, T)$) in a space (the collection of all BST instances),

- demonstrate that the property is true in some subset of the space (the simulation instances), and
• relate the function’s behavior (*via approximate monotonicity*) at other points in the space to its behavior in the subset.

### 3.3 Optimality Requires Monotonicity

Note the critical fact that Splay cannot be approximately monotone if\( \text{OPT} \) is not monotone. Fortunately, monotonicity is one of a select few properties that is relatively easy to prove about optimum BST executions.

**Theorem 3.8.** \( \text{OPT} \) is strictly monotone.

*Proof. Let \( Y \) be a strict subsequence of \( X = (x_1, \ldots, x_m) \) and \( T \) be an initial tree containing the keys of \( X \). Let \( E \) be an optimal execution for \( X \) starting from \( T \) with subtrees \( Q_1, \ldots, Q_m \), transition trees \( Q'_1, \ldots, Q'_m \), and after-trees \( T_1, \ldots, T_m \).

We build an execution for \( Y \) with initial tree \( T \) costing less than \( \text{OPT}(X, T) \).

Note that \( Y \) is formed by eliminating the requests in \( X \) that correspond to some subset of the integer indices from 1 to \( m \). Let \( j \) be the smallest index eliminated, and let \( k \) be the smallest index greater than \( j \) which was not removed from \( X \) to form \( Y \). If \( k \) does not exist (i.e. \( x_j, \ldots, x_m \) are all dropped) then simply remove \( Q_j, \ldots, Q_m \) and \( Q'_j, \ldots, Q'_m \) from \( E \) to form a valid execution for \( Y \). Otherwise we “elide” subtrees as follows.

Replace \( Q_j, \ldots, Q_k \) with \( Q \), the connected subtree of the root of \( T_{j-1} \) that contains all of the nodes in \( Q_j \cup \cdots \cup Q_k \). Then form the tree \( Q' \) by starting from \( Q \) and successively transforming the subtree \( Q_i \) into \( Q'_i \) for \( j \leq i \leq k \) (see Figure 3.3). The transition tree \( Q' \) replaces \( Q'_j, \ldots, Q'_k \), and becomes the transition tree for the access in \( Y \) that corresponds to the \( k^{th} \) access in \( X \). Observe that \( |Q| \leq |Q_j| + \cdots + |Q_k| - (k - j) \). Since \( k - j > 1 \), the cost of substituting \( Q' \) for \( Q \) in \( T_{j-1} \) is strictly less than the commensurate substitutions of \( Q_j, \ldots, Q_k \) in \( X \). Repeat this process for each
Figure 3.3: Example of eliding three transition trees.

subsequent contiguous subsequence of requests in $X$ that are missing from $Y$ to form a valid execution for $Y$ costing less than $E$.

Since $E$ is an optimal execution for $(X, T)$, $\text{OPT}(Y, T) < \text{OPT}(X, T)$. 

**Theorem 3.9.** If Splay is dynamically optimal then it is approximately monotone.

**Proof.** For all subsequences $Y$ of $X$, $\text{cost}(Y, T) \leq c \text{OPT}(Y, T) \leq c \text{OPT}(X, T) \leq c \text{cost}(X, T)$. The first inequality follows if Splay is dynamically optimal.

### 3.4 Partial Proof of Monotonicity

**Execution Encodings**

We can encode the pointer structure of a path from the root to a node by a binary string in which 1 signifies right and 0 signifies left (the empty string signifies access to the root). As mentioned in Remark 3.2, Splay only alters nodes on the path from the root to the accessed node, and in a way depending only on the path's binary encoding. Given a list of encodings of the paths to splayed nodes, it is always possible to uniquely reconstruct the initial tree $T$ and sequence of requests $X$ (up to an order-isomorphic relabelling of the keys). For example, the transformation of tree XI in Figure 3.2 into II via splaying is encoded as $(000, 1, 01, 0, 111)$. 

28
Given that approximate monotonicity is sufficient to establish dynamic optimality, we can rephrase the conjecture, as follows. We are given an execution encoding $C = (c_1, \ldots, c_m)$ and a deletion sequence $F$ of integer indices $1 \leq f_1 < \cdots < f_k \leq m$. If $T$ and $X$ are respectively the initial tree and request sequence implicitly defined by $C$ then the splay induced sub-encoding $G = (g_1, \ldots, g_l)$ corresponding to $(C, F)$ is the sequence of access path encodings encountered while splaying $Y = (y_1, \ldots, y_l)$ starting from $T$, where $Y$ is the subsequence formed by removing requests from $X$ whose indices are in $F$. The question of dynamic optimality is simply whether the total of the lengths of the strings in $G$ is bounded by some fixed multiple of the total of the lengths of the strings in $C$.

**Sequences of Increments and Decrements**

Theorem 3.7 tells us that we only need to show the monotonicity for executions whose encodings consist solely of strings with at most three characters. This is because executing a simulation sequence produced by the embedding of Theorem 3.6 never produces an access path of length greater than four. At least in principle, this is slightly stronger than simply stating that monotonicity implies dynamic optimality, since we only need to consider sub-executions of execution encodings comprising the fifteen different binary strings with length at most three. Beyond idle speculation, we provide the following evidence that this lower bar to proving dynamic optimality could possibly buy real power: we show that Splay is monotone on a subset of such executions.

**Theorem 3.10.** If $C$ is an execution encoding comprising entirely of the strings 0, 00, 1, and 11, then cost of $C$ upper bounds the cost of any of its splay-induced sub-encodings.

**Proof.** Each 0 or 1 will induce a single zig, and each 00 or 11 will induce a single zig-zig. Thus, the initial tree is flat, and each request in the original sequence induces
a single or double-rotation along the top of the tree, while maintaining the flat shape. In this context, each 0 or 1 effectively decrements or increments by one, and each 00 or 11 decrements or increments by two, assuming we canonically label nodes of the initial tree with integer keys 1 to n.

Assume that C only contains 0 or 1. Let X be the request sequence corresponding to C, and let Y = (y₁, . . . , y_l) be any subsequence thereof. By the dynamic finger theorem (Theorem 2.1), cost(Y, T) = O(|T| + \sum_{i=2}^{l} \log(|y_i - y_{i-1}| + 2), where we absorb cost of the first access into |T|. Notice for each pair y_{i-1} and y_i in Y, there are at least |y_i - y_{i-1}| corresponding requests in X, each of cost 2. Since \sum_{i=2}^{l} \log(|y_i - y_{i-1}| + 2) = O(m + \sum_{i=2}^{l} |y_i - y_{i-1}|), it follows that cost(Y, T) = O(cost(X, T)), and hence splay is approximately monotone on all such executions.8 Exactly the same logic applies to executions also containing 00 and 11.

Remark 3.4. The proof of the dynamic finger theorem given in [14, 13] is notoriously complicated. It seems likely that Theorem 3.10 could instead be proven with a slight generalization of the sequential access theorem from [61]. We believe it would likely resemble a proof that splaying (3, 2, 1)-avoiding permutations takes linear time, a topic discussed in Appendix B.4.

In actuality, we do not recommend attempting to prove approximate monotonicity by framing it in terms of path encodings, and we view this characterization as an intriguing dead end. The encoding-based viewpoint renders many aspects of the binary search tree model implicit, and therefore conceptually difficult to work with. The ideas discussed in Chapter 6 will likely prove to be more fruitful.

---

8The constant factors derived by Cole are universally regarded as dramatic overestimates of the true value, so we simply leave the bound inside big-O.
3.5 Competitive Ratios

It is worth asking whether approximate monotonicity is even needed. At least in the case of Splay, it is indeed required. For a simple example, let $T$ be a left spine with integer nodes from 1 to $n$, and let $X = (3,1,2)$ and $Y = (1,2)$. The reader can easily verify that $\text{cost}(X,T) = n + O(1)$ while $\text{cost}(Y,T) = 3n/2 + O(1)$. Hence, Splay’s subsequence overhead is at least $3/2$. With a little more effort, we can get a slightly better lower bound on $a$. Let $T$ be a left spine with integer nodes 1 to $2^k - 1$. If $X = (2^{k-1}, 2^{k-2}, \ldots, 2, 1, 2, 4, \ldots, 2^{k-1})$ and $Y = (1, 2, 4, \ldots, 2^{k-1})$, then $\text{cost}(X,T) = 2^k + o(2^k)$ while $\text{cost}(Y,T) = 2 \cdot 2^k + o(2^k)$. Hence Splay’s subsequence overhead is at least 2. We conjecture that this ratio is tight.

In fact, approximate monotonicity is useful even if Splay is not constant-competitive. For example, showing Splay has subsequence overhead $O(\log \log |T|)$ would suffice to establish that Splay is $O(\log \log |T|)$-competitive with OPT.\footnote{This itself would be a major result, improving over the best-known bound, $\text{cost}(X,T) = O(\log |T| \text{OPT}(X,T))$, that was first derived in [54].} Similarly if, hypothetically, there is some sequence of instances $(X_1,T_1), (X_2,T_2), \ldots$ for which $\text{cost}(X_n,T_n) = \Omega(\log \log n \text{OPT}(X_n,T_n))$, then $\frac{\text{cost}(X_n,T_n)}{\text{cost}(\text{OPT}(X_n,T_n),T_n)} = \Omega(\log \log n)$, and Splay’s subsequence overhead would be $\Omega(\log \log n)$.

In fact, Splay’s competitive ratio with OPT, to within a constant factor, is identical to its subsequence overhead. Formally, let $f(n) = \sup_{|T|=n,Y \subseteq X} \frac{\text{cost}(Y,T)}{\text{cost}(X,T)}$ and $g(n) = \sup_{|T|=n,X} \frac{\text{cost}(X,T)}{\text{OPT}(X,T)}$.\footnote{Here, $Y \subseteq X$ denotes $Y$ is a subsequence of $X$.}

**Theorem 3.11.** $f(n)/g(n) = \Theta(1)$.

Approximate monotonicity and dynamic optimality are very much equivalent.
3.6 Related Work

There are two prior works that construct simulation embeddings for binary search tree algorithms, and we encourage readers to examine both of these.

The earlier construction is tucked away in Dion Harmon’s Ph.D. thesis [28, Chapter 2.3.4]. He uses the geometric model of binary search tree algorithms to show that Greedy is constant-competitive if it is approximately monotone. His simulation embedding is as follows. Let \( E = (Q_1', \ldots, Q_m') \) be an execution \( X \), and for a binary search tree \( Q \) define \( G(Q) = (\bigoplus_{i=1}^{\vert Q \vert} (q_i, q_i)) \oplus q_1 \), where \( q_1, \ldots, q_{\vert Q \vert} \) are the nodes of \( Q \) ordered increasing by key. The simulation embedding for Greedy is the point set corresponding to the request sequence \( G(E) = G(Q_1') \oplus \cdots \oplus G(Q_m') \). Harmon proves that Greedy satisfies this point set by adding at most \( O(\vert Q_1' \vert + \cdots + \vert Q_m' \vert) \) additional points. He is the first to use the term “monotonicity” in this context.

The second investigation was undertaken by Luís Russo [50],\(^{11}\) who analyzed his simulation embedding for Splay using potential functions.\(^{12}\) He uses a model for binary search tree executions in which executions are sequences of Compare, Move-Left/Right, and Rotate operations.\(^{13}\) Rather than induce Splay to simulate executions’ internal states (i.e. their transition trees), he creates request sequences such that the original executions’ individual Rotate and Move operations comprise a subsequence of those enacted while splaying the simulations’ keys.

The first part of Russo’s construction essentially re-implements tree transformations via restricted rotations by showing how to simulate arbitrary executions while only rotating nodes at depth two or less. He then outlines where to add extra splays to synchronize Splay’s executions with those of the depth-restricted executions.

\(^{11}\)This article originally appeared on arXiv in 2015.
\(^{12}\)See Chapter 6.3 for a very brief introduction to potential functions.
\(^{13}\)Kozma [35] provides a nice description of this model, and a proof of its equivalence to the transition tree model.
Russo then upper bounds the cost of splaying a simulation sequence by that of the simulated execution through the use of some inventive modifications to the potential function employed by Sleator and Tarjan in their statement and analysis of the access lemma from [54]. His analysis, unlike ours, uses node identity. (See Remark 2.4.) Let $S$ be tree maintained by executing the simulation sequence with Splay and $T$ be the tree maintained by the execution being simulated.

Russo assigns each node a weight inversely proportional to the exponential of that node’s depth in $T$.\footnote{This weight assignment is very similar to that in an early version of the proof that Splay is statically optimal [52].} As in [54], he defines the rank of each node to be the logarithm of the sum of the weights in its subtree, and sets a tree’s potential to be the sum of its ranks. While nodes with the same key in $S$ and $T$ also have the same weight, the nodes are arranged differently in each tree. Thus, the two trees have different node ranks (and therefore different potentials). Furthermore, the weights change as $T$ changes. Russo uses the difference between the potentials of $S$ and $T$ to bound the amortized cost of the splays by the cost of the original execution. A good portion of Russo’s analysis is determining how rotations in either tree change both the weights and the ranks of the nodes.

Our manner of constructing simulation embeddings for Splay offers several advantages. The analysis is completely combinatorial, and more straightforward to carry through. The simplicity of our arguments also makes them readily generalizable. (See Chapter 7.4.) Furthermore, by building simulations using tree transformations, we are able to prove both the sufficiency and necessity of approximate monotonicity for dynamic optimality using the same machinery. Neither Harmon [28] nor Russo [50] established the necessity of approximate monotonicity.

This investigation is independent of those by Harmon and Russo, having been largely completed before we came across their work. It also seems Russo was unaware of Harmon’s previous work, which makes ours the third independent rediscovery of
simulation embeddings for binary search trees. This is perhaps an indication that there is something “intrinsic” about simulation embeddings in the dynamic optimality conjecture.

Demaine and Iacono independently conjectured that Splay is (approximately) monotone, although it seems they were unaware that monotonicity implies dynamic optimality [17]. To quote lecture notes scribed by Christopher Moh and Aditya Rathnam: “[A]n unpublished but simple conjecture states: Consider a sequence of accesses on a splay tree. If any of these accesses is removed, the resulting sequence of accesses is cheaper than the original sequence of accesses. This conjecture seems obvious, but has so far remained resistant to attacks.” Indeed, this simple conjecture is actually the dynamic optimality conjecture itself! Perhaps the importance of simulation embeddings is that they transform dynamic optimality from a property that seems remarkable into something almost mundane.

Other Simulations

Outside of simulation embeddings, the general notion of simulating binary search tree executions seems to be a recurring theme in the literature on the dynamic optimality conjecture. Appendix C.1 discusses some of the simulation arguments from [64, 40, 28, 35] that have led to the description of the BST Model featured in Definition 2.1. Lucas’ work [40] features simulations of BST executions subject to a number of different restrictions. More recently, Bose et. al. [4] showed how to simulate arbitrary binary search tree executions using transition trees of size $O(\log n)$, where $n$ is the initial tree’s size. Finally, in a slightly different direction, Demaine et. al. show how to simulate a multi-finger search tree data structure in the BST model [20].
Chapter 4

Eliminating Additive Overhead

It is conceivable, a priori, that an algorithm $A$ can be dynamically optimal, but with some “startup overhead.” For example, if $n$ denotes the size of $T$ then perhaps $\text{cost}_A(X, T) = O(\text{OPT}(X, T) + n \log \log n)$. Such a situation is not at all unimaginable. The typical reasoning ascribed to this possibility is that it may take “a few” accesses for Splay, or any other algorithm, to “sort itself out” given a “bad” initial tree. One might imagine starting with an “unbalanced” tree; for example, a left spine. We can formalize this notion:

Definition 4.1. An algorithm $A$ is dynamically optimal with additive overhead $g : T \rightarrow \mathbb{N}$, where $T$ denotes the collection of all binary search trees, if there exists some positive constant $c$ such that $\text{cost}_A(X, T) \leq c \text{OPT}(X, T) + g(T)$ for every request sequence $X$ and corresponding initial tree $T$. The overhead is intrinsic if $\sup_{X,T} \text{cost}_A(X, T)/\text{OPT}(X, T) = \infty$.

Remark 4.1. If $A$ costs at most as much as the optimum with additive overhead $g(T)$, then for $X$ containing $\Omega(g(T))$ requests, we can say that $\text{cost}_A(X, T) = O(\text{OPT}(X, T))$. In fact, this can be used as an alternative definition of additive overhead.

1By Remark 2.3 we could just as easily assume that $g(n) = \max_{|T|=n} g(T)$.
The observations of Chapter 3.2 already provide something of a “lower bound” on Splay’s additive overhead, which must absorb at least $\Omega(n)$ additional as compared with OPT, due to the subsequence discrepancy.\textsuperscript{2} Sleator and Tarjan optimistically speculated an $O(|T|)$ upper bound on the additive overhead separating Splay from OPT [54]. Remarkably, their guess must be right for the conjecture to be true!

4.1 Amplifying Transients

In this section we show that if Splay is dynamically optimal for some additive overhead then it also optimal without additive overhead. Equivalently, if Splay is dynamically optimal then it cannot have any intrinsic additive overhead.

The formalities require care, but the idea is quite straightforward. Create an “augmented” sequence from the original request sequence that induces Splay to restore the tree to its initial state. Both the Splay cost and the optimal cost of executing many repetitions of the augmented sequence scale linearly with both the number of repetitions and with the cost of executing the original sequence. If Splay is optimal with some overhead, then it is possible to determine a sufficient number repetitions in order to absorb this additive overhead into the constant of optimality.

More precisely, let $X$ be a request sequence with initial tree $T$. Let $V$ be the tree produced by splaying the keys $X$ starting from $T$, and define the augmented sequence $U = X \oplus \mathcal{T}(V,T)$, where $\mathcal{T}(V,T)$ is the transformation sequence that makes Splay transform $V$ into $T$. Denote by $k \ast U$ the sequence $U$ repeated $k$ times.

**Lemma 4.1.** $k \cost(X,T) \leq \cost(k \ast U,T)$.

*Proof.* First, note that $\cost(U,T) \geq \cost(X,T)$, since $U$ merely consists of requests appended to $X$. Second, $\cost(k \ast U,T) = k \cost(U,T)$, because executing $U$ with Splay

\textsuperscript{2}We can, if so desired, simply absorb this into OPT (see Remark 2.2), giving an overhead equal to zero. The observations in Chapter 6.3 will provide natural reasons to formally maintain this as a separate overhead.
ensures that the resultant tree shape is reset to $T$, making each repetition an identical execution. Combining the two inequalities yields the desired result.  

**Lemma 4.2.** $\text{OPT}(k \ast U, T) \leq 83k \text{OPT}(X, T)$.  

**Proof.** A bound on the optimal cost of an instance is established by constructing an execution of that instance that satisfies the bound and then noting that OPT, by definition, has the lowest cost of any execution. We establish two inequalities in this fashion.  

First, $\text{OPT}(U, T) \leq \text{OPT}(X, T) + 81|T|$. The upper bounding execution is built by appending to a sequence of transition trees for $\text{OPT}(X, T)$ the trees formed by splaying the paths to the nodes of $T(V, T)$, except with the first such path replaced by the tree formed by splaying the first node of $T(V, T)$ in $V$. The total lengths of the remaining paths are at most $80|T|$ by the transformation property (Theorem 3.5).  

Second, $\text{OPT}(k \ast U, T) \leq k \ast (\text{OPT}(U, T) + |T|)$. The upper bounding execution is constructed as follows. Let $B$ denote a sequence of transition trees for $\text{OPT}(U, T)$. Define $B'$ to be identical to $B$, except the first transition tree of $B'$ includes the entirety of the tree formed by substituting the first transition tree in $B$ into $T$. The execution consists of $B$ followed by $k - 1$ repetitions of $B'$. The cost of $B'$ is at most $|T|$ greater than $B$, hence the inequality.  

Combining the two bounds on OPT gives

$$\text{OPT}(k \ast U, T) \leq k \text{OPT}(X, T) + 82|T|)$$

$$\leq 83k \text{OPT}(X, T),$$

where the last inequality follows from Remark 2.2.  

**Theorem 4.1.** If $\text{cost}(X, T) \leq c \text{OPT}(X, T) + g(T)$ then $g$ is not intrinsic.
Proof. By applying the previous inequalities:

\[
\kappa \cdot \text{cost}(X, T) \leq \text{cost}(k \cdot U, T) \leq c \cdot \text{OPT}(k \cdot U, T) + g(T) \\
\leq 83c \cdot \text{OPT}(X, T) + g(T) \leq (83c + 1)\kappa \cdot \text{OPT}(X, T) \quad \text{[Lemma 4.2]}
\]

Since \(\text{cost}(X, T) / \text{OPT}(X, T) \leq 83c + 1\), the function \(g\) is not intrinsic. \(\square\)

Remark 4.2. Our original proof of this theorem established the contrapositive, by “amplifying” hypothetical instances on which Splay is non-optimal to break any presumed additive overhead. An outline of the above (simpler) version of the proof was kindly supplied to us by Kurt Mehlhorn [44] after he reviewed our manuscript for [37].

It is worth considering the consequences of Theorem 4.1. Suppose, hypothetically, that there is a sequence of instances \(\{(X_1, T_1), (X_2, T_2), \ldots\}\) where \(|T_n| = n\), \(\text{cost}(X_n, T_n) = \Theta(n \log n)\), and \(\text{OPT}(X_n, T_n) = \Theta(n)\). A priori this is compatible with the possibility that \(\text{cost}(X, T) = \Theta(\text{OPT}(X, T) + |T| \log |T|)\). However, by Theorem 4.1 this cannot be the case. Either \(\text{cost}(X, T) = O(\text{OPT}(X, T))\), or Splay cannot be optimal with any additive overhead.

A lovely consequence of Theorem 4.1 is that approximate monotonicity, even with “additive overhead,” still implies dynamic optimality (without additive overhead).

Corollary 4.1. If there exists a constant \(c\) and a function \(g : T \to \mathbb{N}\) such that \(\text{cost}(Y, T) \leq c \cdot \text{cost}(X, T) + g(T)\) for all request sequences \(X\), subsequences \(Y\), and initial trees \(T\) then Splay is dynamically optimal.
Proof. By Theorem 3.6,

$$\text{cost}(X, T) \leq c \text{cost}(S(\text{OPT}(X, T)), T) + g(T)$$

$$\leq 80c \text{OPT}(X, T) + g(T).$$

Apply Theorem 4.1. \qed

4.2 Background

The notion of optimality with an additive overhead has something of a history with respect to the conjecture. The earliest example comes from Sleator and Tarjan’s original paper [54], via the “balance theorem”: splaying $m$ keys in a binary search tree with $n$ nodes costs $O((m + n) \log (m + n))$. According to this bound, if $m$ is asymptotically less than $n$, for example $m = \Theta(n / \log \log n)$, then an optimal execution could perform asymptotically better than Splay on “short” request sequences. The preliminary version of Cole’s work on the dynamic finger theorem [12] included an $O(n \log \log n)$ additive overhead in the bound, which required serious footwork involving inverse-Ackerman functions and significantly bloated constants to eliminate [14, 13].

Some consideration has also been devoted to Greedy’s additive overhead. In particular, it is conjectured in [18] that GreedyFuture’s execution cost never exceeds the optimal by more than an additive factor of $O(m)$ when serving $m$ requests. Finally, Iacono’s “proof-of-concept” algorithm [31], if dynamically optimal, will have an (admittedly ludicrous) super-exponential additive overhead in initial tree size.

Despite these ideas appearing implicitly throughout the literature, there seem to be no published results (prior to [37]) that give an upper bound on the additive over-
head with which a binary search tree algorithm can be optimal. It seems somewhat surprising that this issue had not received further attention, as it has likely applications to recent work on pattern-avoiding access in binary search trees [6], not to mention enabling the reductions of both the deque and traversal conjectures to approximate monotonicity. (See Chapter 7.) From personal conversations, we are aware that John Iacono had independently considered some of the methods employed in this chapter in unpublished investigations.

The only other similar work we are aware of is Koumoutsos’ master’s thesis [34], whose “phase-based” analysis (our terminology) of optimum executions somewhat resembles the techniques used above.
Chapter 5

Wilber’s Crossing Lower Bound

Prior to [37], Wilber’s “crossing” lower bound never had more than supporting roles in various lemmas and corollaries scattered throughout the literature.\(^1\) But it seems this lower bound on the cost of binary search tree executions may yet play a crucial part in the story of the elusive dynamic optimality conjecture.

We introduce a version of the crossing bound defined in terms of the “crossing cost” of an algorithm called “Move-to-Root [3].” This version, implicit in Iacono’s work [30] and alluded to in [31] will be more amenable to the present purposes than Wilber’s original definition of the bound [64].\(^2\)

We will see that the crossing bound is *approximately monotone*. This fact sets the stage for the material in Chapter 6.

5.1 Treaps and Move-to-Root

Move-to-Root is the simplest non-trivial algorithm that fits in Sleator and Tarjan’s cost model. To execute Move-to-Root, first search for the requested key \(k\), and then repeatedly rotate the returned node \(x\) until it becomes the root. Allen and Munro

\(^1\) Perhaps the most interesting of these is Iacono’s work on “key-independent” optimality [30].
\(^2\) See Appendix C.3 for a proof that this definition is equivalent to the original.
were the first to analyze this algorithm, and while it is not dynamically optimal, it has good expected behavior [3]. Move-to-Root’s similarity to Splay is striking, and it is one of Splay’s progenitors [54]. Indeed, we will argue in Chapter 6 that Splay is connected with Move-to-Root in many ways.

A treap is a binary search tree in which each node is assigned a unique priority from a totally ordered set. The nodes of a treap obey the usual symmetric order condition with respect to their keys, and the max-heap order condition with respect to the priorities: a non-root node’s priority never exceeds its parent’s. There is only one way to arrange the nodes of a binary tree in a manner satisfying both the symmetric and heap orderings with respect to a given set of distinct keys and priorities: the root is the node of greatest priority, and the left and right subtrees comprise the treaps of nodes whose keys respectively smaller and greater than the root. Jean Vuillemin constructed treaps from permutations, setting each node’s priority to the position of its key in the the permutation [63]. The term “treap” is due to Seidel and Aragon [51].

Move-to-Root is the unique algorithm such that, after each access, the nodes are arranged as a treap in which a node’s priority is the most recent time at which its key was requested. The action of Move-to-Root is equivalent to setting the priority of the node that holds the requested key to one greater than that of all other nodes, and then restoring the treap order. (The first key is accessed at time 1.)

We can see as follows that Move-to-Root restores the heap-order invariant. Resetting the priority of the requested node $x$ introduces a single “heap-order violation” at the edge between $x$ and its parent (if the parent is present). After each rotation at $x$ that does not result in $x$ becoming the root, only a single edge in the tree violates the heap order, and that edge is always the one between $x$ and its parent. When $x$ becomes the root, it has the largest priority, and no other edges violate the heap

---

3 Consider executing $X = (1, 2, \ldots, n)$ with keys initially arranged in a left spine.
order. That Move-to-Root is the unique algorithm restoring heap order stems from
the uniqueness of the treap for the given priorities.

Remark 5.1. To produce initial tree $T$, give each node $x \in T$ initial priority of
$\tau(x) - |T| - 1$, where $\tau(x)$ is the index at which $x$ appears in postorder($T$) = $(p_1, \ldots, p_n)$.
By definition of postorders, root($T$) will have highest priority under this assignment,
and this will hold true recursively in the subtrees.\footnote{We ensure that the first access time is greater than the initial priorities by making the initial
priorities negative. In fact, any set of negative priorities that make the treap have the same shape
as $T$ will do. Choosing postorders is a matter of convenience.}

5.2 Crossing Cost and Bookkeeping

Let $x$ be a node in a binary tree $T$, and consider the subtree $P$ of $T$ comprising
the nodes of the path connecting $x$ to the root of $T$. The crossing nodes for $x$ in $T$
comprise $x$, the root of $T$, and the nodes in $P$ that are either left children with a
right child on $P$ or right children with a left child on $P$. We refer to the number of
crossing nodes for $x$ as the crossing depth or level of $x$ in $T$, and denote it $\ell_T(x)$.
The bookkeeping nodes are the non-crossing nodes of the access path.

If $X = (x_1, \ldots, x_m)$ is a request sequence with initial tree $T_0$, and $T_1, \ldots, T_m$
are the after-trees of an execution $E$ for $(X,T_0)$, then the crossing cost of execution $E$ is
$\sum_{i=1}^{m} \ell_{T_{i-1}}(x_i)$, and the bookkeeping cost is $\sum_{i=1}^{m} (1 + d_{T_{i-1}}(x_i) - \ell_{T_{i-1}}(x_i))$.

Definition 5.1 (Crossing Bound). The crossing bound (or Wilber’s bound) for in-
stance $(X,T)$, denoted by $\Lambda(X,T)$, is the crossing cost of Move-to-Root’s execution
of this instance.

Theorem 5.1. $\Lambda(X,T) = O(\text{OPT}(X,T))$.

This is essentially \cite[Theorem 7]{64}. Appendix C.3 covers a few technicalities
that are needed in order to translate Wilber’s original result into the BST Model of
Definition 2.1. For the present purposes, we focus on the intuition behind this bound.
At a high level, the binary search tree model is a natural extension of the “list-based” caching model that Sleator and Tarjan explored in tandem with their work on Splay trees [53]. In the list-based model, the data structure is a linked list, searches proceed by linear scan from the front of the list, and the fundamental restructuring operation is “swapping” two adjacent elements. The authors show that Move-to-Front, which simply moves each requested item to the front of the list, is constant competitive in this model. This was a source of inspiration for the dynamic optimality conjecture.

The difference between the list-based model and the BST model is as simple as it is profound: in the list-based model, nodes have a single “next” pointer, whereas nodes in the BST model have two pointers, “left” and “right”. The additional pointer in the BST model allows breaking lists into “short” access paths that comprise alternating left and right children. Thus, as Iacono intimates in [31], the BST model’s true power stems from the existence of crossing nodes, and the remaining nodes seem to simply be for “bookkeeping.” It is thus not so surprising to find a lower bound, Λ, based on crossing costs.

Wilber argues convincingly why we might expect the crossing costs to be related to keys’ prior access times [64]. From Definition 2.1, when a key is accessed, a BST algorithm must bring the node holding it to root. This requires relocating nodes that were brought to the root by previous requests. Wilber’s bound essentially counts the number of such nodes that must be “moved back out of the way” in order to complete a request. Treaps with time-ordered priorities provide a natural way to represent this information. Move-to-Root neatly maintains this treap.

**Remark 5.2.** Crossing nodes can be computed visually. Let \( p_1, p_2, \ldots, p_k \) be the nodes of the access path for \( x \) in \( T \). Draw a line vertically from \( x \) up to infinity.\(^5\) For \( 1 < i < k \), if the edge connecting \( p_{i-1} \) to \( p_i \) crosses this line then \( p_i \) is a crossing node.

\(^5\)A node’s horizontal coordinate is its key’s position in symmetric order; its vertical coordinate is the negative of its depth.
Figure 5.1: Execution of $X = (8, 2, 7, 4)$ starting from $T$ by Move-to-Root. There are 14 total crossing nodes. Hence, $\Lambda(X, T) = 14$.

(The endpoints are automatically crossing nodes.) See Figure 5.1 for an example of graphically computing Wilber’s bound.

5.3 Wilber’s Bound is Approximately Monotone

Proving monotonicity, generally, requires comparing how the algorithm executes two different request sequences starting from the same tree: the sequence and the sub-sequence. For Wilber’s bound, however, the following theorem enables reducing this problem to comparing how Move-to-Root executes the same request sequence starting from different trees.

**Theorem 5.2.** $\Lambda(Z, S) - \Lambda(Z, \text{move-to-root}(S, x)) \leq 4\ell_S(x)$ for all request sequences $Z$, initial trees $S$, and $x \in S$.

We found this innocuous-looking lemma to be surprisingly difficult to prove. However, our basic strategy is straightforward. Let $Z = (z_1, \ldots, z_m)$ and let $S_0 = S$ and $T_0 = \text{move-to-root}(S_0, x)$. For $1 \leq i \leq m$, let $S_i = \text{move-to-root}(S_{i-1}, z_i)$ be the after-trees of Move-to-Root’s execution of $(Z, S_0)$, let $T_i = \text{move-to-root}(T_{i-1}, z_i)$ be the after-trees of Move-to-Root’s execution of $(Z, T_0)$, and define $\Delta_{i-1}(y) = \ell_{S_{i-1}}(y) - \ell_{T_{i-1}}(y)$ for $y \in S$. Note that $\Lambda(Z, S) - \Lambda(Z, \text{move-to-root}(S, x)) = \sum_{i=1}^{m} \Delta_{i-1}(z_i)$ by definition.
We derive explicit expressions for the functions $\Delta_0, \ldots, \Delta_{m-1}$ and directly bound the sum. This involves a substantial amount of case analysis. The details are relegated to Appendix D for reference, and may be skipped upon first reading of this thesis.

**Remark 5.3.** In our paper [37, Lemma 6.1], we *incorrectly* claimed that $\Lambda(Z, S) - \Lambda(Z, \text{move-to-root}(S, x)) \leq \ell_S(x)$, and gave a proof that contains several errors.\(^6\) A counter-example is $S = \text{BST}(1, 7, 4, 2, 3, 6, 5)$ with request sequence $Z = (5, 3)$ and $x = 4$. We also stated that $\Lambda$, as defined above, is a *strictly* monotone function of $Z$, which is false.

Furthermore, in [37, Remark 6.2] we alluded to a “geometric” proof that Wilber’s bound is monotone that would be supplied in a later version of the paper. This proof was supposed to be comparatively simpler than the machinery we use in this thesis. Unfortunately, the geometric proof had fundamental structural issues that we were unable to repair as of the time of writing.

Finally, we also implied that it is “easy” to prove that the independent rectangle bound from [18] is (strictly) monotone. As far as we are aware, it is actually an open question whether or not the Independent Rectangle Bound is strictly, or even approximately monotone. Regardless of the truth, we retract our claim that this is a “trivial” matter.

**Theorem 5.3.** Wilber’s bound is approximately monotone with subsequence overhead at most four.

**Proof.** By induction on the number of request deletions used to form a subsequence from a sequence. Let $X = x_1, \ldots, x_m$ be a request sequence with starting tree $T = T_0$. For $1 \leq i \leq m$, let $T_i = \text{move-to-root}(T_{i-1}, x)$. Let $m \geq e_1 > e_2 > \cdots > e_k \geq 1$ be a “deletion sequence” of request indices. Let $X_0 = X$, and for $1 \leq j \leq k$, form $X_j$ by removing the $e_j^{th}$ request from $X_{j-1}$. Note that $\Lambda(X_0, T) = \Lambda(X, T)$ by construction.

Now suppose $\Lambda(X_{k-1}, T) \leq \Lambda(X, T) + 3 \sum_{i=1}^{k-1} \ell_{T_{e_i-1}}(x_{e_i})$.

\(^6\)The proof of Theorem 5.2 in Appendix D is a corrected and modified version of [37, Section 8].
Note that $X_{k-1} = W \oplus (x_{e_k}) \oplus Z$ while $X_k = W \oplus Z$, where $W = x_1, \ldots, x_{e_k-1}$ and $Z$ is the remainder of the request sequence for $X_{k-1}$. Thus,

$$\Lambda(X_{k-1}, T) = \Lambda(W, T) + \ell_{T_{e_k}}(x_{e_k}) + \Lambda(Z, \text{move-to-root}(T_{e_k-1}, x_{e_k}))$$

$$\Lambda(X_k, T) = \Lambda(W, T) + \Lambda(Z, T_{e_k-1}).$$

By Theorem 5.2, $\Lambda(Z, T_{e_k-1}) \leq 4\ell_{T_{e_k-1}}(x_{e_k}) + \Lambda(Z, \text{move-to-root}(T_{e_k-1}, x_{e_k}))$, thus

$$\Lambda(X_k, T) \leq \Lambda(X_{k-1}, T) + 3\ell_{T_{e_k-1}}(x_{e_k})$$

$$\leq \Lambda(X, T) + 3 \sum_{i=1}^{k} \ell_{T_{e_i-1}}(x_{e_k})$$

$$\leq 4\Lambda(X, T).$$

$\square$
Chapter 6

Proposal for Proving Optimality

I put forward the following proposal for how to structure a proof that Splay is dynamically optimal, that builds on the tools developed so far. The key insight, assuming these ideas prove fruitful, is to separately analyze Splay’s crossing cost and its bookkeeping cost. Let $\Lambda'(X, T)$ denote the crossing cost of Splay’s execution of $(X, T)$ and let $\zeta(X, T)$ denote the bookkeeping cost of this execution.\footnote{Note that, because Splay only rearranges the access path, $\text{cost}(X, T) = \Lambda'(X, T) + \zeta(X, T)$.}

Splay’s crossing cost appears to be strongly correlated with the crossing lower bound (Chapter 6.1). I propose two ways to exploit this. The first, and perhaps more obvious way, is to directly bound the value of $\Lambda'$ with Wilber’s bound, $\Lambda$ (Chapter 6.2). The second approach, and the one I favor, is to adapt our proof of approximate monotonicity from the crossing bound to Splay’s crossing cost (Chapter 6.2). Either one of these analyses will almost certainly involve a potential function; I discuss what this potential might “look like” (Chapter 6.3). Splay’s bookkeeping cost should be bounded by its crossing cost (Chapter 6.4).

Disclaimer. My proposals in this chapter are entirely speculative, and could be partially or even completely wrong. Having said this, my suppositions are informed by extensive informal evidence of some variety, including folklore knowledge gleaned...
from discussions with colleagues, comments from other papers, notebooks filled with my failed attempts to prove various theorems, and most of all, numerical experiments run using my personal implementations of the algorithms discussed herein. The results of my numerical experiments in particular have been corroborated in large part by John Iacono. I am confident that, at a minimum, the conjectures in this chapter can only be wrong for interesting reasons.

6.1 Move-to-Root and Splay

The following conjecture provides the preliminary impetus comparing Splay with the crossing lower bound:

**Conjecture 6.1.** $\Lambda'(X, T) = O(\Lambda(X, T) + |T|).$

**Remark 6.1.** In fact, my numerical experiments seem to imply a much deeper relationship between Splay and the crossing lower bound: I believe $\text{cost}(X, T) = O(\Lambda(X, T) + |T|)$. This is actually stronger than the dynamic optimality conjecture, since it implies the crossing lower bound is tight. This would also imply Conjecture 6.1.

Conjecture 6.1 becomes more intuitive in the *global view* of template algorithms [7]. Wilber’s bound is defined by the crossing cost of Move-to-Root. Quoting the description of Splay from [7]: “Splay extends Move-to-Root: Let $s = v_0, v_1, \ldots, v_k$ be the reversed search path. We view splaying as a two step process, see Figure 6.1. We first make $s$ the root and split the search path into two paths, the path of elements smaller than $s$ and the path of elements larger than $s$. If $v_{2i+1}$ and $v_{2i+2}$ are on the same side of $s$, we rotate them, i.e., we remove $v_{2i+2}$ from the path and make it a child of $v_{2i+1}$.”

Interestingly, it is possible that $\Lambda'(X, T) < \Lambda(X, T)$. An example is $X = (3, 1, 4, 2)$ and $T = \text{BST}(3, 1, 2, 4)$.

This description of Move-to-Root is identical to the “unzipping” operation in Zip Trees [62].
Figure 6.1: A global view of Splay trees. The transformation from the left to the middle illustrates Move-to-Root. The transformation from the left to the right illustrates Splay trees. (Figure and caption from [7]).

Paraphrasing, Splay first executes Move-to-Root, and then performs extra rotations, the zig-zigs, along the side-arms of the after tree to ensure a “depth-halving” effect. In the language of treaps, each of these zig-zigs can create a “violation” in the max-heap ordering with respect to most recent access time.

As the executions of both Splay and Move-to-Root proceed, these zig-zigs will sometimes create further heap-order violations. At other times, the various splay steps will remove some of the heap-order violations. In my experience, tracking the creation and destruction of heap-order violations in Splay’s executions rapidly becomes complicated. Yet, the close relationship between Splay and Move-to-Root provides a reason to suspect that they only get out of sync “gradually.” As we shall see in Chapter 6.3, this strongly suggests the use of a potential function [60] for tracking these violations.

Of course, it has not been specified what this potential would be used for. This question brings us to a fork in our path.
Figure 6.2: Comparison of crossing nodes for Splay and Move-to-Root when $X = (8, 2, 7, 4)$.

6.2 Splay’s Crossing Cost

Road I: Wilber’s Bound

The obvious way forward is attempting to directly prove Conjecture 6.1. I have tried at length to do so. Suffice it to say, this work would have included a proof had I succeeded. John Iacono has also noticed this relationship, and spent some time trying to prove it, to no avail. To the extent that it is possible to describe why a certain kind of proof is ineffective for attacking a given problem, I offer some comments on where the difficulties seem to lie.

Consider a request sequence $X = (x_1, \ldots, x_m)$ and let $P_1, \ldots, P_m$ and $P'_1, \ldots, P'_m$ denote the paths encountered while executing $X$ starting from $T$ with, respectively, Move-to-Root and Splay. Path-for-path comparisons on various request sequences reveal that these two algorithms appear to share more than crossing costs. In fact, for most $1 \leq i \leq m$, the keys of the crossing nodes along each $P_i$ and $P'_i$ will be extremely similar (see Figure 6.2), albeit sometimes “offset” from each other in
symmetric order by a small amount. The crossing nodes in $P_i$ do not always appear directly in $P'_i$, however.

Usually, $P'_i$ contains about one half to one third of the crossing nodes in $P_i$. Other of $P_i$’s crossing nodes appear in $P'_{i+1}$, a few more in $P'_{i+2}$, and so on. In essence, there appears to be some “temporal spreading” in terms of when Move-to-Root’s crossing nodes appear within the splay paths. The extent of temporal mixing is somewhat varied and depends on the particular request sequence. Likely, this is due to the differing extents to which nodes of the splayed tree violate the heap order with respect to most recent access time.

Joan Lucas has remarked that, in a number of contexts, OPT does not seem amenable to analysis by direct inductive proofs [40]. I believe the observed temporal mixing is one manifestation of this difficulty. Essentially, any inductive proof relating $\Lambda'(X, T)$ to $\Lambda(X, T)$ must account for multiple previous requests at the inductive step. If true, this issue essentially destroys what makes inductive arguments simple.

Nonetheless, I think Conjecture 6.1 merits further investigation. It would certainly be elegant and useful if true, and it is quite possible the above impediments to proving it are a mere product of my lack of imagination.

**Road II: Approximate Monotonicity**

I see one other viable route to analyzing $\Lambda'$:

**Conjecture 6.2.** $\Lambda'(Y, T) = O(\Lambda'(X, T) + |T|)$ for all subsequences $Y$ of $X$.

The machinery that we built in order to prove Wilber’s bound is approximately monotone offers a clear starting point for attacking Conjecture 6.2. By contrast, I am not aware of any currently existing techniques that seem readily applicable to proving Conjecture 6.1. Thus, Conjecture 6.2 appears more tractable at the present time.

---

4This behavior is not seen in the figure; the examples are larger.
Adapting the machinery of Theorem 5.3 from Wilber’s bound to Splay requires changing the form of the induction. In particular, it seems rather unlikely that Conjecture 6.2 can be proved by analyzing the resultant increase in Splay’s crossing cost due removing a single request, as we did for Wilber’s bound using Theorem 5.2. In fact, the removal of a single request can increase Splay’s crossing cost by \( \Omega(\left| T \right|)! \)5

Thus, Conjecture 6.2 seems to require a “direct” demonstration that removing an arbitrary subset of requests increases Splay’s crossing costs by at most a constant factor in total.

More precisely, let \( X = (x_1, x_2, \ldots, x_m) \), with \( Y = (y_1, y_2, \ldots, y_{m-k}) \) produced as a subsequence of \( X \) through deleting request numbers \( 1 \leq d_1 < \cdots < d_k \leq m \). For \( 1 \leq i \leq m \), define \( X_i = (x_1, \ldots, x_i) \), \( Y_0 = \emptyset \), and \( Y_i = (y_1, \ldots, y_{i-\max\{j|d_j \leq i\}}) \). Conjecture 6.2 requires a double induction on both \( i \) and \( k \) that establishes \( \Lambda'(Y_i, T) \leq c(\Lambda'(X_i, T) + |T|) \) for some constant \( c \). This style of induction entails comparing how Splay executes different request sequences starting from different trees.

A step in this direction is reproving the approximate monotonicity of the crossing bound via the above-mentioned double induction. Unfortunately, even this preliminary step is likely to be arduous: the proof of Theorem 5.3 via Theorem 5.2 is already unpleasantly complicated, so the more intricate version, frankly, will not be pretty. On the bright side, Theorem 5.3 ensures that a proof via this alternate method is, at least in principle, achievable.

This alternate proof of Theorem 5.3 must then be converted from \( \Lambda \) to \( \Lambda' \). The conversion process will almost certainly require a potential function in order to smooth out the effects of occasional requests whose removal produces a high increase in Splay’s crossing costs. I discuss this matter in the next section.

---

5 For example, let starting tree \( T \) be formed by Splaying key 2\( n \) in a right spine comprising integer keys 1 to 2\( n \). Let \( Y = (2, 4, \ldots, 2n - 2) \) and \( X = (1) \oplus Y \). The crossing cost for splaying the first item of \( X \) starting from \( T \) is 2, and every subsequent splay in \( X \) has crossing cost 3. Meanwhile, the first splay in \( Y \) has crossing cost 4, and every subsequent splay in \( Y \) has crossing cost 5. Thus, \( \Lambda'(Y, T) - \Lambda'(X, T) = \Omega(|T|) \).
I leave readers at a fork in the road. Road I is more obvious, yet so far it has led explorers astray. Perhaps the second road, less well-travelled, can make all the difference.

6.3 Potentials for Heap-Order Violations

A potential function is a tool for analyzing algorithms that have individual operations with high cost, but for which the cost per operation, amortized over all requests in a sequence, is low. (Splay is of course a perfect example.) Each possible configuration of the data structure (e.g. the tree) is assigned a numerical value, called its potential. We then redefine the cost of an operation to depend on both the “actual” cost, and on how the potential changes due to the operation’s effect on the data structure. If carefully constructed, the sum of the redefined costs over a sequence of operations will be an upper bound on the sum of the actual costs, yet no individual operation’s “redefined” cost will ever be very large. This relates to our problem in the following way.

Consider request sequence $X = (x_1, \ldots, x_m)$. While the crossing depth may differ greatly for any individual $x_i$ at the time it is accessed in the splayed tree vs. the tree maintained by Move-to-Root, Conjecture 6.1 states that $\Lambda'(X, T)$ and $\Lambda(X, T)$ are always tightly coupled. Similarly, despite evidence that the removal of any individual request may greatly increase $\Lambda'$, if Conjecture 6.2 is correct then removing an arbitrary subset of requests will not increase $\Lambda'$ too much in total. Both of these conjectures are ripe for analysis via a potential function.

I am convinced that the correct potential for either of the above problems should in some way “smooth out” Splay’s heap-order violations. Constructing this potential function is one of the biggest remaining roadblocks to proving dynamic optimality that is left open by this work. It is, however, possible to infer something very important.
As noted in [33], a potential function’s design is closely tied to the extent to which the potential’s value can increase or decrease; i.e. its range. Typically, the potential’s range is used to determine the algorithm’s additive overhead. By Theorem 4.1, if Splay is optimal then its additive overhead will not exceed \( O(|T|) \). Hence, the potential(s) used to prove Conjectures 6.1 or 6.2 should have maximum value at most \( O(|T|) \). This narrows the “design space” that we might otherwise want to explore.

I speculate on two possible forms for the potential. The first simply counts the number of edges in the tree being splayed that violate the heap-order condition with respect to most recent access time. I have spent some time analyzing this simple potential, but not enough to form an opinion about whether it will suffice for the purpose at hand. If this potential is not up to the task, the likely reason will be that it is too “coarse,” in that it fails to capture heap-order violations between nodes not immediately connected by an edge.

In case the potential does require more granularity, it seems reasonable to address this through weighting each node by some function of the difference between its crossing depth in the splayed tree and in the treap maintained by Move-to-Root. Russo’s potential from [50] may be a good starting point for gleaning inspiration. But any further speculation that I provide on the form of this second type of potential is more likely than not to simply make readers liable for my own ignorance.

### 6.4 Splay’s Bookkeeping Cost

Because Move-to-Root’s crossing cost forms a lower bound on OPT, Move-to-Root is only non-optimal due to sometimes having high bookkeeping cost. Splay tweaks Move-to-Root by breaking apart bookkeeping edges in the side-arms of the after-tree via zig-zig steps. Splay thus seems to be precisely the modification needed to make Move-to-Root optimal. The numerical evidence backs this up, and I believe
Conjecture 6.3. $\zeta(X, T) = O(\Lambda'(X, T) + |T|)$. The constant inside the big "O" is at least two.

Theorem 6.1. If Conjecture 6.3 and either of Conjectures 6.1 or 6.2 are correct then Splay is dynamically optimal.

Proof. If Conjectures 6.3 and 6.1 are correct then $\text{cost}(X, T) = O(\Lambda(X, T) + |T|) = O(\text{OPT}(X, T))$, where the last bound follows by Wilber’s proof of Theorem 5.1. If Conjectures 6.3 and 6.2 are both correct then Splay is approximately monotone, and therefore dynamically optimal by Theorem 3.7.

There is good empirical reason for separately exploring Splay’s crossing and bookkeeping cost. Qualitatively, $\zeta$ has a different “character” from $\Lambda'$. Most notably, $|\zeta(X, T) - \Lambda(X, T)|$ exhibits far more variability than $|\Lambda'(X, T) - \Lambda(X, T)|$: sometimes the former difference is much greater, and for other instances the former difference is negligible.

Comparing the trees resulting from splaying paths with many zig-zigs to those resulting from splaying paths with many crossings yields some intuition about why Conjecture 6.3 might hold. Heuristically, Splay turns crossings into zig-zigs, and zig-zigs into crossings. Precisely tracking the creation and destruction of crossings and zig-zigs as Splay’s execution progresses seems to become quickly unmanageable, which suggests the use of a potential function to do so.

It seems reasonable that an appropriate potential function will act as a proxy for "the number of bookkeeping nodes" in the tree being splayed. Intuitively, a leftward or rightward path should maximize this potential, and a perfectly balanced binary search tree (which has $2^k - 1$ nodes all with depth at most $k - 1$) should minimize it. I have encountered unexpected difficulties when trying to appropriately formalize this notion. My investigations into this matter are only preliminary, so I decline to comment further on what the potential might look like.
Figure 6.3: The access paths for $x$ and $y$ contain both crossing nodes and bookkeeping nodes. Every splay step when splaying $x$ is a zig-zig, and every splay step (except for the last) when splaying $y$ is a zig-zag.

The factor of two in Conjecture 6.3 is again derived from numerical experiments. In particular, it follows from the example showing Splay’s subsequence overhead in Chapter 3.2: the extra Splay steps when executing the subsequence are almost entirely zig-zigs. It seems possible that Splay’s subsequence overhead stems from mostly from its bookkeeping cost.

Remark 6.2. In [37] we referred to bookkeeping nodes as “zig-zig” nodes, and also referred to crossing nodes as “zig-zag” nodes. This was an unfortunate choice of terminology. There exist access paths that contain an arbitrary number of both crossing nodes and bookkeeping nodes, yet where the Splay steps are either entirely zig-zigs or entirely zig-zags. See Figure 6.3.

6.5 Similar Ideas

Some of the ideas proposed here resemble those which gave rise to Tango Trees [19]. This data structure was essentially constructed by turning Wilber’s other lower bound
(the “alternation” bound) into an algorithm. By comparing the executions of the algorithm with the known lower bound, Demaine et. al. were able to prove that Tango costs no more than $O(\log \log n)$ times the optimum for a tree of size $n$. Many ideas in this chapter essentially boil down to comparing Splay’s behavior with the crossing lower bound. In fact, the similarities between Tango and the ideas in this chapter run deeper. Tango trees actually began as attempt to turn the crossing bound into an algorithm. One can view this chapter as a second attempt to make use of the crossing bound, albeit in a very different way.
Chapter 7

Extensions and Comments

7.1 Insertion and Deletion

The transition-tree model can be extended to cover insertions and deletions. Start with an infinite binary tree where all nodes have two children. Every node is distinct, but initially no node has a key. Begin by populating the nodes at the top of this tree with the same keys as the initial binary search tree of the instance, \( T \). The nodes with keys will always be arranged in symmetric order as a connected subtree of the root of this infinite tree; this is called the visible tree. Transition trees must always be subtrees of the visible tree.

A request for key \( k \) in the visible tree is served in the same manner as Definition 2.1: choose some connected subtree \( Q \) of the visible tree \( T \) that contains both \( k \) and \( \text{root}(T) \), and replace it with binary search tree \( Q' \) with the same keys as \( Q \), where \( k = \text{root}(Q') \). Reattach the subtrees of \( Q \) to \( Q' \) in the same left-to-right order as they appear on \( Q \).

\[1\]While invisible nodes do not have keys, they have identities, so they can be properly reattached using the symmetric order of the nodes if not the keys.
To insert \( k \), perform a binary search in the visible tree and assign the key \( k \) to the first keyless node found by the search. The remainder of the insertion is treated as a regular access to \( k \), and \( k \) becomes the root of the tree.

To delete key \( k \), an execution produces a transition tree \( Q' \), where the \( k \)'s successor (the node with smallest key greater than \( k \)) is the root of \( Q' \), \( k \)'s predecessor is the root’s left child, and the node holding \( k \) is the predecessor’s right child. (If one of the successor or predecessor is not present, then the node holding \( k \) becomes the child of the other. Otherwise \( k \) becomes the root.) After substituting \( Q' \), the node holding \( k \) has its key removed and becomes invisible.

This definition of deletion is unusually strong. Typically, deletion is implemented by accessing the successor or predecessor, but not both. However, this version can be implemented using a subtree transformation. We leave the analysis of standard deletion algorithms as an open problem.

These representations of insertion and deletion are carefully constructed to be restricted versions of operations in the model of Definition 2.1. Since Splay can simulate arbitrary subtree transformations, it can induce transformations of the restricted form as well. Thus, if Splay is approximately monotone in the model of Definition 2.1 then it will also be constant competitive with \( \text{OPT} \) on request sequences that allow these operations.

### 7.2 The Deque Conjecture

Consider the following problem. Start with a tree \( T \) having integers 1 through \( n \) as elements, and consider a sequence of \( m \) of the following operations: delete the minimum key, delete the maximum key, insert a new minimum key, and insert a new maximum key. (These are called \textit{deque} operations.) These operations can be
performed with a total cost $O(m+n)$ using the above model of insertion and deletion, as follows.\(^2\)

Initially, the starting tree is reshaped so that the median key $x$ (or a key closest to the median) is the root, the nodes less than $x$ form a rightward chain as $x$’s left subtree, and the nodes greater than $x$ form a leftward chain as $x$’s right subtree. At any time, let $w$ and $z$ be the nodes with minimum and maximum key, respectively. To insert a new minimum (respectively maximum), rotate at $w$’s left child (respectively $z$’s right child) and assign this child a key that is less than the current minimum (respectively greater than the current maximum). To delete the current minimum (respectively maximum), remove the deleted node’s key, and then rotate at the right child of $w$ (respectively left child of $z$), if the child is present. If the child is not present then the tree’s visible nodes are rearranged in the same manner that was used to reshape the initial tree. The situation is depicted in Figure 7.1.

In [61], Tarjan considered whether the total cost of performing deque operations via splaying costs $O(m+n)$, calling this the \textit{deque conjecture}. Extrema insertion can be performed using insertion splaying. Deletion of the minimum (respectively maximum) key can be implemented by splaying the successor (respectively predecessor) and then removing the node with minimum (respectively maximum) key. By Theorems 3.7 and 4.1, if Splay is approximately monotone then these implementations of the deque operations will have total cost $O(m+n)$.

This does not entirely resolve the original question posed by Tarjan in [61]. Tarjan’s original Splay-based implementations of the deque operations fall outside of the insertion and deletion model of the previous section. We leave determining whether approximate monotonicity also implies that the original implementations satisfy the deque conjecture as an open problem.

\[^2\]We omit certain details about degenerate trees with fewer than three nodes. See [8, Lemma 36] for a more thorough description.
7.3 The Traversal Conjecture

A corollary of Theorem 4.1 is that if Splay is approximately monotone (and hence is dynamically optimal) then it satisfies the traversal conjecture, which is as follows. Let $T_1$ and $T_2$ be any two binary search trees on the same set of $n$ keys. The traversal conjecture [54] states that $\text{cost}(T_1, \text{preorder}(T_2)) = O(n)$. The traversal conjecture would be an immediate consequence of Splay being dynamically optimal with linear additive overhead, but would not necessarily follow from Splay being optimal with super-linear additive overhead. Theorem 4.1 removes the difference.

---

3Splaying preorder($T$) starting from $T$ takes linear time. (See Appendix B.1.) OPT can substitute any other tree for preorder($T$) with cost at most the size of the initial tree and then Splay.
7.4 Path-Based Algorithms

A natural generalization of Splay is what we refer to as a *path-based algorithm*. The transition trees of a path-based algorithm comprise nodes solely lying on the access path, and the transition trees’ shapes, up to relabelling of the keys, are determined entirely by the *binary encoding* of the access path. A reexamination of Chapters 3 and 4 reveals that all of the results in these chapters\(^4\) apply to any path-based algorithm whose transition graph \(G_n\) is strongly connected for some \(n \geq 3\). Any such algorithm will have the transformation property, and the overhead of the transformations (and hence simulations) will be bounded above by \(n\) times the diameter of the algorithm’s transition graph \(G_n\).

The transition graph of “Simple Splay,” described in [54], is also strongly connected. Hence, if Simple Splay is approximately monotone then it is dynamically optimal. Interestingly enough, the results of these chapters *also* apply to Move-to-Root, since Move-to-Root can be used to transform between three-node trees (proof omitted). It is easy to show that Move-to-Root is not optimal. Hence, by Theorem 3.7, Move-to-Root is not approximately monotone.\(^5\) If nothing else, this indicates Theorem 3.9 could be a useful tool for showing algorithms are *not* optimal.

The results of Chapters 3 and 4 do *not* immediately apply to the “Top-Down” variants of Splay, described in [54]. Top-Down Splay cannot be used to transform trees at linear cost. It is still possible to build a simulation embeddings for Top-Down Splay, however. (See Appendix A.3.)

Any *template based* algorithm of the types discussed in [57] and [23] whose transition graph \(G_n\) is strongly connected for some \(n \geq 3\) is dynamically optimal if and only if it is approximately monotone. We suspect, but have not tried to prove, that

---

\(^4\)With the sole exception of Theorem 3.10, which relies on Cole’s “Splay-specific” proof of the dynamic finger theorem.

\(^5\)If \(T\) is a leftward path of \(n\) nodes, \(X = (n, n-1, \ldots, 2, 1, 2, \ldots, n-1, n)\), and \(Y = (1, 2, \ldots, n)\) then the cost of executing Move-to-Root on the subsequence \(Y\) is \(\Theta(n)\) times the cost of executing Move-to-Root on the super-sequence \(X\).
there are simple sets of combinatorial criteria by which one can evaluate a template to determine if it has some strongly connected transition graph, and that a large sub-family of template algorithms will meet such criteria.

7.5 The Power of Simplicity

Reflecting on the observations about path-based algorithms, there seems to be a peculiar conflict in the question of dynamic optimality between the directions that one’s intuition might naturally lead.

On the one hand, the dynamic optimality conjecture is colloquially considered to be a “surprisingly strong” one. What is meant by this is that, compared to the “power” or “capabilities” available to an arbitrary execution in the BST Model, Splay is severely “restricted.” An optimum execution is offline, i.e. it sees all requests at once, and “plans” how to structure the transition trees accordingly. In contrast, Splay is an online algorithm, which cannot see the future. Splay is also history independent, merely adjusting the tree according to the given rules, without regard to previous requests. Perhaps more severely, Splay does not use any knowledge about the global structure of the tree, or the keys within it. It is, essentially, a very simple algorithm for splitting paths in half. The “simplicity” of Splay juxtaposed with the intricacy of the BST Model can elicit a visceral feeling that it would indeed be quite remarkable if the conjecture were true.

Yet, upon further reflection, it becomes very clear that the same properties that make Splay seem so restricted are exactly what enable the creation of relatively simple and explicit simulation embeddings. If, for example, an algorithm outputs a transition tree for request $x$ whose structure depends on the requests either preceding or following $x$, it could seriously complicate catenation of the key sequences that induce particular subtree transformations, since the transition tree for $x$ is altered by adjac-
cent transformations. In contrast, the effect of splaying $x$ depends on neither past nor future. Hence, transformation sequences for Splay can be linked together. Similarly, if an algorithm’s behavior depends on the tree structure as it exists outside the path, it is no longer possible to treat each subtree transformation cleanly and independently, as we could do when proving Theorem 3.6. In a very poignant sense, Splay’s simplicity and restrictiveness make its execution controllable. It seems unlikely that algorithms without these properties could be so easily coerced to simulate arbitrary executions.\(^6\)

Perhaps this strange duality, in which the most restricted algorithms are the easiest to “control,” and hence prove optimal, will serve as a guiding principle for building simulation embeddings in other models of computation. It does seem to resonate with peoples’ pervasive aesthetic preference for using “elegant” algorithms wherever possible.

\(^6\)As discussed in Chapter 3.6, Harmon builds a simulation embedding in [28] for GreedyFuture whose execution is history-dependent. That he proves this in a very different (although cost-equivalent) geometric model of computation where this algorithm’s behavior is far simpler to describe may well actually bolster our claim.
Concluding Remarks

I have spent a Ph.D. on this fascinating and beautiful conjecture. It remains open despite my best efforts, and the path forward does not appear easy. The time has come for me to put this problem down. My hope is that I have at least brought this problem from the realm of the unyielding to the realm of what is merely difficult. I leave this guide in the hopes that others may pick up where I left off, and forge new paths from Splay to dynamic optimality.

Thanks. I thank Luís Russo for suggestions that much improved Figure 3.2, Kurt Mehlhorn for supplying the simplified proof of Theorem 4.1, Amit Halevi for providing comments that greatly clarified Definition 2.1, and Siddhartha Sen for editorial feedback. I am indebted to John Iacono for his guidance in understanding the equivalence between Wilber’s bound and the crossing nodes of treaps, along with corroborating my empirical comparisons between the behaviors of Splay and Wilber’s bound. Finally, I am grateful to David Galles for making his “Data Structure Visualizations” available on his website.\(^7\) I spent many hours prototyping the ideas featured in this thesis using Galles’ visualizer. This would have been a lesser work without this publicly available tool.

\(^7\)https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
Appendix A

Generalized Transformations

A.1 Universal Transformations

This section covers a generalization of transformation sequences, called universal transformations. A universal transformation sequence $U$ for a binary search tree $Q$ with at least four nodes is a sequence of keys such that, if $T$ is any binary search tree whose keys are a superset of the keys in $Q$, then $Q$ will be a connected subtree of the root in the final after-tree, $T'$, produced by splaying the sequence $U$ starting from $T$. (The subtrees hanging from $Q$ in $T'$ may be arranged arbitrarily.) We stress that the sequence of $U$ is independent of the structure or number of keys in $T$, and comprises only keys in $Q$.

Our universal transformation sequences comprise three basic parts: a reverse sequential access of $Q$’s nodes, a “cleanup sequence”, and transformation of the left spine to $Q$. Let $q_1 < \cdots < q_{|Q|}$ be the keys in $Q$. (For convenience, we may assume $|Q| = 2k + 3$ for some integer $k > 0$.) Let $A(Q)$ denote the sequence of $Q$’s keys listed largest to smallest, and let $C(Q) = \mathbb{T}(L, Q)$ denote the transformation sequence that induces Splay to turn a left spine $L$ with the same keys
into \( Q \). Define the sequence valued function \( G(x, y, z) = (z, y, z, x, y, z) \), and let 
\[ B(Q) = \bigoplus_{i=1}^{(2|Q|-3)/2} G(q_{2i-1}, q_{2i}, q_{2i+1}). \]
Define \( U(Q) \equiv A(Q) \oplus B(Q) \oplus C(Q). \)

As the next theorem shows, \( U(Q) \) is a universal transformation sequence for \( Q \). Furthermore, the cost of executing this transformation is linear in the size of \( T \). More precisely:

**Theorem A.1.** Let \( Q \) be a binary search tree with \( |Q| = 2k + 3 \) for \( k > 1 \), and let \( T \) be a binary search tree containing the keys of \( Q \) as a subset. Denote by \( C_T(Q) \) the smallest connected subgraph of \( T \) that contains both root\((T)\) and the keys in \( Q \). If \( T' \) is the final after-tree from splaying \( U(Q) \) starting from tree \( T \), then

- \( |U(Q)| = O(|Q|) \), and
- \( C_{T'}(Q) = Q \), and
- \( \text{cost}(U(Q), T) = O(|C_T(Q)|) \).

**Proof.** The first part of the theorem is the simplest. The sequential access comprises of \( |Q| \) nodes. The cleanup sequence is a concatenation of \((|Q| - 3)/2\) sequences of length six, and the transformation portion comprises at most \( 20|Q| \) keys by Theorem 3.5.

The reverse sequential access arranges the nodes of \( Q \) in a rightward path at the top of the tree. However, between each successive pair \( q_i \) and \( q_{i+1} \) there will sometimes be a node from \( T \setminus Q \), which occurs whenever the last splay step when splaying \( q_i \) was a zig-zig. Whether this happens or not is a function of \( T \)'s structure, and cannot be anticipated by the transformation sequence. However, from Figure A.1 we can see that splaying the sequence \( G(q_1, q_2, q_3) \) ensures that \( q_1 \) becomes the left child of \( q_2 \) which becomes the left child of \( q_3 \) which becomes the root, regardless of whether or not there was an extra node from \( T \setminus Q \) between either, or both of the pairs \( \{q_1, q_2\} \) and \( \{q_2, q_3\} \) at the beginning of the splay operations. By repeating this process for each grouping \((q_{2i-1}, q_{2i}, q_{2i+1})\), the end result of splaying the cleanup sequence \( B(Q) \)
is to ensure that the nodes of \( Q \) are arranged as a connected subtree of the left spine. The final transformation sequence arranges this left spine into \( Q \). Hence, \( C_T(Q) = Q \).

Note that \( A(Q) \) is a subsequence of the nodes of \( C_T(Q) \) arranged in reverse symmetric order. Applying the dynamic finger theorem in a manner similar to that used for proving Theorem 3.10, we see that the cost of splaying \( A(Q) \) is \( O(|C_T(Q)|) \). By Theorem 3.5, the sequence \( B(Q) \) comprises \( 3|Q| - 9 \) splays, each one for a node with depth at most 4. Thus, splaying \( C(Q) \) takes \( O(|Q|) \) time.

It is somewhat remarkable that it is possible that it is possible to create the “cleanup sequence” \( B(Q) \) while only using keys from \( Q \) itself. We believe universal transformations might be useful for proving theorems related to multi-finger search trees [20].

### A.2 Simultaneous Transformations

Taking a closer look at Figure A.1, it seems like it should be possible to produce request sequences which cause two different algorithms to transform the same starting tree. This is indeed the case for Splay and Move-to-Root. We have declined to provide pictures of this process, which will look similar to Figure A.1. Instead, we supply instructions for inducing simultaneous four-node transforms for these algorithms.

Assume that the starting tree has four nodes whose keys are \( \{1, 2, 3, 4\} \). Whatever the starting arrangement, accessing the sequence \( (1, 2, 3, 4) \), both with Splay and with Move-to-Root, produces a left spine. Accessing the keys in reverse order produces a right spine. There are fourteen binary search trees with these four keys, and the above covers two of them. Each of twelve remaining trees comprising these four nodes has a symmetric variant.

The following six request sequences induce both Splay and Move-to-Root to produce the same distinct member from the twelve remaining trees when accessed start-
Figure A.1: Splaying the sequence $G(1, 3, 5)$ cleans up the path no matter what its state was when the splaying started.
ing from the left spine: \((3, 1, 4, 1), (2, 3, 4, 1, 2, 1), (2, 3, 4, 1, 3, 1), (2, 3, 4, 2, 4, 1), (2, 3, 4, 2, 4, 1, 2)\). None of the distinct trees so produced is a mirrored variant of one of the others. The reflections of these trees can be produced symmetrically by starting from the right spine, with the corresponding access sequences produced and substituting 4 for 1, 3 for 2, 2 for 3 and 1 for 4 in the sequences above. Thus, for every pair of binary search trees with the same keys, there is a request sequence that simultaneously induces Splay and Move-to-Root to transform between them.

By stringing these four node transformations together, it is possible to produce request sequences that induce both Splay and Move-to-Root to simultaneously transform one tree into another in linear time.\(^1\) We conjecture that an arbitrary number of “template algorithms” [57] can be made to transform simultaneously, with overhead depending on the particular structure of the templates.

### A.3 Top-Down Splay

Top-Down Splay is most easily defined in the *global view* [7]. Let \(p_0, \ldots, p_k\) be the reversed path going from \(x\) to the root. First execute Move-to-Root, and then rotate the edge connecting \(p_{2i}\) and \(p_{2i+1}\) whenever both nodes lie on the same side of \(x\). This is identical to (regular) Bottom-Up Splay, except that the index of rotation is offset by one. (See Chapter 6.1.) This change in “parity” makes Top-Down Splay identical to Bottom-Up Splay on paths of odd length, and different on paths of even length greater than two [43].

Sleator and Tarjan describe how to implement Top-Down Splay in a “single pass,” so that the Splay operations may be carried out as the binary search is performed [54]. Thus, Top-Down Splay can be implemented without parent pointers. (This idea was inspired by a method of Stephenson [56].)

---

\(^1\)It is not possible to produce universal simultaneous transforms, since Move-to-Root does not have the sequential access (and hence dynamic finger) property.
Theorem A.2. The transition graph, $G_n$, for Top-Down Splay is not strongly connected for any $n \geq 3$.

Proof. For $n = 3$, one cannot go from a left spine to zig-zag shaped tree. Let $n > 4$, let $Q$ have integer keys $\{2, \ldots, n - 1\}$, and let $S$ be the tree with keys $\{1, 2, \ldots, n\}$ whose root key is 1, where the right child of root($S$) key $n$, and where $n$’s left subtree in $S$ is $Q$. Let $T$ be formed by top-down splaying some $j \in \{2, \ldots, n\}$ in $S$. We show that no sequence of top-down splays can restore $S$ to $T$.

Top-Down Splay, like Splay, always places the node with the requested key at the root. Thus, root($T$) = $j$. Because root($S$) = 1, the last key in any sequence of requests that induce Top-Down Splay to transform $T$ into $S$ must also be 1. Suppose, for the sake of contradiction, that the key top-down splayed prior to the final access to 1 is $k \in \{2, \ldots, n - 1\}$, so that $k$ is the root when 1 is splayed, and let $R$ be the tree immediately prior to the final top-down splay for 1. Let $R'$ be formed by performing a top-down splay for 1 in $R$. We show $R' \neq S$, as follows.

Because $1 < k < n$, the key 1 is in the left subtree of $k$ in $R$, and the key $n$ is in the right subtree of $k$. Thus, $n$ is not on the access path for 1 in $R$. After a top-down splay operation, the access path to the splayed node remains a connected subtree of the root of the after-tree. Because $k$ is the largest node on the access path for 1 in $R$, the access path for $k$ in $R'$ comprises a series of right pointers from 1. Similarly, $n$, as the maximum key, is always the right-most node in the tree; i.e. $n$ is always found by starting from the root and following right pointers until reaching the node with null right child. Thus, $k$ is found on the path connecting 1 to $n$ in $R'$. This is incompatible with $n$ being the right child of 1 in $S$. Thus, $R' \neq S$.

The only remaining possibility is that the final two keys in a sequence that induces top-down splay to restore $T$ to $S$ must be $n$ followed by 1. The reader can verify by hand that top-down splaying a four-node left spine does not achieve the desired effect, and the case for a left spine with three nodes follows from above. By induction on the
length of the left spine, the final two top-down splay operations will not transform the tree into the desired state. Hence, there is no path from $T$ to $S$ in $G_n$, and $G_n$ is not strongly connected.

**Remark A.1.** This issue does not arise for Bottom-Up Splay because splaying a four-node left spine makes $1$ the root and $n$ the right child of the root.

### A Simulation Embedding for Top-Down Splay

Despite the impossibility of employing Top-Down Splay to directly transform between subtrees, it remains possible to build a simulation embedding for Top-Down Splay with only minor modifications to the simulation embedding produced in Chapter 3.2.

Let $Q_1, \ldots, Q_m$ be the subtrees, $Q'_1, \ldots, Q'_m$ be the transition trees, and $T_1, \ldots, T_m$ be the after-trees of an optimal execution of the request sequence $X$ starting from initial tree $T = T_0$ comprising integer keys $\{1, \ldots, n\}$.

For $1 \leq i \leq m$, form $T'_i$ by removing $1$ from $T_i$ and setting right-child$_{T_i}(1)$ to be the left child of parent$_{T_i}(1)$ if the parent is present. Form $T''_i$ from $T'_i$ by removing $2$ from $T'_i$ and setting right-child$_{T'_i}(2)$ to be the left child of parent$_{T'_i}(2)$ if the parent is present. Form $T'''_i$ from $T''_i$ by removing $n$ from $T''_i$ and setting left-child$_{T''_i}(n)$ to be the right child of parent$_{T''_i}(n)$ if the parent is present. Form $\hat{T}_i$ from keys $\{1, \ldots, n\}$ by making root($\hat{T}_i$) = $n$, setting left-child$_{\hat{T}_i}(n)$ = $2$ and left-child$_{\hat{T}_i}(2)$ = $1$, and making $T'''_i$ the right subtree of $2$.

Let $\hat{Q}_i = C_{\hat{T}_i-1}(Q_i \cup \{1\})$ denote the smallest connected subtree of the root of $\hat{T}_i$ containing keys $Q_i \cup \{1\}$, and similarly let $\hat{Q}'_i = C_{\hat{T}_i}(Q'_i \cup \{1\})$. Note that $|\hat{Q}'_i| = |\hat{Q}_i| \leq |Q_i| + 3$. It is possible to use Top-Down Splay to perform restricted rotations on the nodes $\{3, 4, \ldots, n - 1\}$ in each $\hat{T}_i$, see Figure A.2. Thus, Top-Down Splay can be induced to substitute each $\hat{Q}'_i$ for $\hat{Q}_i$ in time $O(|Q_i|)$.

Transforming $T_0$ into $\hat{T}_1$ can be done by first accessing $(n, 2, 1, n)$ at cost less than $4|T_0|$ and then performing one of the above transforms on the resultant tree,
Figure A.2: Schematic of the top portion of the augmented trees $\hat{T}$. To induce a rotation at $u$, perform top-down splays at $(a, u, a, z)$. To induce a rotation at $v$, top-down splay the sequence $(v, a, z)$. To induce a rotation at $w$, top-down splay at $(a, w, a, z)$. To induce a rotation at $y$, top-down splay at $(y, a, z)$.

for a total initial cost of $O(|T|)$. This can be absorbed into the simulation cost due to Remark 2.2. Stringing together the transformation sequences for each subtree substitution produces a simulation embedding for Top-Down Splay. To conclude:

**Theorem A.3.** If Top-Down Splay is approximately monotone then it is dynamically optimal.
Appendix B

Preorders and Postorders

Remark B.1. This section is largely based off of material that will appear in [38].

An auxiliary question to determining if Splay (or any other algorithm) is dynamically optimal is: “what class(es) of access patterns have optimum executions with ‘low’ cost?” This issue is not a mere curiosity, as almost every permutation of length $n$ has optimal execution cost $\Theta(n \log n)$ [36], a bound achieved by any balanced search tree. Thus, in the absence of insertions or deletions, adjusting the tree after every access only gives an advantage on a small subset of “structured” request sequences. In addition, these structured request sequences provide candidate counter-examples to dynamic optimality.

Preorders and postorders are a well-studied family of structured request sequences. For a tree $T$ of size $n$, I prove the following results about the behavior of splaying preorders and postorders:

1. Inserting the nodes of preorder($T$) into an empty tree via splaying costs $O(n)$.

2. Inserting the nodes of postorder($T$) into an empty tree via splaying costs $O(n)$.

3. If $T'$ has the same keys as $T$ and $T$ is weight-balanced [46] then splaying either preorder($T$) or postorder($T$) starting from $T'$ costs $O(n)$. 
For items 1 and 2, we use the fact that preorders and postorders are pattern-avoiding: i.e. they contain no subsequences that are order-isomorphic to (2, 3, 1) and (3, 1, 2), respectively. Pattern-avoidance implies certain constraints on the manner in which keys are inserted. I exploit this structure with a simple potential function that counts inserted nodes lying on access paths to uninserted nodes. These methods can likely be extended to permutations that avoid more general patterns. The proof of Item 3 uses the fact that that preorders and postorders of balanced search trees do not contain many large “jumps” in symmetric order, and exploits this fact using the dynamic finger theorem \[14, 13\].

Items 2 and 3 are both novel. Item 1 was originally proved by Chaudhuri and Höft \[9\]; my proof simplifies theirs.

**Pattern Avoidance**

Two permutations \(\alpha = (a_1, \ldots, a_n)\) and \(\beta = (b_1, \ldots, b_n)\) of the same length are order-isomorphic if their entries have the same relative order, i.e. \(a_i < a_j \iff b_i < b_j\). For example, \((5, 8, 1)\) is order-isomorphic to \((2, 3, 1)\). A sequence \(\pi\) avoids a sequence \(\alpha\) (or is called \(\alpha\)-avoiding) if it has no subsequence that is order-isomorphic with \(\alpha\). If \(\pi\) is \(\alpha\)-avoiding then all subsequences of \(\pi\) are \(\alpha\)-avoiding. We use \(\pi \setminus \alpha\) as shorthand for “an (arbitrary) permutation \(\pi\) that avoids \(\alpha\).” Both preorders and postorders may be characterized as pattern-avoiding permutations:

**Lemma B.1** (Lemma 1.4 from \[35\]). For any permutation \(\pi\):

1. \(\pi = \text{preorder}(T)\) for some binary search tree \(T\) \(\iff\) \(\pi\) avoids \((2, 3, 1)\).

2. \(\pi = \text{postorder}(T)\) for some binary search tree \(T\) \(\iff\) \(\pi\) avoids \((3, 1, 2)\).

For preorders, Kozma builds a bijection between binary search trees and \((2, 3, 1)\)-avoiding sequences, and uses a simple argument by contradiction to show preorders
avoid (2, 3, 1) [35]. The proof for postorders is a nearly symmetric variation of this argument.

Related Work

The first result about Splay’s behavior on pattern-avoiding request sequences was the sequential access theorem [61]: the cost of splaying the nodes of $T$ in order is $O(|T|)$. This is a special case of the traversal conjecture (when the preorder is formed from a rightward path). Theorem B.1 (originally [9]) is another special case: when $T_1 = T_2$. In Section B.3 we prove a new special case: when splaying the preorder of a tree that is $\alpha$-weight balanced [46]. (The initial tree does not matter.)

Interest in the behavior of binary search tree algorithms on “structured” request sequences was revived by Seth Pettie’s analysis of the performance of Splay-based deque data structures using Davenport-Schinzel sequences [47], and his later reproof of the sequential access theorem via the theory of forbidden submatrices [48]. This analysis was later adapted to Greedy. In [6], Chalermsook et. al. show that Greedy has nearly-optimal run-time on a broad class of pattern-avoiding permutations. Moreover, they demonstrate that if Greedy is optimal on a certain class of “non-decomposable” permutations then it is dynamically optimal. Chalermsook et al.’s analysis was later simplified in [25].

Insertion Splay

Lemma B.2. If $x$ is a proper ancestor of $y$ in $\text{BST}(\pi)$ then $x$ precedes $y$ in $\pi$.

Proof. Let $\pi_{<y}$ denote the prefix of $\pi$ containing the elements preceding $y$. By construction, $y$ is inserted as a child of some node $z$ in $\text{BST}(\pi_{<y})$. Every proper ancestor of $y$ is an ancestor of $z$, thus $x \in \text{BST}(\pi_{<y})$. Hence, $x$ precedes $y$. \qed

77
Insertion splaying $\pi$ has the same cost as splaying $\pi$ starting from $\text{BST}(\pi)$.\footnote{This is because the manner in which Splay restructures the access path is independent of nodes outside the path.} For the purposes of analysis, assume that every node in $\text{BST}(\pi)$ is initially marked as untouched. An insertion splay marks the node as touched, and then splays the node. The touched nodes form a connected subtree containing the root, called the touched subtree. The untouched nodes form subtrees, each of which contains no touched node. Call an untouched node with a touched parent a sub-root. The subtrees rooted at sub-roots have identical structure in both the splayed tree and $\text{BST}(\pi)$. By Lemma B.2, the next node to be touched is always a sub-root.

For $1 \leq i \leq n$, form $T_i$ by touching and then splaying $p_i$ in $T_{i-1}$, where $T_0 = \text{BST}(\pi)$ starts with all nodes untouched. At any time, the potential is defined to be twice the number of touched nodes that are ancestors of sub-roots, and the potential of $T_i$ is denoted $\Phi_i$. The amortized cost of splaying $p_i$ in $T_{i-1}$ is defined as $c_i = t_i + \Phi_i - \Phi_{i-1}$, where $t_i$ denotes the actual cost. By a standard telescoping sum argument, the cost of insertion splaying $\pi$ is $\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} c_i + \Phi_0 - \Phi_n$ [60]. Since $\Phi_0 = \Phi_n = 0$, an upper bound on amortized cost provides an upper bound on the actual cost.

Pattern-avoidance provides certain information about both $\text{BST}(\pi)$ and about which sub-root can be touched next. We exploit this information in the next two sections.

**B.1 Inserting Preorders**

There are no restrictions on the possible structure of preorder insertion trees, as $\text{BST}(\text{preorder}(T)) = T$.\footnote{In fact, this property is shared by any permutation $\pi$ for which every node in $T$ appears in $\pi$ before those in its left and right subtrees.} However, the manner in which sub-roots are chosen is particularly simple.
Figure B.1: Possible locations for the next sub-root $x$ to be insertion splayed in $\pi \setminus (2, 3, 1)$. The case on the left occurs when the next splayed node has left-depth 0, and the case on the right occurs when it has left-depth 1. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.

**Lemma B.3.** If $\pi \setminus (2, 3, 1) = (p_1, \ldots, p_n)$ is a preorder then, for $1 \leq i \leq n$, $p_i$ is the smallest sub-root of $T_{i-1}$, where all nodes begin untouched in $T_0 = \text{BST}(\pi)$ and $T_i$ is formed by touching and splaying $p_i$ in $T_{i-1}$.

**Proof.** The statement is vacuously true for $i = 1$. We prove for $i > 1$ by contradiction, as follows. Suppose $T_{i-1}$ has some sub-root $q$ that is smaller than $p_i$. Since $q$ and $p_i$ are both sub-roots in $T_{i-1}$, they are both children of respective (though not necessarily distinct) nodes $a$ and $b$ in $T_{i-1}$. Let $r = \text{lca}_{T_{i-1}}(a, b)$. Since $q \neq a$ and $p_i \neq b$, all of $p_i$, $q$ and $r$ are distinct nodes in $T_{i-1}$, and furthermore $q < r < p_i$. By Lemma B.2, $r$ precedes both $q$ and $p_i$ in $\pi$, and by construction $p_i$ precedes $q$. Thus, $(r, p_i, q)$ is a subsequence of $\pi$. But $(r, p_i, q)$ is order-isomorphic with $(2, 3, 1)$, contradicting $\pi \setminus (2, 3, 1)$. \hfill $\square$

**Lemma B.4.** Insertion splaying preorder($T$) keeps each sub-root at left-depth at most 1 and takes $O(1)$ amortized time per splay operation.
Proof. The theorem is trivial for the first insertion splay. The inductive hypothesis is that every sub-root has left depth 0 or 1. Let $x$ be the next sub-root to be splayed, and let $y$ and $z$ (either or both of which can be missing) be its left and right children. Touching $x$ makes $y$ and $z$ into sub-roots.

Suppose $x$ has left depth 0 before it is touched. Converting $x$ from untouched to touched (without splaying it) increases the potential by at most 2 and gives the new sub-roots $y$ and $z$ left depths of 1 and 0, respectively. (In this case they are the only two sub-roots.) Each splay step, except possibly the last, is a zig-zig in which $x$ starts as a left child with parent $p$ and grandparent $g$. After completing the zig-zig, $g$ is no longer an ancestor of any untouched node, which decreases the potential by 2. The zig-zig also preserves the left depths of $y$ and $z$. ($y$ becomes the right child of $p$.) No other sub-roots can increase left-depth, as $x$ is the smallest sub-root. If the last splay step is a zig, the potential does not change (although the length of the path to $y$ increases by 1).

More complicated is the case in which $x$ has left depth 1. Converting $x$ from untouched to touched (without splaying it) makes $y$ a sub-root of left depth 2 and $z$ a sub-root of left depth 1. Let $w$ be the parent of the ancestor of $x$ that is a left child. All other sub-roots are in the right subtree of $w$, which is unaffected by splaying $x$. The splay of $x$ consists of 0 or more left zig-zigs, followed by a zig-zag (which can either left-right or right-left), followed by zero of more left zig-zigs, followed possibly by a zig. Each zig-zig reduces the potential by 2 and preserves the left depths of all sub-roots. The zig-zag does not increase the potential, reduces the left depth of $y$ from 2 to 1, and that of $x$ from 1 to 0, and preserves the left depth of $z$. Now $x$ has left depth 0, and the argument above applies to the remaining splay steps.

By Lemma B.3, the next node to be splayed will be $y$ if present, otherwise $z$ if present, otherwise $w$ if present. All three of these nodes have left-depth 0 or 1, hence an identical form to Figure B.1. Thus the hypothesis holds.
Observe that converting $x$ from untouched to touched increases the potential by 2. Each zig-zig step pays for itself: it involves two distinct nodes, each one paid for by the potential decreasing by at least 2. The zig-zag requires 2 rotations, and the zig requires 1 rotation. If the cost of a splay is the number of nodes on the splay path, equal to the number of rotations plus 1, then the amortized cost is 6 per splay. □

**Theorem B.1.** If $\pi \setminus (2, 3, 1)$ is of length $n$ then $\text{cost}(\pi, \text{BST}(\pi)) \leq 6n$.

*Proof.* Repeated application of Lemma B.4. □

### B.2 Inserting Postorders

Postorder insertion trees are more restricted. A binary search tree $C$ is a *(left-toothed)* comb if the access path for $x \in C$ always comprises some number $j \geq 0$ of right children followed by some number $k \geq 0$ of left children. The nodes of $C$ are partitioned into teeth, where every node in the $i^{th}$ tooth has right-depth $i - 1$. The shallowest node in a tooth is called the head. The insertion trees of postorders are combs:

**Lemma B.5.** If $\pi$ is a postorder then no left child in $\text{BST}(\pi)$ has a right child.

*Proof.* By contradiction. Let $y$ be a left child in $\text{BST}(\pi)$ with right child $z$, and let $x = \text{parent}(y)$. As $z$ is $y$’s right child, $y < z$. Similarly, as both $y$ and $z$ are in $x$’s left subtree, $y < z < x$. By Lemma B.2, $y$ can be an ancestor of $z$ only if $y$ precedes $z$ in $\pi$, and similarly $x$ must precede $y$. Thus, $(x, y, z)$ is a subsequence of $\pi$ that is order-isomorphic to $(3, 1, 2)$. By Lemma 2, $\pi$ is not a postorder. □

While postorder insertion trees are less varied than for preorders, there may be many postorders with a given insertion tree. This affords some amount of freedom for choosing different sub-roots.
Lemma B.6. Let $\pi \backslash (3,1,2) = (p_1, \ldots, p_n)$ be a postorder with insertion tree sequence $T_0, T_1, \ldots, T_n$. For $1 \leq i \leq n$, $p_i$ is either:

(a) The single sub-root greater than $\max T_{i-1}$ (if present), or

(b) The largest sub-root smaller than $\max T_{i-1}$ (if present).

Proof. The result is vacuous for $i = 1, 2$. If $p_i$ is case (a), we merely note that if $p_i$ is a new maximum then it must be the right child of the largest node in $\max T_{i-1}$. There can be at most one sub-root in this position. Hence, $p_i$ is unique.

For the sake of contradiction, suppose $p_i$ is not of the form in case (a) or (b), and let $q$ be the largest sub-root smaller than $\max T_{i-1}$. By Lemma B.2, the keys of each tooth are inserted in decreasing order. As $q$ is not the head of its tooth, its successor $r$ must be in $T_{i-1}$, and furthermore $r$ precedes both $p_i$ and $q$ in $\pi$. By construction, $(r, p_i, q)$ is a subsequence of $\pi$. Yet this subsequence is isomorphic to $(3, 1, 2)$ since $p_i < q < r$, contradicting Lemma 2. 

Lemma B.7. Insertion splaying postorders maintains the following invariants:

1. After each insertion splay, the path to every sub-root comprises $j \geq 0$ left pointers followed by $k \geq 0$ right pointers. (Furthermore, after the first insertion, $k \geq 1$.)

2. The left-depth of every sub-root decreases from smallest to largest.$^3$

3. The splay operation takes constant amortized time.

Proof. The base case is trivial. Lemma B.6 dictates that the next splayed sub-root is either greater than those of all marked nodes, or it is the largest sub-root smaller than the root. Let $x$ be the next node to be insertion splayed, $y$ its left child, and $z$ its right child (either or both children may be missing).

---

$^3$The first two invariants dictate that the ancestors of sub-roots form a right-toothed comb.
Figure B.2: Possible locations for the next sub-root $x$ to be insertion splayed in $\pi \setminus (3, 1, 2)$. The case on the left occurs when the next splayed node is less than the root, and the case on the right occurs when the next sub-root is the new tree maximum. Dashed nodes may or may not be present, and any number of nodes may lie on the paths denoted by dashed lines.

Suppose $x$ is greater than the current tree root. Marking $x$ increases the potential by 2 and makes $y$ and $z$ new sub-roots. The splay operation brings $x$ to the root by a sequence of left zig-zigs followed possibly by a left zig (depending on whether the length of the access path is odd or even). After each one of these zigs or zig-zigs, $y$’s left-depth remains 1, and $z$’s left depth remains 0. Let $v$ be the root prior to the splay operation. If the last splay step is a zig then the last splay operation increases the left depth of $v$ and everything in its left subtree by either 1 or 2. Since the left-depth of $x$ was 0 and $x$ was the largest sub-root, the inductive hypothesis ensures that all sub-roots had left-depth at least 1 before the splay operation, and therefore at least 2 afterward. Thus, when $x$ becomes the root, the left-depths of each sub-root decrease from left to right.

Otherwise, $x$ largest sub-root less than the tree root. Marking $x$ again increases the potential by at most 2. By Lemma B.5, $x$ has no right child (see Figure B.2), so we only need to worry about its left child $y$. Let $w$ be the last ancestor of $x$ that is
left child. Each left zig-zig prior to the splay step involving \( w \) maintains the left-depth of \( y \) to be one greater than the left-depth of \( x \). The splay step involving \( w \) will either be a left zig-zig or a left-right zig-zag, depending on the length of the original path connecting \( w \) to \( x \). Regardless, immediately after the splay step involving \( w \), the ancestor of \( y \) that is the left child of \( x \) is either the left child of \( w \) or the left child of \( w \)'s parent. Since all the sub-roots less than \( y \) are in the left subtree of \( w \), and thus have left-depth greater than the left-depth of \( y \), the invariant is restored, and remains true after each right zig-zig or zig that brings \( x \) to the root.

All that remains is showing constant amortized time. As noted before, marking \( x \) costs 2. If \( x \) is greater than the root then each left zig-zig, except possibly the last, pays for itself, giving amortized cost of 4. In the other case, all splay steps except for the one involving \( w \) and the one making \( x \) the root pay for themselves, giving amortized cost at most 6.

\[ \text{Theorem B.2. If } \pi \setminus (3, 1, 2) \text{ has length } n \text{ then } \text{cost}(\pi, \text{BST}(\pi)) \leq 6n. \]

\[ \text{Proof. Repeated application of Lemma B.7.} \]

\[ \text{□} \]

### B.3 Balanced Trees

Let \( |x| \) denote the size of the subtree rooted at \( x \). Following [46], we say \( T \) is \( \alpha \) weight balanced for \( \alpha \in (0, 1/2] \) if \( \min\{|\text{left-subtree}(x)|, |\text{right-subtree}(x)|\} + 1 \geq \alpha \cdot (|x| + 1) \) for all \( x \in T \), and write \( T \in \text{BB}[\alpha] \).

\[ \text{Theorem B.3. For any (fixed) } 0 < \alpha \leq 1/2, \text{ if } S \in \text{BB}[\alpha] \text{ and } T \text{ has the same keys as } S, \text{ then the cost of splaying preorder}(S) \text{ or postorder}(S) \text{ starting from } T \text{ is } O(|T|). \]

\[ \text{Proof. By Theorem 2.1, it suffices to show that } \text{DF}_T(\text{preorder}(S)) = O(|T|). \text{ Let } \]

\[ A_\alpha(n) \equiv \max\{\text{DF}_T(\text{preorder}(S)) \mid S \in \text{BB}[\alpha] \text{ and } |T| = n\}. \]
Recall that preorder(S) = (root(S)) ⊕ preorder(L) ⊕ preorder(R), where L and 
R are the left and right subtrees of the root of S, respectively. Notice that the rank 
differences between root(S) and the first key in preorder(L), and between the last key 
in preorder(L) and the first key in preorder(R), are at most |T| by definition. Hence,

$$DF_T(\text{preorder}(S)) \leq DF_T(\text{preorder}(L)) + DF_T(\text{preorder}(R)) + 2 \log_2(|T| + 1).$$

Observe that $$(|L| + 1)/(|S| + 1) \in [\alpha, 1 - \alpha]$$ since $S \in \text{BB}[\alpha]$, and by definition 
$|R| < |S| - |L|$. Hence,

$$A_\alpha(n) = \max_{\alpha \leq \beta \leq 1/2} \{A_\alpha(\beta \cdot n) + A_\alpha((1 - \beta) \cdot n)\} + O(\log n).$$

Akra-Bazzi’s result [2] suffices to show $A_\alpha(n) = O(n)$ for fixed $\alpha$. The proof for 
postorders is identical.

**Remark B.2.** In actuality, $A_\alpha(n) = O(f(\alpha) \cdot n)$ for some function $f$ of $\alpha$. Unfor-
tunately, the computation appears to be messy. I have declined to do the necessary 
footwork. Given the other results in this thesis, we strongly suspect that, regardless, 
$A_\alpha(n)$ does not tightly bound the cost of splaying these sequences.

**Remark B.3.** This result extends to any binary search tree algorithm that satis-
ifies the dynamic finger bound. Iacono and Langerman proved Greedy also has the 
dynamic finger property [32]; their analysis does not consider initial trees, however.

### B.4 Thoughts on $k$-Increasing Sequences

Patterns that avoid $(2, 1, 3)$ are “symmetric” to those that avoid $(2, 3, 1)$: if $\pi \setminus (2, 1, 3)$ 
then $\pi$ is the preorder of the mirror image of BST($\pi$). Similarly, patterns that avoid

---

4Technically, since $|L|/|S| < (|L| + 1)/(|S| + 1)$, we need to pick $S$ sufficiently large for a given alpha, and offset the recurrence term by a corresponding constant. This does not asymptotically affect the result.
(1, 3, 2) are symmetric to patterns that avoid (3, 1, 2). Thus, insertion splaying $\pi \setminus (2, 1, 3)$ and $\pi \setminus (1, 3, 2)$ takes linear time.

The only other patterns of length three are (3, 2, 1) and its symmetric counterpart (1, 2, 3). The pattern (3, 2, 1) was explored in [6], where it was shown that Greedy executes (3, 2, 1)-avoiding permutations in linear time starting from an arbitrary tree. In fact, they showed that executing $\pi \setminus (k, \ldots, 2, 1)$ takes time proportional to $n \cdot 2^{O(k^2)}$; this is linear in $n$ for fixed $k$. These permutations are called $k$-increasing because they can be partitioned into $k - 1$ disjoint monotonically increasing subsequences [6]. They form the natural generalization of sequential access, which is the (unique) permutation of the tree nodes that avoids (2, 1).

More general invariants can be derived about insertion tree structure and sub-root insertion order based on pattern-avoidance. As one particularly interesting example:

**Theorem B.4.** If $\pi \setminus (k, \ldots, 2, 1)$ then no node in BST($\pi$) has left-depth more than $k - 2$, and the next sub-root inserted (without splaying) is always the smallest sub-root with its given left-depth.

The proof is similar to Lemmas B.5 and B.6. In particular, the insertion trees of (3, 2, 1)-avoiding permutations look like the combs of postorder insertion trees, except the teeth are rightward, instead of leftward paths.

For $k$-increasing sequences, the potential used for Theorems B.1 and B.2 needs modifications. The main issue is that in both of these cases, the zig-zigs paid for themselves because the nodes knocked off the access path did not have sub-root descendants. This structure no longer holds for (3, 2, 1)-avoiding sequences, since we must splay the nodes of the teeth in increasing order. The proof seems to require a generalization of the sequential access theorem. It is possible that the notion of kernel trees used by Sundar in [58] for a potential-based proof of the sequential access theorem could be useful.
Appendix C

Folklore

C.1 Rotation-Based BST Model

Sleator and Tarjan originally defined binary search tree executions in a different but equivalent manner to the transition tree model (up to constant factors). In [54], an execution $R$ for request sequence $X = (x_1, \ldots, x_m)$ and initial tree $T_0 = T$ is as follows. For each request, $1 \leq i \leq m$, perform some number $e_i \geq 0$ of rotations starting from $T_{i-1}$ to form $T_i$, followed by a search for $x_i$ in $T_i$. The cost of this execution is $\sum_{i=1}^{m} (1 + e_i + d_{T_i}(x_i))$, and the optimum cost $\text{OPT}_{st}(X,T)$ in this model is the minimum possible cost execution for $X$ starting from $T$.\footnote{In fact, Sleator and Tarjan allowed the optimum algorithm to choose the initial tree for free. However, this can only save an additive $O(|T_0|)$ in the cost. (See Remark 2.3.)}

It is folklore knowledge that $\text{OPT}(X,T) = O(\text{OPT}_{st}(X,T))$. Pieces of the proof have appeared throughout the literature, but none complete enough for our liking. Thus, we provide a detailed proof here.

**Theorem C.1.** $\text{OPT}(X,T) \leq 4 \text{OPT}_{st}(X,T)$.

**Proof.** For every execution $R$ of $(X,T)$ in the rotation-based model, we describe how to build a corresponding transition-tree based execution $E$ with at most four times
the cost of $R$, and point to relevant literature along the way. The theorem follows by applying this transformation to an optimal rotation-based execution.

Wilber was the first to notice that we can require $x_i$ to be the root at the time the search is performed [64]. An arbitrary execution $R$ in Sleator and Tarjan’s model can be simulated by an execution $R'$ obeying this restriction, as follows. For each request $i$, perform the same rotations as $R$ in order to bring the tree’s shape to $T_i$. However, prior to the search, perform $d_{T_i}(x_i)$ rotations at $x_i$ to bring it to the root, creating tree $T_i'$. The search for $x_i$ takes place in $T_i'$. Prior to enacting the rotations of $R$ for $x_{i+1}$, first perform $d_{T_i}(x_i)$ more rotations to place $x_i$ back in its original position. Applying this process to all of the requests gives $R'$ an additive extra cost that is less than $\sum_{i=1}^{m} d_{T_i}(x_i)$. This is at most a multiplicative factor of 2 overhead compared to $R$. One consequence of this reduction is that the search cost can be absorbed into the rotation cost. This makes it possible re-define the execution cost to be the number of requests plus the number of rotations.

Lucas observed that, without loss of generality for executions obeying Wilber’s restriction, the nodes featured in rotations during an access will form a connected subtree of the root at the start (and hence end) of said access [40]. This is intimated to be a consequence of the following observation. If $w$, $x$, $y$ and $z$ are four distinct nodes in a binary search tree, such that $w$ is the parent of $x$ and $z$ is the parent of $y$, then rotating first at $x$ and then at $y$ will produce an identical tree to rotating first at $y$ and then at $x$. In other words, rotating disjoint edges in the tree is a commutative operation. Koumoutsos adds more detail [34]: “It is clear that all edges in the access path are rotated, as we are forced to move the next requested element to the root. Moreover, if two edges do not share an endpoint, then rotating them in either order produces the same tree. This implies that any rotation of an edge that does not belong into a connected component including the access path could be delayed until
immediately after the access of the requested element at the root, without any change in the cost of the algorithm."

More precisely, let \( \{u_1, v_1\}, \ldots, \{u_r, v_r\} \) be the sequence of rotations between nodes \( u_1, \ldots, u_r \) with respective parents \( v_1, \ldots, v_r \) over the course of execution \( R' \). We will produce a modified execution \( \hat{R}' \) costing at most as much as \( R \) where the nodes of edges that are rotated\(^2\) between the first and second searches of \( \hat{R}' \) form a connected subtree of the initial tree. To produce an execution \( R'' \) obeying both Wilber’s and Lucas’ restrictions, simply apply this procedure to the rotations before each subsequent request.

First, observe that rotating an edge inside a connected subgraph of a binary tree will not disconnect the subgraph. To produce the modified execution \( \hat{R}' \), we keep track of the maximal disjoint connected sets of nodes featured in overlapping rotations. Initially, every node is in a singleton set. Let \( k \) be the index of the last rotation of \( R' \) prior to the second search.\(^3\) For \( 1 \leq i \leq k \), unite the sets containing \( u_i \) and \( v_i \). Let \( Q \) be the set containing \( x_1 \). Move all rotations whose nodes are not contained in \( Q \) and place them immediately after the second search. (If there is no second search, these rotations can simply be dropped.) By construction, all such rotations commute with rotations between nodes in \( Q \). The nodes of \( Q \) are connected in both the initial tree and prior to the second search, and Wilber’s restriction ensures this set contains the root of the initial tree.

Finally, the transition tree model entirely eliminates rotations and searches from the description of binary search tree executions. Starting with \( R'' \), the sequence of rotations corresponding to access \( i \) can be represented by the subtree \( Q_i \) and transition tree \( Q'_i \) that the rotations for that access implicitly define. Each rotation involves at most 2 distinct nodes in \( Q \). Hence, the size of \( Q \) is at most twice the number of rotations for the access, plus one. This idea is implicit in [40], and Harmon supplied

\(^2\)A rotation at \( x \) rotates the edge between \( x \) and its parent.
\(^3\)If there is no other search, then \( k = r \).
details in [28]. The cost of execution $E$ is at most $2r + m$, which is less than four times the cost of $R$. \hfill \square

**Theorem C.2.** $\text{OPT}_{\text{st}}(X,T) \leq 3 \text{OPT}(X,T)$.

**Proof.** For each transition tree execution $E$ with subtrees $Q_1, \ldots, Q_m$, transition trees $Q'_1, \ldots, Q'_m$, and after-trees $T_1, \ldots, T_m$, we produce a rotation-based execution $R$ with cost at most three times that of $E$.

There are numerous ways to transform $Q$ using $O(|Q|)$ rotations. Perhaps the first of these was [16]. Sleator et al. showed that at most $2n - 6$ rotations suffice to transform between two binary search trees with the same $n$ keys [55].

Let $R$ be a sequence of rotations that enacts the same subtree transformations as $E$ using the method in [55], so that $R$ contains at most $2(|Q_1| + \cdots + |Q_m|)$ rotations. For $1 \leq i \leq m$, the search for $x_i$ in $R$ takes place immediately after the transformation of $Q_i$ into $Q'_i$, at which point $d_{T_{i-1}}(x_i) = 0$. The total cost of $R$ is at most twice the cost of $E$ plus $m$. By Remark 2.2, the additive $m$ may be absorbed into the the cost of $E$, for a total cost three times $E$. \hfill \square

## C.2 Sequences of Given Optimal Cost

This section may be viewed as a simplification of [8, Theorem 48]. The goal is to show that for any $n$ and any $2 < k < \log n$ there exists a permutation $X$ of $n$ keys for which $\text{OPT}(\pi) = \Theta(kn)$. \footnote{The initial tree is simply the insertion tree $\text{BST}(\pi)$.} We will produce a collection of $\Omega(2^{kn})$ permutations whose optimal costs are all $O(kn)$. By [36], any execution can be uniquely encoded with a binary string of length proportional to the execution’s cost, hence the counting argument implies the desired result.

Let $b = \lfloor 2^k \rfloor$ and $l = \lfloor n/b \rfloor$. Partition the integers 1 to $n$ into at most $l$ contiguous blocks of $b$ items each (we can discard the remainder), and for $1 \leq i \leq l$ let $B_i =$
A permutation in the family is generated as follows. First, successively pick and remove one key from $B_1$, then from $B_2$, and so on up through $B_l$. Then pick one remaining key from each bin in reverse order; i.e. one from $B_l$, then one from $B_{l-1}$, and so on down to $B_1$. Repeat this process in round-robin fashion $l/2$ times until all items are exhausted.

The symmetric-order distance between each pair of keys in any permutation generated as above is at most $2 \cdot b = 2^{k+1}$. By the dynamic finger bound on OPT (due to Splay), the amortized time per access is $O(\lg(2b)) = O(k + 1)$. Hence, $\text{OPT}(\pi) = O(kn)$ for any permutation $\pi$ in the family. The number $N$ of such permutations is as follows. In the first half-round, we can choose any of $b$ elements from each bin, and there are $l$ bins, providing $b^l$ total choices. In the second half-round, we choose from $b-1$ elements in each of $l$ bins, for $(b-1)^l$ choices. Continuing this process gives:

$$N = (bl)^l = b^l(b-1)^l \cdots 2^l \cdot 1$$

$$\implies \lg(N) = \sum_{i=1}^{b} \lg i \geq \frac{l}{2} \sum_{i=b/2}^{b} \lg(b/2)$$

$$\geq \frac{l}{2} \times \frac{b}{2} \times (\lg b - \lg 2)$$

$$\geq \frac{n}{4} \times (k - 1)$$

$$= \Omega(kn)$$

Thus, there is a member of this family whose encoding length (and hence optimal cost) is at least $\Omega(kn)$. Note that this proof is entirely non-constructive. We suspect that it can be made constructive by choosing keys from each block in a manner analogous to Wilber’s “bit-reversal sequence” from [64].
C.3 Wilber’s Version of the Crossing Bound

Computing Crossing Nodes

In Wilber’s original description, the lower bound for request sequence $X = (x_1, \ldots, x_m)$ is defined by the sum of $m$ and the scores of each access number $i$, for $1 \leq i \leq m$.\(^5\) The scoring function, $\kappa(X, i)$, is calculated by looking backward from $i$ in the request sequence $X$ and determining certain critical access numbers and associated keys which correspond to $i$.\(^6\)

A formal program for $\kappa(X, i)$ is provided in [64, Figure 1], along with a detailed example computation.\(^7\) It finds what Wilber calls inside and crossing access numbers, and the corresponding crossing and inside keys for $i$.\(^8\) Assuming $i > 1$, the first crossing access number for $i$ is $c_1 = i - 1$. For all crossing access numbers, the corresponding crossing keys are denoted $w_l = x_{c_l}$. If $w_1 > x_i$, then define $v_0 = -\infty$, and otherwise $v_0 = \infty$.

The next crossing number, $c_{l+1}$, is the largest access number preceding $c_l$ for a key lying between $x_i$ (inclusive) and $v_{l-1}$ (exclusive) in symmetric order. The inside key, $v_l$, is the key closest in symmetric order to $x_i$ (exclusive) on the same side (in symmetric order) of $x_i$ as $w_l$ whose access number, $b_l$, is greater than $c_{l+1}$ and at least $c_l$. If there are multiple requests for $v_l$ between those access numbers, then the inside access number $b_l$ is the greatest among them.

The procedure terminates at the value $l$ such that either $w_l = x_i$ or no further crossing keys can be found. For our purposes, we may assume that a node’s initial access time is the negative of the index of its first appearance in $X$. Hence, the former condition is always met. The output of $\kappa(X, i)$ is the index of the final inside key.

\(^5\)Wilber’s score function is meant to count rotations. He adds 1 to each score to account for the cost of the actual “access” in the lower bound.

\(^6\)The access numbers correspond to the priorities of the crossing nodes at time $i - 1$.

\(^7\)Wilber’s algorithm contains a small mistake (or ambiguity): the loop index, $l$, should be incremented at the end of each execution of the loop body.

\(^8\)Wilber refers to these as nodes instead of keys.
Figure C.1: Configuration of crossing and inside nodes in the proof of Theorem C.3. The node pairs $s_i$ and $t_i$, and $t_{i+1}$ and $x$, may or may not be distinct.

For convenience, we will overload Wilber’s notation, and let $\kappa(X, x) = \kappa(X \oplus (x), |X| + 1)$ whenever $x$ is a node.

**Theorem C.3.** If $T$ is the treap of keys maintained by most recent access time (with initial access times as defined above), then the crossing nodes of Definition 5.1 for $x$ in $T$, ordered from $\text{root}(T)$ to $x$, contain precisely the keys output by $\kappa(X, x)$. The inside keys computed by $\kappa$ are the keys in the parents of the respective crossing nodes in $T$.

**Proof.** Let $v_0, \ldots, v_{l-1}$ and $w_1, \ldots, w_l$ be the respective inside and crossing keys output by $\kappa(X, x)$. (The access numbers $c_1, c_2, \ldots$ decrease monotonically. Hence, $l$ is well-defined.) Let $t_1, \ldots, t_k$ be the crossing nodes for $x$ in $T$ according to Definition 5.1, and let $s_1, \ldots, s_{k-1}$ be the respective parents of $t_2, \ldots, t_k$, with $s_0 \equiv v_0$. (For convenience sake, we may assume that $t_1$ is the right child of $s_0$ if $s_0 = -\infty$, and that it is the left child of $s_0$ if $s_0 = \infty$.) We use induction on $k$.

As $T$ is max-heap ordered by most recent access time, the key of $t_1 = \text{root}(T)$ was requested immediately prior to $x$. Hence, $t_1 = w_1$. This covers the base case. Now suppose that $t_j = w_j$ and $s_{j-1} = v_{j-1}$ for $1 \leq j \leq i$ and $i < k$. Without loss of generality, assume $t_i > x$. Thus, $x$ is in the left subtree of $t_i$. 

93
By construction, \( t_{i+1} \) is the highest descendant of \( t_i \) on the path to \( x \) that is either \( x \), or that has \( x \) in its right subtree. Thus, the sequence of keys on the path from \( t_i \) to \( t_{i+1} \) is a decreasing sequence. The only node on this path with key less than or equal to \( x \) is \( t_{i+1} \). By Definition 5.1, \( t_{i+1} \) is the left child of its parent, \( s_i \). Thus, \( s_i > x \). (See Figure C.1.)

Because \( t_i \) is a crossing node with a left child on the path to \( x \), it is the right child of its parent, \( s_{i-1} \), and \( s_{i-1} \) is an ancestor of every node on the path connecting \( t_i \) to \( t_{i+1} \). Hence, \( \text{lca}_T(s_{i-1}, s_i) = s_{i-1} \), and every node in the symmetric order interval \((s_{i-1}, s_i)\) is in the right subtree of \( s_{i-1} \). The root of this subtree, \( t_i \), is greater than \( s_i \). Thus, keys in the symmetric order interval \((s_{i-1}, s_i)\) lie in the left subtree of \( t_i \).

Similarly, these keys lie in the left subtree of every node on the path beginning at \( t_i \) and ending at \( s_i \). As stated before, the root of \( s_i \)'s left subtree is \( t_{i+1} \).

By the above argument, \( t_{i+1} \) is an ancestor of every node in the interval \((s_{i-1}, s_i)\). By the max-heap order on \( T \), \( t_{i+1} \) was accessed most recently among them. By the symmetric order, \( s_{i-1} < t_{i+1} \leq x \). As \((v_{i-1}, x] \subseteq (v_{i-1}, s_i)\), \( t_{i+1} \) has the key that was most recently requested in \( X \) prior to \( w_i \) that lies between \( s_{i-1} \) and \( x \) in symmetric order. By the inductive hypothesis, \( s_{i-1} = v_{i-1} \), and therefore \( t_{i+1} = w_{i+1} \).

As seen earlier, \( x < s_i \leq t_i \). Thus, \( s_i \) lies on the same side of \( x \) as \( t_i \). The symmetric-order interval \((x, s_i)\) of keys greater than \( x \) that lie closer to \( x \) than \( s_i \) is itself contained in the interval \((s_{i-1}, s_i)\). All nodes in the latter interval are ancestors of \( t_{i+1} \). By the max-heap order, they were accessed no more recently than \( t_{i+1} \). Furthermore, the max-heap order implies \( s_i \) was accessed more recently than \( t_{i+1} \), because \( s_i \) is the parent of \( t_{i+1} \). Similarly, \( s_i \) was accessed no earlier than \( t_i \), because \( s_i \) is an ancestor of \( t_i \). We conclude that \( s_i \) has the smallest key greater than \( x \) that was accessed after \( t_{i+1} \) but no later than \( t_i \). From above, \( t_{i+1} = w_{i+1} \), and by the inductive hypothesis, \( t_i = w_i \). Thus, \( s_i = v_i \).
To summarize, the first crossing node output by Definition 5.1 contains the first crossing key output by $\kappa(X, x)$, and every subsequent crossing node from Definition 5.1 contains the corresponding key $\kappa(X, x)$. The last crossing node of Definition 5.1 is $t_k = x$, and $\kappa(X, x)$ also terminates after it finds $x$. In conclusion: $k = l$ and Definition 5.1 is equivalent to Wilber’s.

Remark C.1. A great way to test one’s code is to implement both methods of computing crossing nodes and compare them for equality.

The Bound

Definition 5.1 of the crossing bound differs from Wilber’s in two respects. First, Wilber does not account for initial trees. (Or rather, Wilber’s initial tree is an implicit function of the request sequence.) Second, Wilber does not count a requested node among the crossing nodes until after the first access to that node’s key. The exact relationship is

$$\Lambda_2(X) = \Lambda(X, \text{BST}(X)) - n + \delta,$$

where $\Lambda_2(X)$ denotes Wilber’s version of the crossing bound on $X$ [64], $\text{BST}(X)$ is the insertion tree for $X$, $n$ is the number of unique keys in $X$, and $\delta$ is 0 if $X$ is empty and 1 otherwise.

Let $\text{OPT}_{st}^{mr}$ denote the optimum cost to execute $(X, T)$ in Sleator and Tarjan’s original model (See Appendix C.1), subject to the restriction that executions must rotate requested nodes to the root prior to the search.

Theorem C.4 (Theorem 7 from [64]). $\Lambda_2(X) \leq \text{OPT}_{st}^{mr}(X, T)$.

The proof of this theorem is rather technical, and we do not attempt to summarize it here. Instead, we use it as a black box in the following analysis.

Proof of Theorem 5.1. Assume $X = (x_1, \ldots, x_m)$. Let $T_0 = T$ and $Z_0 = \text{postorder}(T)$. For $1 \leq i \leq m$, let $T_i = \text{move-to-root}(T_{i-1}, x_i)$ and $Z_i = Z_{i-1} \oplus (x_i)$.
By Remark 5.1 and Theorem C.3, \( \sum_{i=1}^{m} \ell_{T_{i-1}}(x_i) = \sum_{i=1}^{m} \kappa(Z_{i-1}, x_i) \). Thus,

\[
\Lambda(X, T) = \Lambda_2(\text{postorder}(T) \oplus X) - \Lambda_2(\text{postorder}(T)) \\
\leq \Lambda_2(\text{postorder}(T) \oplus X) \\
\leq \operatorname{OPT}_{st}^m(\text{postorder}(T) \oplus X, T) \quad \text{[Theorem C.4]} \\
\leq 3 \operatorname{OPT}(\text{postorder}(T) \oplus X, T) \quad \text{[Theorem C.2]} \\
\leq 3(\operatorname{OPT}(\text{postorder}(T), T) + \operatorname{OPT}(X, T)) \quad \text{[Theorem 3.8]} \\
\leq 3(|T| + \operatorname{OPT}(\text{postorder}(T), \text{BST}(\text{postorder}(T)))) + \operatorname{OPT}(X, T) \quad \text{[Remark 2.3]} \\
\leq 3(|T| + \text{cost}(\text{postorder}(T), \text{BST}(\text{postorder}(T)))) + \operatorname{OPT}(X, T) \\
\leq 3(7|T| + \operatorname{OPT}(X, T)) \quad \text{[Theorem B.2]} \\
\leq 24 \operatorname{OPT}(X, T). \quad \text{[Remark 2.2]}
\]

\[\Box\]
Appendix D

Proof of Theorem 5.2

As a reminder, we wish to show that $\Lambda(Z, S) - \Lambda(Z, \text{move-to-root}(S, x)) \leq 4\ell_S(x)$. The function $\Lambda$ is computed algorithmically: we execute Move-to-Root and count the crossing nodes encountered while doing so. Thus, we must compare Move-to-Root’s execution of the same request sequence starting from two different, but related, configurations. We want to bound the difference between the crossing cost of these executions by a constant multiple of the crossing cost required to transform the first initial configuration into the second.

Executing the request sequence $Z$ from two different starting trees can cause the after-trees to differ dramatically. However, Move-to-Root’s recursive structure ensures that the manner in which the after-trees $S_i$ and $T_i$ differ can be precisely characterized by using a small number of parameters.

D.1 Window Subtrees

The most important parameters are the left and right “window boundaries,” $u_i$ and $v_i$. Let $u_0 = -\infty$ and $v_0 = \infty$. (We assume $u_0$ and $v_0$ are not in $Z$.) For $1 \leq i \leq m$, let $u_i = z_i$ if $u_{i-1} \leq z_i \leq x$; otherwise, let $u_i = u_{i-1}$. Similarly, let $v_i = z_i$ if $x \leq z_i \leq v_{i-1}$; otherwise, let $v_i = v_{i-1}$. Note that $u_i$ is the largest among the first
Figure D.1: Depiction of the after-tree $S_i$ with zipped subtree $J_i$ and after-tree $T_i$, and unzipped subtree $K_i$. The top tree $I_i$ is identically arranged in both $S_i$ and $T_i$ by Lemma D.1. By Lemma D.3, we need merely compute crossing depths for nodes on the access path for $x$ and their immediate children. Horizontally striped nodes are those for which $A_i = 1$. (See Section D.3.) Vertically striped nodes are those for which $B_i = 1$. Crosshatched nodes are those for which $A_i B_i = 1$. For this tree, $E_i = 1$ when root($J_i^+$) = $v_i$. The nodes $w_i^1, w_i^2, \ldots$ are the crossing nodes for $x$ in $J_i$.

requested keys that is less than or equal to $x$ (assuming $u_i \neq -\infty$), and $v_i$ is the smallest among the first $i$ requested keys that is greater than or equal to $x$ (assuming $v_i \neq \infty$). The symmetric order interval lying between these keys narrows from one request to the next.

We will also need the most recent access times, $s_i$ and $t_i$, for the left and right window boundaries. Define $s_0 = t_0 = -\infty$. If $u_i = z_i$ then $s_i = i$; otherwise, $s_i = s_{i-1}$. Similarly, if $v_i = z_i$ then $t_i = i$; otherwise, $t_i = t_{i-1}$.

Let $I_i$ denote the set of keys in $S$ that lie outside the symmetric order interval $(u_i, v_i)$. As the next lemma shows, the nodes whose key lies outside $I_i$ are identically

\footnote{This unusual definition helps to reduce case analysis.}
arranged as a connected subtree of the root in both $S_i$ and $T_i$. As such, we will also let $I_i$ denote this subtree, and refer to it as the “top tree.” The remaining keys in $S_i$ reside in the subtree $J_i$, and the remaining keys of $T_i$ reside in the subtree $K_i$, where, for $0 \leq i \leq m$,

$$J_i = \begin{cases} S_0 & i = 0 \\ \text{right-subtree}_{S_i}(u_i) & s_i < t_i \\ \text{left-subtree}_{S_i}(v_i) & s_i > t_i \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$K_i = \begin{cases} S_0 & i = 0 \\ \text{right-subtree}_{T_i}(u_i) & s_i < t_i \\ \text{left-subtree}_{T_i}(v_i) & s_i > t_i \\ \emptyset & \text{otherwise} \end{cases}$$

We will call $J_i$ the “zipped subtree” and $K_i$ the “unzipped subtree” at time $i$. Importantly, $K_i = \text{move-to-root}(J_i, x)$. (See Figure D.1.) Putting these observations together:

**Lemma D.1 (Subtree Structure).** For $0 \leq i \leq m$:

(a) If $i \geq 1$ then $\text{root}(S_i) \in I_i$ and $\text{root}(T_i) \in I_i$, and $\text{root}(S_i) = \text{root}(T_i)$, and

$$\text{parent}_{S_i}(y) = \text{parent}_{T_i}(y) \quad \text{for} \quad y \in I_i \setminus \{\text{root}(S_i)\}.$$  

(b) The following are equivalent:

- $y \in J_i$
- $y \in K_i$
- $u_i < y < v_i$.

(c) $K_i = \text{move-to-root}(J_i, x)$ whenever $u_i \neq v_i$.

The proof of this lemma is tedious and not of immediate interest. Thus, it is relegated to Section D.5.

**Remark D.1.** In fact, the subtree structure of $J_i$ and $K_i$ are independent of everything except for $S_0$, $u_i$ and $v_i$. This is because the priority of a node whose key lies
in the symmetric order interval \((u_i, v_i)\) is the same in \(S_i\) and \(T_i\) as it is in \(S_0\) and \(T_0\), respectively.

**Remark D.2.** We may assume that the priorities of nodes in \(S_0\) are as in Remark 5.1. The priority of a node in \(T_0\) is the same as the node of \(S_0\) with the same key, except that the node in \(T_0\) containing the key \(x\) has priority 0. The priority of \(z_i\) is set to \(i\) in both \(S_i\) and \(T_i\).

If \(z_i = x\) then \(u_j = v_j = x\) and \(s_j = t_j\) and \(S_j = T_j\) and \(\Delta_j(z_{j+1}) = 0\) for \(i \leq j \leq m\). This is because accessing \(x\) sets its priority to be the same in both \(S_i\) and \(T_i\). Thus, all nodes share the same priority in \(S_i\) and \(T_i\), and the trees remain identical for every request thereafter. Since requests after \(x\) contribute no crossing depth difference, we may assume that if \(z_i = x\) then \(i = m\) is the last request. Thus, for \(1 \leq i \leq m - 1\), we may assume without loss of generality that \(z_i \neq x\) and that \(J_i\) and \(K_i\) are nonempty.

### D.2 Simplifying Level Difference Functions

By Lemma D.1, the access path for every key in the top tree is the same in both \(S_i\) and \(T_i\). Thus, we concern ourselves with the subtrees \(J_i\) and \(K_i\). Understanding the structure of \(J_i\) and \(K_i\) is almost, but not quite, sufficient for characterizing the “level difference function,” \(\Delta_i\). The only additional required detail is the directionality of \(J_i\) and \(K_i\) with respect to the parent of these subtrees in \(S_i\) and \(T_i\). Thus, we define the “augmented” subtrees, \(J_i^+\) and \(K_i^+\).
If $s_i = t_i$ then $J_i^+ = J_i$ and $K_i^+ = K_i$. Otherwise, if $s_i < t_i$, then

$$u_i = \text{root}(J_i^+) = \text{root}(K_i^+)$$
$$\text{null} = \text{left-child}_{J_i^+}(u_i) = \text{left-child}_{K_i^+}(u_i)$$
$$J_i = \text{right-subtree}_{J_i^+}(u_i)$$
$$K_i = \text{right-subtree}_{K_i^+}(u_i),$$

and if $s_i > t_i$, then

$$v_i = \text{root}(J_i^+) = \text{root}(K_i^+)$$
$$\text{null} = \text{right-child}_{J_i^+}(v_i) = \text{right-child}_{K_i^+}(v_i)$$
$$J_i = \text{left-subtree}_{J_i^+}(v_i)$$
$$K_i = \text{left-subtree}_{K_i^+}(v_i).$$

Without loss of generality, we may assume that $z_i \in J_i^{+}$. More precisely:

**Lemma D.2 (Augmented Funnel).** For $0 \leq i \leq m - 1$:

(a) $\Delta_i(y) = \begin{cases} 
\ell_{J_i^+}(y) - \ell_{K_i^+}(y) & y \in J_i^+ \\
0 & \text{otherwise}
\end{cases}$

(b) If $z_i+1 \notin J_i^+$ then $J_{i+1}^+ = J_i^+$ and $K_{i+1}^+ = K_i^+$.

**Remark D.3.** An example where the un-augmented subtrees provide insufficient information to compute crossing depth differences is $S = \text{BST}(1, 2, 3)$, $x = 3$, $z_1 = 1$ and $z_2 = 2$. Note that, $\ell_{J_1}(z_2) = \ell_{J_1}(z_2) = 2$ while $\Delta_1(z_2) = -1 \neq \ell_{J_1}(z_2) - \ell_{K_1}(z_2)$.

In fact, even $J_i^+$ and $K_i^+$ have more information than needed. Most of what we require is determined by the structure of the access paths for $x$, and $x$’s symmetric order neighbors, in $J_i$. Let $P_i$ denote the tree formed by first removing the off-path
subtrees hanging from the access path for \( x \) in \( J_i \) and then setting \( \text{left-subtree}_{P_i}(x) = \text{right-spine}(\text{left-subtree}_{J_i}(x)) \) and \( \text{right-subtree}_{P_i}(x) = \text{left-spin}(\text{right-subtree}_{J_i}(x)). \) (If \( J_i = \emptyset \) then \( P_i = \emptyset. \)) We call \( P_i \) the “access path” for \( x. \) The “augmented” access path \( P_i^+ \) is defined similarly, except we start with the access path for \( x \) in \( J_{i+1}^+ \) instead of in \( J_i. \)

The level differences are identical for all keys in a given subtree hanging from the access path. More precisely:

**Lemma D.3** (Off-Path Subtrees). For \( 0 \leq i \leq m \) and \( y \in P_i \setminus \{x\}, \) if

\[
Q_i(y) = \begin{cases} 
\text{left-subtree}_{J_i}(y) & y < x \\
\text{right-subtree}_{J_i}(y) & y > x
\end{cases}
\]

and \( \hat{Q}_i(y) = \begin{cases} 
\text{left-subtree}_{K_i}(y) & y < x \\
\text{right-subtree}_{K_i}(y) & y > x
\end{cases}\)

then

(a) \( Q_i(y) = \hat{Q}_i(y). \)

(b) \( \Delta_i(z) = \Delta_i(z') \) for \( z, z' \in Q_i(r). \)

Lemmas D.2 and D.3 are both proved in Appendix D.5.

A consequence of the previous two lemmas is that we may assume without loss of generality that either \( z_i \in P_i^+ \) or \( \text{parent}_{S_{i-1}}(z_i) \in P_i^+ \setminus \{x\}. \) More precisely:

**Lemma D.4** (Level Differences). For \( 1 \leq i \leq m, \) if \( \bar{z}_i \) denotes the deepest ancestor of \( z_i \) in \( J_{i-1}^+ \) such that either \( \bar{z}_i \in P_{i-1}^+ \) or \( \text{parent}_{S_{i-1}}(\bar{z}_i) \) is defined and lies in \( P_{i-1}^+ \) then \( \Delta_{i-1}(z_i) = \ell_{J_{i-1}^+}(\bar{z}_i) - \ell_{K_{i-1}^+}(\bar{z}_i). \)

**Proof.** If \( z_i \in P_{i-1}^+ \) then \( \bar{z}_i = z_i. \) Otherwise, \( \bar{z}_i \) is the root of the path subtree containing \( z_i. \) In either case, \( \ell_{J_{i-1}^+}(z_i) - \ell_{K_{i-1}^+}(z_i) = \ell_{J_{i-1}^+}(\bar{z}_i) - \ell_{K_{i-1}^+}(\bar{z}_i) \) by Lemma D.3. By Lemma D.2, \( \Delta_{i-1}(z_i) = \ell_{J_{i-1}^+}(z_i) - \ell_{K_{i-1}^+}(z_i). \) □

\(^2\)While \( P_i \) is not necessarily a path, it has at most one node, \( x, \) with two children, so it is “path-like.”
D.3 Formulae for Level Differences

As we shall see, the difference between $z_i$’s level in $S_{i-1}$ and $T_{i-1}$ is, to within constant-order additive terms, the number of $z_i$’s ancestors in $S_i$ that are crossing nodes for $x$ in $P_i$.

More precisely, let $k_i$ denote $x$’s crossing depth in $J_i$. (If $J_i = \emptyset$ then define $k_i = 0$.) Let $w_i^1, \ldots, w_i^{k_i-1}$ be the first $k_i - 1$ crossing nodes for $x$ in $J_i$, ordered descending. Define $w_i^{-1} = x$, and let

$$w_i^0 = \begin{cases} 
\text{null} & i = 0 \\
\text{root}(J_i^+) & \text{otherwise}
\end{cases}$$

$$w_i^{k_i} = \begin{cases} 
\text{left-child}_{J_i}(x) & x = \text{left-child}_{J_i}(\text{parent}_{J_i}(x)) \\
\text{right-child}_{J_i}(x) & x = \text{right-child}_{J_i}(\text{parent}_{J_i}(x))
\end{cases}$$

$$w_i^{k_i+1} = \begin{cases} 
\text{right-child}_{J_i}(x) & x = \text{left-child}_{J_i}(\text{parent}_{J_i}(x)) \\
\text{left-child}_{J_i}(x) & x = \text{right-child}_{J_i}(\text{parent}_{J_i}(x))
\end{cases}$$

Defining the “extended” crossing nodes $w_i^0, w_i^{k_i}$ and $w_i^{k_i+1}$ in this manner allows certain details to be swept into index arithmetic and relieves us from having to provide duplicitous descriptions of symmetric situations.

**Remark D.4.** If $k_i = 1$ then $x$ is the node of greatest priority in $J_i$. By Remark D.2, $x$ is also the node of greatest priority in $K_i$. All other keys and priorities in $J_i$ and $K_i$ are the same, so $J_i = K_i$. By Lemma D.1, this means $S_i = T_i$ and $\Delta_j(z_{j+1}) = 0$ for $i \leq j < m$. Therefore, we may assume $k_i > 0$ for $i < m$, in which case $\text{parent}_{J_i}(x)$ is well-defined.

Let $c_i$ be the index of the deepest ancestor of $z_i$ in $J_i$ that is a crossing node for $x$ in $J_i$. That is, $c_i = \max\{j \mid w_i^j \in \text{ancestors}_{J_i^+}(z_i)\}$. (If $z_i = x$ then $c_i = -1$.)
\[ z_i \neq w_i^{0-1} \text{ then define } l_i = \ell_{J_{i-1}}(w_i^{c_i-1}); \text{ otherwise, define } l_i = 0. \] (We refer to \( l_i \) as \( z_i \)'s “crossing zone.”) Note that \( l_i = c_i \) except when \( z_i = x \).

As the next two lemmas demonstrate, \( \ell_{J_{i-1}^+}(z_i) = l_i + O(1) \) while \( \ell_{K_{i-1}^+}(z_i) = O(1) \).

The terms inside the \( O(1) \) depend on the following indicator functions:

\[ \delta_i = \begin{cases} 
1 & \text{ if } i > 1 \\
0 & \text{ otherwise} 
\end{cases} \quad \text{ First Access Indicator} \]

\[ A_i = \begin{cases} 
1 & \text{ if } z_i \notin P_{i-1}^+ \\
0 & \text{ otherwise} 
\end{cases} \quad \text{ Subtree Indicator} \]

\[ B_i = \begin{cases} 
1 & \text{ if } z_i \neq w_i^{c_i} \\
0 & \text{ otherwise} 
\end{cases} \quad \text{ Non-Crossing Indicator} \]

\[ E_i = \begin{cases} 
1 & \text{ if } \ell_{J_{i-1}}(x) < \ell_{J_{i-1}^+}(x) \\
0 & \text{ otherwise} 
\end{cases} \quad \text{ Extra Level Indicator} \]

\[ F_i = \begin{cases} 
1 & \text{ if } \ell_{K_{i-1}}(z_i) < \ell_{K_{i-1}^+}(z_i) \\
0 & \text{ otherwise} 
\end{cases} \quad \text{ Extra Level Indicator} \]

\textbf{Lemma D.5 (Zipped Subtree Levels).} For \( 1 \leq i \leq m \):

\[ \ell_{J_{i-1}^+}(z_i) = l_i + \begin{cases} 
E_i & c_i = -1 \\
1 & c_i = 0 \\
(1 - A_i)(1 - B_i)\delta_i + B_i(1 + A_i + E_i) + A_i(1 - B_i)(1 + \delta_i(1 - E_i)) + B_i(1 + A_i) + E_i & c_i = 1 \\
2 \leq c_i \leq k_{i-1} + 1 
\end{cases} \]
Proof. By Lemma D.4, we may assume that $z_i$ is either on path $P_{i-1}^+$ or is a child of a node on $P_{i-1}^+$.

In $J_{i-1}$, the crossing depths of the $x$’s crossing nodes is simply their crossing zone. Beyond the first crossing zone, the off-path children of the crossing nodes for $x$ in $J_{i-1}$ are children of the same direction as their parents. Thus, the crossing depths of these children are the same as their parents. Since $\ell_{J_{i-1}}(x) > 1$ by Remark D.4, $\ell_{J_{i-1}^+}(x) = \ell_{J_{i-1}^+}(w_{i-1})$ by definition.

Non-crossing path nodes beyond the first crossing zone are children of opposite direction from the crossing nodes for their respective zones. Thus, the crossing depths of these nodes are greater by one than the respective crossing zones that they reside in. Off-path children of non-crossing nodes always have crossing depth one-greater than their parents.

From Figure D.1, it is apparent that $\ell_{J_{i-1}^+}(z_i) = \ell_{J_{i-1}}(z_i) + E_i$ whenever $c_i > 1$. The expression for when $c_i = 1$ is derived via case analysis.

Lemma D.6 (Unzipped Subtree Levels). For $1 \leq i \leq m$:

$$\ell_{K_{i-1}^+}(z_i) = \begin{cases} 1 + \delta_i & c_i = -1 \\ 1 & c_i = 0 \\ 2 + F_i + B_i(1 + A_i) & 1 \leq c_i \leq 2 \\ 3 + F_i + A_i & 3 \leq c_i \leq k_{i-1} + 1 \end{cases}$$

Proof. By Lemma D.1(c), $K_{i-1} = \text{move-to-root}(J_{i-1}, x)$, thus $x = \text{root}(K_{i-1})$, and $x$ has another crossing node in $K_{i-1}^+$ whenever $\text{root}(K_{i-1}) \neq \text{root}(K_{i-1}^+)$, which is true after the first access.

For the remaining nodes, one can compare Move-to-Root to the unzipping operation in Zip Trees [62]. Accessing $z$ with Move-to-Root splits the path to $z$ into a path $L$ containing all nodes with keys less than $z$’s and a path $R$ containing all nodes with
keys greater than z’s. Along L from top to bottom, nodes are in increasing order by key; along R from top to bottom, nodes are in decreasing order by key. This process preserves the left subtrees of the nodes on L and the right subtrees of the nodes on R. We make z’s left subtree the rightmost on L and its right subtree the leftmost on R. Finally, z’s left subtree becomes L and its right subtree becomes R.

Thus, the nodes less than x in $P_{i-1}$ comprise the right spine of left-subtree $K_{i-1}(x)$ and the nodes that are less than x comprise the left spine of right-subtree $K_{i-1}(x)$. The roots these subtrees, $w_{i-1}^1$ and $w_{i-2}^2$, have crossing depth 2 in $K_{i-1}$, as do the off-path children of these roots. The remaining path nodes have crossing depth three in $K_{i-1}$, and the off-path children of these nodes have crossing depth four in $K_{i-1}$. All descendants of x lying between root($K_{i-1}$) and root($K_{i-1}^{+}$) have one more crossing node in $K_{i-1}^{+}$ than in $K_{i-1}$.

\section{D.4 Bounding the Sum}

The value $k_i$ is, to within a constant-order additive term, the maximum possible difference, taken over all keys, between a key’s level in $S_i$ and in $T_i$. The bound of Theorem 5.2 comes from showing that $k_i = k_{i-1} - \Delta_{i-1}(z_i) \pm O(1)$. More precisely:

\textbf{Lemma D.7} (Bound on Level Differences). For $1 \leq i \leq m$,

$$\Delta_{i-1}(z_i) \leq \begin{cases} 
0 & 1 \leq l_i \leq 2 \\
 l_i & \text{otherwise}
\end{cases}$$

\textbf{Proof.} The only important things to note are that if $c_i = 1$ then $F_i = E_i$; if $c_i = 2$ then $F_i = \delta_i(1 - E_i)$; and if $E_i = 1$ or $F_i = 1$ then $\delta_i = 1$. The rest is case analysis from Lemmas D.5 and D.6. \qed
Lemma D.8 (Max Level Difference Decreases). For $1 \leq i \leq m$,

$$k_i \leq k_{i-1} - \begin{cases} 
  k_{i-1} & c_i = -1 \\
  0 & 0 \leq c_i \leq 2 \\
  l_i - 2 & 3 \leq c_i \leq k_i \\
  l_i - 3 & 2 < k_i < c_i 
\end{cases}$$

Proof. Let $P_i^- = \text{access-path}_{J_{i-1}}(x)$. We first deal with the special cases. If $c_i = 1$ then $P_i^-$ is simply a strict sub-path of $P_{i-1}^-$, in which case $k_i \leq k_{i-1}$. If $c_i = 0$ then $P_i^- = P_{i-1}^-$ and $k_i = k_{i-1}$. If $c_i = -1$ then $k_i = 0$. Accessing a key in left-subtree $J_{i-1}(x)$ makes $P_i^-$ a rightward path to $x$, and accessing a key in right-subtree $J_{i-1}(x)$ makes $P_i^-$ a leftward path to $x$. In either case, $\ell_{P_i^-}(x) \leq 2$. Also, note that if $k_{i-1} = 2$ then $w_{i-1}^{k_{i-1}+1}$ is on the same side of $x$ as parent $J_{i-1}(x)$ by construction. Thus, if $k_{i-1} = 2$ then $c_i = 3$ means that all remaining nodes are knocked off the access path, and $k_i = 1$.

Otherwise, if $1 \leq c_i \leq k_{i-1} - 1$, let

$$q_i = \begin{cases} 
  \text{parent}_{J_{i-1}}(z_i) & z_i \notin P_{i-1}^- \\
  \text{right-child}_{J_{i-1}}(z_i) & z_i < x \\
  \text{left-child}_{J_{i-1}}(z_i) & z_i > x.
\end{cases}$$

Let $L_i$ be a rightward path comprising the strict ancestors of $q_i$ in $P_{i-1}^-$ that are less than $x$, and let $R_i$ be leftward path of the strict ancestors of $q_i$ in $P_{i-1}^-$ that are greater than $x$. Let $M_i$ denote the path from $q_i$ to $x$ in $P_{i-1}^-$. Let $M_i$ denote the path from $q_i$ to $x$ in $P_{i-1}^-$. 

If $z_i < x$ then $P_i^-$ is formed by attaching $M_i$ as the left subtree of the leftmost node in $R_i$; if $z_i > x$ then $P_i^-$ is formed by attaching $M_i$ as the right subtree of the rightmost
node in $L_i$. There were at least $l_i$ crossing nodes for $x$ in $P_{i-1}^-$ on access-path$_{P_{i-1}^-}(q_i)$.

Meanwhile, access-path$_{P_i^-}(q_i)$ comprises pointers of a single direction. (That is, every pointer is left or every pointer is right.) Thus access-path$_{P_i^-}(q_i)$ contains at most two crossing nodes for $x$ in $P_i^-$, in which case $k_i \leq k_{i-1} - (l_i - 2)$.

**Proof of Theorem 5.2.** By Lemma D.8, $l_i \leq k_{i-1} - k_i + 3$. Let

$$G_i = \begin{cases} 1 & k_i < k_{i-1} \\ 0 & \text{otherwise.} \end{cases}$$

By Lemma D.7, $\Delta_{i-1}(z_i) \leq k_{i-1} - k_i + 3G_i$. Since $k_0 = \ell_S(x)$, $k_m \geq 0$, and $G_i$ is only 1 when $x$'s $k_i < k_{i-1}$ (which can happen at most $\ell_S(x)$ times), we have $\sum_{i=1}^{m} \Delta_{i-1}(z_i) \leq 4\ell_S(x)$.

**D.5 Proofs of Lemmas**

**Miscellaneous Observations**

We will use the following observations. They are stated without proof, and some may be considered standard textbook exercises, such as those in [15].

**Definition D.1.** For a set of keys $V$ and $y \in V$, if $y > \min V$ then define $\text{predecessor}_V(y) = \max\{y' \in V \mid y' < y\}$, and if $y < \max V$ then define $\text{successor}_V(y) = \min\{y' \in V \mid y' > y\}$. (Both $\text{successor}_V(\max V)$ and $\text{predecessor}_V(\min V)$ are undefined.)
Lemma D.9. For a binary search tree $V$ and $y \in V$

\[
\begin{align*}
\text{predecessor}_V(y) &= \begin{cases} 
\max \text{ left-subtree}_V(y) & \text{left-child}_V(y) \neq \text{null} \\
\max \{y' \mid \text{right-child}_V(y') \in \text{ancestors}_V(y)\} & \text{otherwise}
\end{cases} \\
\text{successor}_V(y) &= \begin{cases} 
\min \text{ right-subtree}_V(y) & \text{right-child}_V(y) \neq \text{null} \\
\min \{y' \mid \text{left-child}_V(y') \in \text{ancestors}_V(y)\} & \text{otherwise}
\end{cases}
\end{align*}
\]

whenever each of these quantities is defined.

Lemma D.10. Let $y$ be a child in $V$. If $y$ is a right child then $\text{left-subtree}_V(y) = \text{left-subtree}_V(y)$. If $y$ is a right child then $\text{right-subtree}_V(y) = \text{right-subtree}_V(y)$.

Lemma D.11. Let $y$ and $z$ be distinct nodes in $V$, and let $V' = \text{move-to-root}(V, z)$. If $\text{left-child}_V(y) \notin \text{access-path}_V(z)$ then $\text{left-subtree}_V(y) = \text{left-subtree}_V(y)$; if $\text{right-child}_V(y) \notin \text{access-path}_V(z)$ then $\text{right-subtree}_V(y) = \text{right-subtree}_V(y)$.

Lemma D.12. Let $V$ be a binary tree, and let $y, z \in V$. If $y \in \text{ancestors}_V(z)$ and $V'$ is formed by replacing $V_y$ with $\text{move-to-root}(V_y, z)$, where $V_y$ is the subtree rooted at $y$ in $V$, then $\text{move-to-root}(V, z) = \text{move-to-root}(V', z)$.

Remark D.5. This “overlapping sub-structure” property of Move-to-Root is perhaps the distinguishing feature which makes it easier to analyze than Splay. In Splay, the zig-zig operations destroy this property.

Lemma D.13. Let $W$ be a connected subtree of the root of $V$, let $V$ be a connected subtree of the root of binary search tree $U$, and let $W'$ be a binary search tree with the same keys as $W$. If $V'$ is the result of substituting $W'$ for $W$ in $V$, $U'$ is the result of substituting $W'$ for $W$ in $U$, and $U''$ is the result of substituting $V'$ for $V$ in $U$, then $U' = U''$.

Lemma D.14. If $y \neq z$ and $y \in \text{crossing-nodes}_V(z)$ then $y \in \text{crossing-nodes}_V(z')$ for $z' \in \text{descendants}_V(z)$.
**Definition D.2.** Let $V$ be a binary search tree and $y \notin V$. If $y < \min V$ or $y > \max V$ then define the binary search tree $V' = \text{augment-top}(V, y)$, as follows. $\text{root}(V') = y$. If $y < \min V$ then $\text{left-child}_{V'}(y) = \text{null}$ and $\text{right-subtree}_{V'}(y) = V$. Otherwise, if $y > \max V$ then $\text{right-child}_{V'}(y) = \text{null}$ and $\text{left-subtree}_{V'}(y) = V$.

**Proofs for Section D.1**

**Proof of Lemma D.1.** We induct on the number of requests in $Z$. Note that $I_0 = \emptyset$. Thus, (a) is vacuously true for $m = 0$. Both $J_0$ and $K_0$ contain every node in $S$, so (b) is true when $m = 0$. By construction $J_0 = S_0$, $K_0 = T_0$, and $T_0 = \text{move-to-root}(S_0, x)$, so (c) is true when $m = 0$. Now suppose the lemma is true for all request sequences of length at most $m - 1$.

Item (c) is a consequence of (b), as follows. $J_m$ and $K_m$ have the same keys. None of these keys have been accessed so far in $Z$. The only key whose priority in $K_m$ differs from its priority in $J_m$ is $x$, which has been set to be greater than the priority of every other key in $K_m$. This is exactly the effect of executing $\text{move-to-root}(J_m, x)$. (See Remark D.2.) We now deal with (a) and (b).

**Case I: $z_m \in I_{m-1}$.** In this case, $u_m = u_{m-1}$, $v_m = v_{m-1}$, and $I_m = I_{m-1}$. Without loss of generality, assume $s_{m-1} < t_{m-1}$. (The other case is symmetric.) Thus, $J_{m-1} = \text{right-subtree}_{S_{m-1}}(u_m)$ and $K_{m-1} = \text{right-subtree}_{T_{m-1}}(u_m)$.

By the inductive hypothesis for (a) on $m - 1$, the nodes of $I_{m-1}$ form an identical connected subtree of the root of $S_{m-1}$ and $T_{m-1}$. Let $\hat{I}_{m-1}$ denote this subtree. Note that $\text{access-path}_{I_{m-1}}(z_m) = \text{access-path}_{S_{m-1}}(z_m) = \text{access-path}_{T_{m-1}}(z_m)$. Move-to-Root is a “path-based” algorithm. (See Chapter 7.4.) Thus, Move-to-Root outputs identical transition trees for $z_m$ in $\hat{I}_{m-1}$, $S_{m-1}$ and $T_{m-1}$. By Lemma D.13, $S_m$ and $T_m$ are formed by substituting $\hat{I}_m = \text{move-to-root}(\hat{I}_{m-1}, z_m)$ for $\hat{I}_{m-1}$ in $S_{m-1}$ and $T_{m-1}$. 

---

3This is essentially how Tarjan implemented push and inject operations in [61]. See Chapter 7.2.
Suppose $z_m \neq u_m$. By Lemma D.11, right-subtree$_{S_m}(u_m) = \text{right-subtree}_{S_{m-1}}(u_m)$ and right-subtree$_{T_m}(u_m) = \text{right-subtree}_{T_{m-1}}(u_m)$. Because Move-to-Root is path-based, the subtrees $J_{m-1}$ and $K_{m-1}$ remain unaltered in $S_m$ and $T_m$, respectively. Furthermore, $s_m < t_m$ since $s_m = s_{m-1}$ and the priorities of other nodes can only increase. Thus, $J_m = \text{right-subtree}_{S_m}(u_m) = J_{m-1}$ and $K_m = \text{right-subtree}_{T_m}(u_m) = K_{m-1}$. By the inductive hypothesis on (b) for $m - 1$, $y \in J_{m-1} \iff u_{m-1} < y < v_{m-1} \iff y \in K_{m-1}$. Since $u_m = u_{m-1}$ and $v_m = v_{m-1}$, $J_m = J_{m-1}$ and $K_m = K_{m-1}$, (b) holds for $m$.

Suppose $z_m = u_m$ and $t_{m-1} = \infty$. Since $\infty$ is never accessed, $t_m = t_{m-1}$. Node $u_m$ is the maximum element of $\hat{I}_{m-1}$, and therefore all of $u_m$’s non-root ancestors are right children. By Lemma D.10, the rotations during the execution of move-to-root($I_{m-1}, u_m$) do not change $u_m$’s right subtree. Thus, right-subtree$_{S_m}(u_m) = J_{m-1}$ and right-subtree$_{T_m}(u_m) = K_{m-1}$. Since $t_m = \infty$, $J_m = \text{right-subtree}_{S_m}(u_m)$ and $K_m = \text{right-subtree}_{T_m}(u_m)$. Thus, $J_m = J_{m-1}$ and $K_m = K_{m-1}$ and the lemma holds.

Suppose $z_m = u_m$ and $t_{m-1} < \infty$. In this case, $v_m = \text{successor}_{\hat{I}_{m-1}}(u_m)$. Since right-child$_{\hat{I}_{m-1}}(u_m) = \text{null}$, by Lemma D.9, $v_m$ is the lowest ancestor of $u_m$ that is not a left child. Starting from $S_{m-1}'$ and $T_{m-1}'$, form $S_{m-1}''$ and $T_{m-1}''$ by repeatedly rotating at $u_m$ until it becomes the left child of $v_m$, and let $S_{m-1}'' = \text{rotate}(S_{m-1}', z_m)$ and $T_{m-1}'' = \text{rotate}(T_{m-1}', z_m)$.

Because $u_m$ was a right child during all of these rotations, by Lemma D.10, right-subtree$_{S_{m-1}''}(u_m) = J_{m-1}$ and right-subtree$_{T_{m-1}''}(u_m) = K_{m-1}$. Because $u_m$ is a left child in $S_{m-1}'$ and $T_{m-1}'$, left-subtree$_{S_{m-1}''}(v_m) = J_{m-1}$ and left-subtree$_{T_{m-1}''}(v_m) = K_{m-1}$. Note that $S_m = \text{move-to-root}(S_{m-1}'' , u_m)$ and $T_m = \text{move-to-root}(T_{m-1}'' , u_m)$ by Lemma D.12. Because $v_m$ is not on the access path for $u_m$ in either $S_{m-1}''$ or $T_{m-1}''$,
the subtree rooted at \( v_m \) remains unchanged in \( S_m \) and \( T_m \) from \( S_{m-1}^u \) and \( T_{m-1}^u \), respectively. In particular, left-subtrees \( S_m(v_m) = J_{m-1} \) and left-subtree \( T_m(v_m) = K_{m-1} \). Because \( u_m \) now has highest priority, \( s_m > t_m \), and \( J_m = \text{left-subtrees}_m(v_m) \) and \( K_m = \text{left-subtree}_m(v_m) \). Thus, \( J_m = J_{m-1} \), \( K_m = K_{m-1} \), and (b) holds for access \( m \).

**Case II:** \( z_m \notin I_{m-1} \). If \( u_m = x \) then \( u_i = v_i = x \), \( K_m = J_m = \emptyset \), \( I_m \) contains every node in \( S \), and \( S_m = T_m \) by Remark D.2. Now suppose \( z_m \neq x \).

Without loss of generality, suppose \( z_m < x \), so that \( u_m = z_m \). (The other case is symmetric.) Let \( S_{m-1}' \) be formed by substituting \( \hat{J}_{m-1} = \text{move-to-root}(J_{m-1}, u_m) \) and \( \hat{K}_{m-1} = \text{move-to-root}(K_{m-1}, u_m) \) for \( J_{m-1} \) and \( K_{m-1} \) in \( S_{m-1} \) and \( T_{m-1} \), respectively.

Let \( J_{m-1}' = \text{right-subtree}_{m-1}(u_m) \) and \( K_{m-1}' = \text{right-subtree}_{m-1}(u_m) \). Define the intermediate top trees \( \tilde{I}_{m-1} = \text{augment-top}(\text{left-subtree}_{m-1}(u_m), u_m) \) and \( \tilde{I}_{m-1}' = \text{augment-top}(\text{left-subtree}_{m-1}(u_m), u_m) \).

The priorities of every key in \( \tilde{I}_{m-1} \) and \( \tilde{I}_{m-1}' \) are identical, thus \( \tilde{I}_{m-1}' = \tilde{I}_{m-1} \). If \( I_{m-1} = \emptyset \) then let \( \tilde{I}_{m-1}' = \tilde{I}_{m-1} \). Otherwise, let \( \tilde{I}_{m-1} \) be the connected subtree of the root of \( S_{m-1} \) and \( T_{m-1} \) whose nodes are in \( I_{m-1} \), and form \( \tilde{I}_{m-1}' \) by starting from \( \tilde{I}_{m-1} \) and attaching \( \tilde{I}_{m-1} \) as the left subtree of \( v_m \) if \( s_{m-1} > t_{m-1} \) and attaching it to \( u_m \) if \( s_{m-1} < s_{m-1} \). Note that the keys in \( \tilde{I}_{m-1}' \) are precisely those in \( I_m \), and that \( J_{m-1}' \)'s and \( K_{m-1}' \)'s keys are precisely those lying between \( u_m \) and \( v_m \). By Lemma D.12, \( S_m = \text{move-to-root}(S_{m-1}', z_m) \) and \( T_m = \text{move-to-root}(T_{m-1}', z_m) \), and \( z_m \in I_m \). Thus, we may apply the same case analysis as is used for \( z \in I_m \) to \( S_{m-1}' \) and \( T_{m-1}' \).

**Proofs for Section D.2**

**Lemma D.15.** If \( y \) is a child in \( V \), \( V_y \) denotes the subtree rooted at \( y \) in \( V \), and \( V_y^+ = \text{augment-top}(V_y, \text{parent}_V(y)) \), then \( \ell_V(z) = \ell_V(y) + \ell_{V_y^+}(z) - 2 \) for \( z \in V_y \).
Proof. By Lemma D.14, every crossing node of $y$ in $V$ that is a strict ancestor of $y$ in $V$ is also a crossing node for $z$. Furthermore parent$_{V_y^+}(z') = \text{parent}_V(z')$ and left-child$_{V_y^+}(z') = \text{left-child}_V(z')$ and right-child$_{V_y^+}(z') = \text{right-child}_V(z')$ for $z' \in V_y$. Thus, for $z' \in V_y$, $z' \in \text{crossing-nodes}_{V_y^+}(z)$ if and only if $z' \in \text{crossing-nodes}_V(z)$. The nodes $y$ and parent$_V(y)$ are double counted, necessitating an additive factor of $-2$.

Proof of Lemma D.2. By Remark D.2, we may assume that $J_i \neq \emptyset$ and $K_i \neq \emptyset$, and thus formula (a) is well-defined for $0 \leq i \leq m - 1$.

$J^+_0 = S_0$ and $K^+_0 = T_0$, so the formula is true by construction when $i = 0$. Otherwise, by Lemma D.15, $\ell_{S_i}(y) = \ell_{S_i}(\text{root}(J_i)) + \ell_{J^+_i}(y) - 2$ and $\ell_{T_i}(y) = \ell_{T_i}(\text{root}(K_i)) + \ell_{K^+_i}(y) - 2$. Let $a_i = \text{root}(J_i)$ and $b_i = \text{root}(K_i)$, and let $g_i = \text{parent}_{S_i}(a_i)$. By construction, parent$_{T_i}(b_i) = g_i$, and $a_i$ and $b_i$ are children of the same direction. By Lemma D.1(a), access-path$_{S_i}(g_i) = \text{access-path}_{T_i}(g_i)$. Thus, $\ell_{S_i}(g_i) = \ell_{T_i}(g_i)$ and $\ell_{S_i}(\text{root}(J_i)) = \ell_{T_i}(\text{root}(K_i))$. By Lemma D.1(b), $J^+_i$ and $K^+_i$ have the same nodes. Thus, formula (a) holds for $0 \leq i \leq m - 1$ and $y \in J^+_i$. If $y \notin J^+_i$ then $y \in I_i$, and $\ell_{S_i}(y) = \ell_{T_i}(y)$ by Lemma D.1(a). Thus, the formula holds for $y \in S$.

Part (b) also follows from Lemma D.1(a): if $z_{i+1} \in I_i$ then only an access to the less recently accessed of $u_i$ or $v_i$ can change root($J^+_i$) from $u_i$ to $v_i$ or vice versa. Otherwise, $J^+_i$ and $K^+_i$ remain the same. By construction, $g_i$ is the less recently accessed of $u_i$ and $v_i$, so the lemma holds.

Proof of Lemma D.3. Since $K_i = \text{move-to-root}(J_i, x)$ whenever $P_i \setminus \{x\} \neq \emptyset$ by Lemma D.1(c), we can apply Lemma D.11 to obtain (a). Statement (b) follows by an application of Lemma D.15 similar to the one found in Lemma D.2(a).

113
Appendix E

Behind the Scenes

Very few people read Ph.D. theses; fewer still read appendices. This section is full of easter eggs for those who do. It contains no substantive results, and it will make little sense to anyone who has not already read the remainder of the thesis. It is a rough chronology; a sampling of the dead ends and serendipitous discoveries from which this work emerged. While I see no obvious applications of these comments, my work was strongly influenced by my colleagues’ anecdotal observations, and perhaps mine too will be of some use to others.

The Birth of Approximate Monotonicity and Simulation Embeddings

This thesis, perhaps ambitiously, directly attacks the dynamic optimality conjecture itself, as opposed to focusing on various corollaries and related problems. However, I began my investigation with far more modest aims. Specifically seeking to “test the waters” of this area of research by focusing on a narrow sub-problem, I began by examining the traversal conjecture.¹

¹I settled on this problem because, out of all of the material in Sleator and Tarjan’s original article [54], the traversal conjecture seemed to have a more “combinatorial” flavor, which stuck out at me.

The essential format of this conjecture is “for every starting tree, and every request sequence of a certain type (preorders), splaying takes linear time.” At this time, there
were two special cases where the traversal conjecture was known to hold. The first was with the sequential access theorem [61], whose format is “for every starting tree, and for a fixed request sequence (in-order access), splaying takes linear time.” The other, by Chaudhuri and Höft [9], is of the form “given an arbitrary preorder, splaying by starting from the fixed tree it defines takes linear time.” My idea for proving the traversal conjecture, captured by the following theorem, was to “combine” the ideas in these two sub-results:

**Theorem E.1.** The traversal conjecture would be a consequence of the following two statements:

1. splaying an arbitrary preorder starting from a left spine takes linear time, and
2. removing requests from the request sequence decreases the cost of splaying. (This was, of course, the precursor of approximate monotonicity.)

**Proof.** Consider splaying the augmented sequence $Z$ starting from $T'$, where $Z$ comprises $\text{preorder}(T)$ prepended with a list of $T$'s keys sorted in increasing order (i.e. a sequential access). A side-effect of splaying the sequential access portion of $Z$ is to transform $T'$ into the left spine in linear time [61]. (This observation eventually helped inspire tree transformations and simulation embeddings.) Thus, $\text{cost}(Z, T')$ is linear in $|T|$ if supposition 1 is correct. Since $\text{preorder}(T)$ is a subsequence of $Z$, the traversal conjecture would follow by supposition 2. □

I felt that approximate monotonicity, being a more “basic” and combinatorial property, would be easier to prove, so I immediately set about trying to do so. I quickly ran into a wall, and even basic special cases seemed unattainable. I emphatically do not recommend this approach. It took me a while to even realize that Splay’s subsequence overhead was greater than one. The smallest counter-examples are not so simple.
diately asked Bob what he knew about this “subsequence property,” and he replied with a blank stare.

Now, anyone who knows Bob Tarjan will immediately understand that it is an exceedingly rare occurrence for him to reply with a blank stare when asked a question about a topic that he is intimately familiar with. I immediately, instinctively, knew that approximate monotonicity must somehow be important. I set about trying to find out how.

I soon realized that if we could induce Splay to perform arbitrary tree transformations then it would be possible to simulate arbitrary executions in the BST Model, and reduce the entirety of the dynamic optimality conjecture to approximate monotonicity. (This is what we later called a “simulation embedding.”) I spent a while playing with tree transformations. Eventually I built a method of turning a right spine into an arbitrary tree using augmented postorder sequences and constant depth splays. This was enough to establish that approximate monotonicity implied dynamic optimality.\(^5\) (Later, we discovered Lucas’ much simpler description of transformations using restricted rotations.) The day I got the details worked out was one of the most memorable of my Ph.D.

**Necessity and Additive Overhead**

The precursors of both Theorems 3.9 and 4.1 actually arose together around the same time as my ideas for attacking the traversal conjecture. Before I realized that Splay had subsequence overhead greater than one, I tried to prove that if Splay was not strictly monotone then it would be possible to amplify Splay’s behavior on non-optimal subsequences and demonstrate that Splay is not optimal. Thus, the ideas of amplifying overhead and of proving necessity were born from the same origin. The idea of amplifying additive overhead went through many formalizations to get to the

\(^5\)I had postorders on my mind due the presence of preorders in the traversal conjecture.
current version. I am still not sure I got it entirely to my liking, although I am reasonably satisfied with how Chapter 4 currently looks.

Proposal for Proving Optimality

Understanding the intuition behind Wilber’s crossing bound was actually one of the most conceptually difficult parts of this research for me. I have tried very hard to convey my understanding in this work. I first became interested in this bound after reading Kozma’s description [35]. He noted that the crossing lower bound could be computed by counting the number of zig-zags on the access paths when executing an instance with Move-to-Root. This was also a precursor to the idea of counting Splay’s zig-zig and zig-zag steps separately.

I got the idea that the crossing bound is (approximately) monotone from numerical experiments, after implementing it in Python using Wilber’s original description. I worked out the formalities much later on.

The idea of using a potential to smooth out heap-order violations came from playing with Galles’ data structures visualizer. I noticed that, regardless of the initial tree, splaying the same permutation tended to result in qualitatively similar final trees. I later realized this must be due to the modified heap-order properties of Splay, but the observation had perplexed me for quite some time.
Bibliography


120


