A New Path from Splay to Dynamic Optimality*

Caleb Levy† Robert Tarjan‡

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 [27], 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.

We offer the first *systematic* proposal for settling the dynamic optimality conjecture. At the heart of our methods is what we term a *simulation embedding*: a mapping from *executions* to lists of keys that induces a target algorithm to simulate the execution. We build a simulation embedding for Splay by inducing it to perform arbitrary subtree transformations, and use this to show that if the cost of splaying a sequence of items is an upper bound on the cost of splaying every subsequence thereof, then Splay is dynamically optimal. We call this the *subsequence property*. Building on this machinery, 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. As a corollary, the subsequence property is also a *necessary* condition for dynamic optimality. The subsequence property also implies both the *traversal* [27] and *deque* [30] conjectures.

The notions of simulation embeddings and bounding additive overheads should be of general interest in competitive analysis. For readers especially interested in dynamic optimality, we provide an outline of a proof that a lower bound on search costs by Wilber [32] has the subsequence property, and extensive suggestions for adapting this proof to Splay.

1 Introduction

1.1 Terminology. By an *instance*, we mean a list of tasks together with a specified initial configuration. An *execution* for 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 items 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 to us will be Splay [27], although we will also make use of Move-to-Root [1]. 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, we can sometimes hope for 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.

1.2 Simulation Embeddings. The main issue is how to prove an algorithm is constant competitive without knowing what optimal executions "look like." We answer this question by constructing *simulation embeddings*, which combine two concepts.

The first starts with a simple observation: in many situations, we intuitively expect 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 has the *subsequence property* 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.

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

*Research at Princeton University partially supported by an innovation research grant from Princeton and a gift from Microsoft.

†Princeton University & Intertrust Technologies

‡Princeton University & Intertrust Technologies

[1] That is, the same constant applies to all instances.

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*. An algorithm with a simulation embedding has the *simulation property*.

An algorithm with the simulation property can 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 *also* has the subsequence property, then the cost of executing the original instance will never exceed a fixed multiple of the simulation's cost, and hence of the optimal cost. We conclude: algorithms with both the subsequence and simulation properties are constant-competitive!

We should emphasize that 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 has the subsequence property. Many other uses of simulations exist, but they do not fall under the purview of this work.

1.3 The State of Knowledge. 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.

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, or even approximately compute, the cost of an optimum binary search tree execution for an instance. There are several known lower bounds [10, 32], none known to be tight (though some 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 in which each task in the instance can request multiple keys *is* NP-Complete [10]. The theoretical and practical difficulties that 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.

The other major candidate on-line optimal binary search tree algorithm is called *greedy* (variously, Greedy-Future and Geometric Greedy). An *off*-line version of this algorithm was originally conjectured to be constant-competitive for binary search trees by Lucas [20]. 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 [10]. Since then, many of the interesting properties that first drew attention to Splay have been proved for Greedy, as well as some additional properties. See [18, Chapters 1 & 2] for a thorough survey.

Many of the techniques developed in this paper should also be applicable to Greedy, but we do not focus on Greedy, and applying our results in the geometric model would likely constitute another investigation in its own right.

1.4 Our Main Contributions. The following results, given in the beginning half of the paper (§3 and §4), will likely be of interest to the broadest audience.

- We show that the subsequence property is a sufficient *and* necessary condition for settling the dynamic optimality conjecture, the first "non-trivial" and conceptually "intuitive" condition we are aware of for this problem.
- We codify the notion of a simulation embedding, which has arisen in binary search tree algorithms. We have spent a great deal of effort presenting our simulation embedding for Splay in a manner that lends itself to generalization beyond problems for binary search trees.
- We formalize the notion of an algorithm being optimal with an "additive overhead" in the form of *transient bounds*, and show that Splay can only be optimal with additive overhead at most *linear* in the sum of the number of requests and the size of the initial tree.

The remainder of the paper will be of greatest utility to those interested in directly building on our results.

- We show that the traversal and deque conjectures are also implied by the subsequence property, and extend the simulation embedding for Splay to a much broader class of algorithms. (§5)
- We show Wilber's second lower bound function [32] on the cost of an instance has the subsequence property, and provide extensive commentary on how to adapt the proof of the subsequence property for Wilber's bound to Splay. (§6, §7 and §8).

1.5 Related Work. We know of two prior works that construct simulation embeddings for binary search trees. The earlier one is tucked away in Dion Harmon's Ph.D. thesis [13], where he uses the *geometric model* of binary search tree algorithms (made popular in [10]) to prove that a well-known *geometric greedy* algorithm is constant-competitive if it satisfies the subsequence property. The second investigation was undertaken by Luís Russo, who analyzed a simulation embedding for Splay using *potential functions* [23]. Neither of these works seem to have garnered as much attention as we feel they deserve, and we encourage others to take a close look at their methods.

Our manner of constructing simulation embeddings for Splay offers several advantages over [23]. The analysis is completely combinatorial, and more straightforward to carry through. The simplicity of our arguments also makes them readily generalizable (see §5.5). Furthermore, by building simulations using tree transformations, we are able to prove both the sufficiency and necessity of the subsequence property for dynamic optimality using the same machinery. We also note that both [13] and [23] declined to observe (or at least formalize) the *necessity* of the subsequence property.

We are aware of no published results that give an upper bound on the additive overhead for which a binary search tree algorithm can be optimal. We are somewhat surprised this issue has not received further attention, as it has likely applications to recent work on pattern-avoiding access in binary search trees [2], not to mention enabling our reductions of the deque and traversal conjectures to that of proving the subsequence property. From personal conversations, we are aware that John Iacono has independently considered some of the methods we employ in §4 in unpublished investigations.

Some of the ideas that we propose in "The Path Forward" (§7) resemble those that gave rise to *Tango Trees* [11]. This data structure was essentially constructed by turning Wilber's *first* lower 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. In §7, we argue for analyzing Splay by comparing portions of its cost with Wilber's *second* lower bound. In fact, the similarities run deeper, as Tango trees arose from an attempt to turn Wilber's *second* lower bound into an algorithm.

The proposed path forward also draws inspiration from several other sources. John Iacono's perspective on Wilber's second lower bound bound in [15, 16, and personal communication] was especially helpful, as were Wilber's own thoughts about BST executions [32]. We were also influenced by the *global view* of binary search tree algorithms [3] and Joan Lucas' comments on the difficulty of applying standard inductive proofs to binary search tree algorithms [20].

Our proof that Wilber's bound has the subsequence property is novel.

2 Preliminaries

2.1 Binary Search Trees and Rotations. Our terminology for describing binary search trees is essentially paraphrased from [18, Chapter 1.3]. A *binary tree* T comprises 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 either to a different node or to a childless `null` object. Every node in T, spare for the root, has a single *parent* node of which it is a child. (The root has no parent). By the *subtree rooted at node* x we mean the set comprised of x along with all other nodes reachable by starting from x and following left and right pointers. Nodes thus have *left* and *right subtrees* rooted respectively at their left and right children. (Subtrees are *empty* for `null` children). 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 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 `null`. The *cost* of this search is set by convention to be the number of nodes, including `null`, encountered prior to termination (this is called the *length* of the search path). If k is in the tree then the *depth* of the node containing it is the number of pointers followed during the search.

We will use a local restructuring primitive called a *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 1).

2.2 Splay and Move-to-Root. The main subject of our study is the Splay algorithm, first described in [25]. 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 `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,

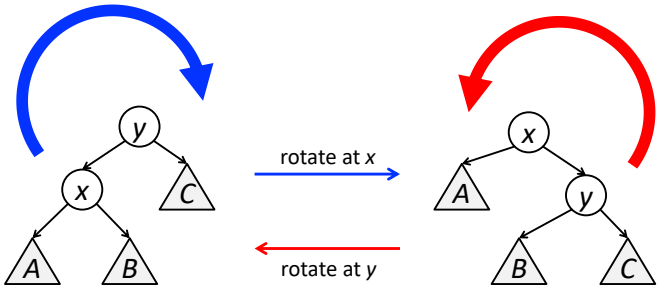

Figure 1: Rotation at node x with parent y (left), and reversing the effect by rotating at y (right).

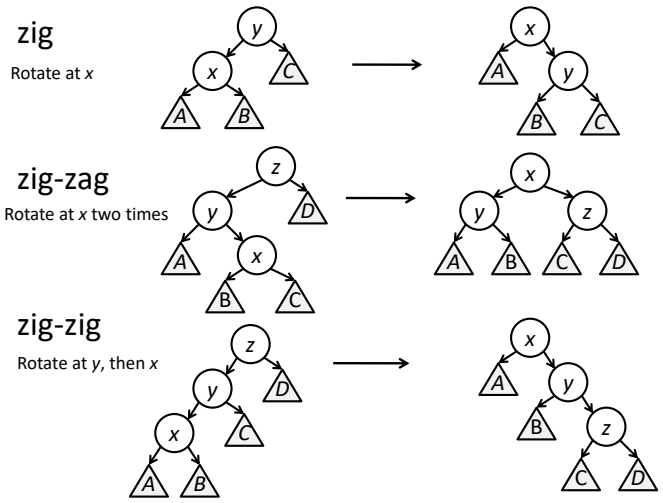

Figure 2: A splaying step at node x. Symmetric variants not shown. Triangles denote subtrees.

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 [27] 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 referred to as *splaying at* x, or simply *splaying* x. The three cases are depicted in Figure 2.

We will also make use of a second algorithm called Move-to-Root in our analysis. Move-to-Root is likely the simplest non-trivial algorithm fitting in Sleator and Tarjan's cost model. First search for a key k, and then repeatedly rotate the returned node x until it becomes the root. Allen and Munro were the first to analyze this algorithm [1]. Its similarity to Splay is striking, and it is one of Splay's progenitors [27]. Indeed, we will argue later that Splay's is connected with Move-to-Root on many levels.

2.3 The Binary Search Tree Model. The following model for binary search tree executions is based on "transition trees." It is cost-equivalent to other models defined throughout the literature. Our definition most closely resembles [18, 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. 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 3.

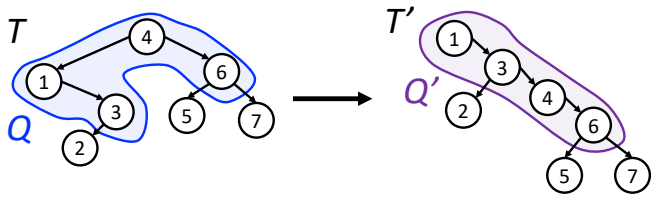

Figure 3: Example of a subtree substitution of cost four.

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.

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 $1 \le i \le m$:

- 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 $x_i = \text{root}(Q'_i)$, and
- 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 denote the execution by this sequence. Figure 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, and we define

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

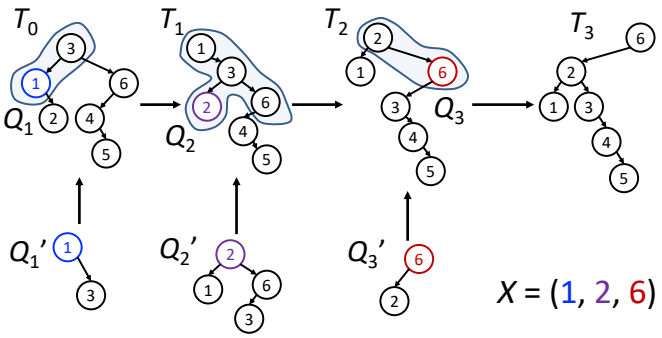

Figure 4: Example of an execution for an instance $X = (1, 2, 6)$. The total cost is $|Q_1'|+|Q_2'|+|Q_3'| = 2+4+2 = 8$.

REMARK 2.1. We will sometimes abuse notation and let $\mathrm{OPT}(X, T)$ also refer to a sequence of transition subtrees 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 assume one is chosen arbitrarily.

REMARK 2.2. By [4, Theorem 43], we can assume without loss of generality that subtrees without requested keys can be ignored. 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. We therefore have $\mathrm{OPT}(X, T) \geq |T|$. Similarly, every execution produces at least one transition tree per requested item, hence $\mathrm{OPT}(X, T) \geq |X|$, where $|X|$ is the number of requests in X.

A few comments are in order. 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 requested item, and therefore the path connecting them (we call this the *access path*). This accounts for the cost of searching. We describe how to extend this model to encompass insertions and deletions in §5.2.

2.4 The Dynamic Optimality Conjecture. As intimated in the introduction, we may view binary search tree *algorithms* (such as Splay and Move-to-Root) as maps from *instances* to *executions*. More formally, an *algorithm* $\mathcal{A}$ is a map from instances (X, T) to a sequence of valid transition trees $Q_1', \ldots, Q_{|X|}'$ for executing request sequence X with starting tree T. We denote the cost of this execution by $\mathrm{cost}_{\mathcal{A}}(X, T)$. Note that we will be dealing almost exclusively with the cost of the *Splay* algorithm, hence $\mathrm{cost}(X, T)$ will always refer to the cost of *splaying* the keys of X with initial tree T. Also note that in our BST model, the cost of splaying a node is simply the length of the access path.

DEFINITION 2.2. An algorithm $\mathcal{A}$ is said to be *dynamically optimal* (or *constant competitive*) if there exists some constant $c \geq 1$ so that, for all sequences of keys X and all corresponding initial trees T, we have that $\mathrm{cost}_{\mathcal{A}}(X, T) \leq c\,\mathrm{OPT}(X, T)$. 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 adopt it in this paper.

3 A Simulation Embedding for Splay.

We show how to induce Splay to perform *restricted rotations*, as defined by [5], in constant time, and thereby transform a subtree arbitrarily in time proportional to the subtree's size. We use these tree transformations to build a *simulation embedding* for Splay in the BST Model, and establish as a consequence that the *subsequence property* implies *dynamic optimality*.

3.1 Restricted Rotations. We use the term *restricted rotation* to refer to rotating a node whose parent is either the root, or the root's left child.

THEOREM 3.1. ([6] AND [19]) *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 [5], using properties of Thompson's Group F, a group-theoretic concept related to binary search trees. Cleary and Taback improved this bound to $4n$ in [6]. Lucas presents Cleary's result in terms of the standard BST model [19]. She essentially describes an algorithm that uses restricted rotations to "unwrap" T_1 into a "flat" tree, which is transformed into T_2 by applying the flattening algorithm in reverse. Figure 5 contains a sketch of this algorithm.

3.2 Inducing Restricted Rotations. The next theorem, despite its utter simplicity, forms the basis of our entire method. To begin, we view splaying as a *function* that takes a tree T and a node $x \in T$ 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. We can define G_n, the *transition graph* for Splay on binary search trees

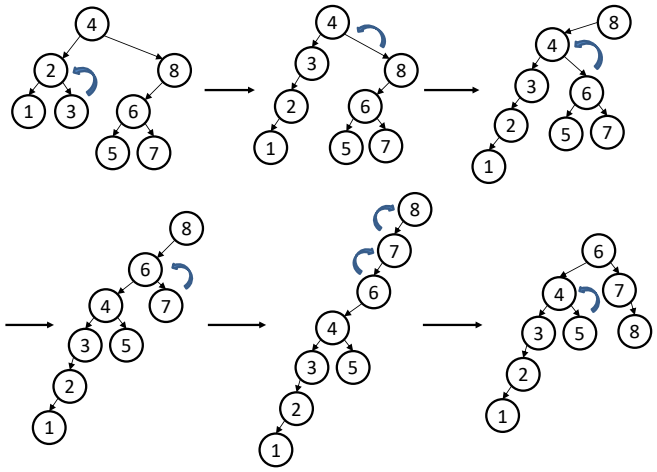

Figure 5: Applying restricted rotations to unwind a binary search tree as in [19].

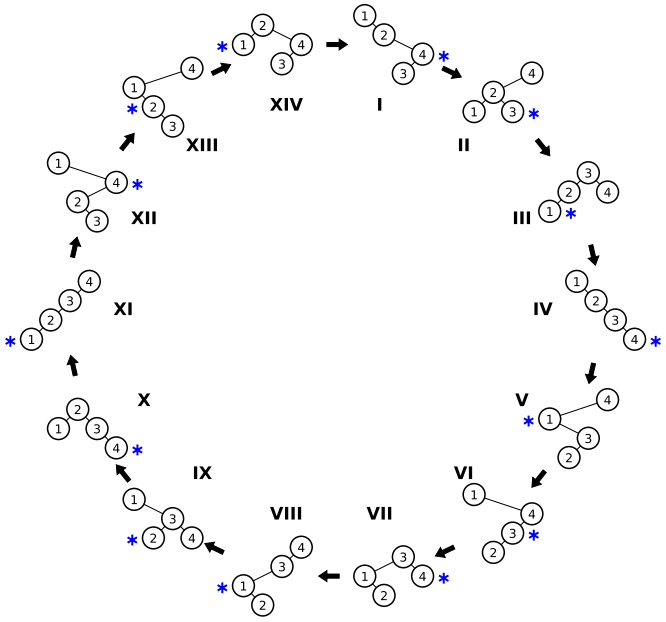

Figure 6: A Hamiltonian cycle in G_4. Splay at nodes marked by $*$ to convert one tree into the next.

of n nodes, as follows. Assign to every binary tree with n elements a node in G_n. For each $T \in G_n$ and $x \in T$ draw an edge from T to $splay(T, x)$.

THEOREM 3.2. *G_4 is strongly connected.*[2]

Proof. The reader may feel free to verify by hand the Hamiltonian cycle of G_4 depicted in Figure 6. □

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 [21] and [22].

Theorem 3.2 opens the door to *inducing* restricted rotations in trees with four nodes. We can see this most easily by example. Suppose we start with tree **XI** in Figure 6, 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 6 and splay at the indicated keys to transform accordingly.

While Figure 6 provides a striking visual depiction of this process, we can use more pedestrian methods to improve the cost overheads of performing restricted rotations in this fashion.

[2]Note that G_3 is *not* strongly connected.

THEOREM 3.3. *A sequence of at most* 5 *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.

3.3 Tree Transformations. 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 3.2 and Remark 3.2 is that Splay has the transformation property! Stated more precisely:

THEOREM 3.4. *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 3.1, it takes at most $4n$ restricted rotations to transform T into T'. By Theorem 3.3, 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 $4n * 5 * 4 = 80n$. □

The proof of Theorem 3.4 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 [5], 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.

3.4 Constructing the Embedding. 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 3.4 is a recipe for inducing Splay to perform *subtree transformations*. This can immediately be used to build a simulation embedding, $\mathcal{S}$, 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 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 $\mathbb{T}(T, T')$ denote the sequence of keys generated by using the process described in Theorem 3.4 for transforming T into T'.[3] When $T = T'$, $\mathbb{T}(T, T') \equiv \text{root}(T)$. We refer to $\mathbb{T}(T, T')$ 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 $\mathcal{S}$ is 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.

DEFINITION 3.1. $\mathcal{S}(E) \equiv \mathbb{T}(Q_1, Q'_1) \oplus \cdots \oplus \mathbb{T}(Q_m, Q'_m)$.

THEOREM 3.5. *The map $\mathcal{S}$ 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 $1 \le i \le m$ 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 $\mathcal{S}(E)$. Finally,

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

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 $\mathcal{S}$ is a simulation embedding for Splay. □

3.5 The Subsequence Property.

DEFINITION 3.2. A BST algorithm $\mathcal{A}$ is said to satisfy the *subsequence property* if there is some constant $a > 0$ so that for all requests X, all corresponding initial trees T, and all subsequences Y of X, we have $\text{cost}_{\mathcal{A}}(Y, T) \le a\,\text{cost}_{\mathcal{A}}(X, T)$. We refer to the constant a as the *overhead*.

Note that a subsequence is not necessarily contiguous. We form a subsequence Y by any sequence of deletions of items from X. Readers will likely anticipate the next result.

THEOREM 3.6. *If Splay has the* subsequence property *then it is dynamically optimal.*

Proof. Suppose Splay has the subsequence property with overhead a, and let (X, T) be an instance of a BST problem. Let Z denote the sequence of subtrees generated by $\text{OPT}(X, T)$. Note that

$$\text{cost}(X, T) \le a\,\text{cost}(\mathcal{S}(Z), T) \le 80a\,\text{OPT}(X, T).$$

Theorem 3.5 gives us the second inequality, and ensures us that X is a subsequence of $\mathcal{S}(Z)$. The first inequality is due to the subsequence property. □

It is worth asking whether we even need the subsequence overhead a. At least in the case of Splay, it is indeed required. For the simplest example, let T be a left path 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 overhead $a \ge 3/2$. With a little more effort, we can get a slightly better lower bound on a. Let T be a left path 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 $\text{cost}(X, T) = 2^k + O(k)$ while $\text{cost}(Y, T) = 2 \cdot 2^k + O(k)$, hence $a \ge 2$ for Splay. We conjecture that this ratio is tight.

[3] 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.

4 Transient Bounds.

4.1 Additive Overheads. It is conceivable, a priori, that an algorithm $\mathcal{A}$ can be dynamically optimal, but with some "startup overhead." For example, if n denotes the size of T then perhaps $\text{cost}_{\mathcal{A}}(X,T) = O(\text{OPT}(X,T) + n \lg\lg n)$. 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 path. We can formalize this as follows.

DEFINITION 4.1. We say that an algorithm $\mathcal{A}$ is dynamically optimal with *transient bound* $g : \mathbb{N} \to \mathbb{N}$ if there exists some positive constant c so that for every request sequence X and corresponding initial tree T we have $\text{cost}_{\mathcal{A}}(X,T) \leq c(\text{OPT}(X,T) + g(|T|))$.

REMARK 4.1. If $\mathcal{A}$ 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 $\text{cost}_{\mathcal{A}}(X,T) = O(\text{OPT}(X,T))$. In fact, this can be used as an alternative definition of transient bounds.

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 [27], via the "balance theorem". They prove that for a sequence of m keys in a binary search tree with n nodes Splay 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* [7] 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 [8, 9].[4]

Some consideration has also been devoted to GreedyFuture's transients. In particular, it is conjectured in [10] that GreedyFuture's execution cost never exceeds the optimal by more than an additive factor of $n/2$. Finally, Iacono describes a "proof-of-concept" algorithm in [16] which is dynamically optimal under certain conjectured assumptions with an (admittedly ludicrous) super-exponential additive overhead in initial tree size.[5]

The observations of §3.5 already give us something of a "lower bound" on Splay's transients, which must absorb at least $\Omega(n)$ additional cost for Splay as compared with OPT due to the subsequence discrepancy.[6] Sleator and Tarjan optimistically speculated an $O(n)$ *upper bound* on the additive overhead separating Splay from OPT [27]. Remarkably, their guess *must* be right for the conjecture to be true, as we shall see in the next section.

4.2 Amplifying Transients. We prove in this section that $O(|T|)$ is an upper bound on the permissible functional form of *any* additive overhead in initial tree size with which Splay can be dynamically optimal. Equivalently, we show that proving that Splay is dynamically optimal for *some* additive overhead automatically establishes it is optimal with overhead $O(|T|)$.

The formalities require care, but the idea is quite straightforward. Start with request sequences and initial trees inducing such "transient" effects, and augment the request sequences with *transformation sequences* returning the trees at the end of Splay's executions back to their initial states. By repeating the augmented sequences we can produce instances of *arbitrary* length on which Splay is uncompetitive.

Before proceeding to the proof, it is important to remember that the optimality of an *algorithm* is never defined for any *particular* instance. To illustrate this, we note that $\text{cost}_{\mathcal{A}}(X,T) \leq |T|\,\text{OPT}(X,T)$ for every algorithm $\mathcal{A}$, hence for *fixed* initial tree size $|T|$, *every* algorithm is $|T|$-competitive. The problem, of course, is that $|T|$ can be of arbitrary size, hence (obviously) not all algorithms are necessarily dynamically optimal. This elementary fact is a reminder that Definition 2.2 is only meaningful for an *infinite family* of request sequences and corresponding initial trees that contains arbitrarily large trees. Hence a proof that an algorithm is not optimal must make use of *sequences* of instances, and show that the algorithmic execution costs of the terms in the sequence *asymptotically* diverge from those of OPT. More formally, an algorithm $\mathcal{A}$ is *not* optimal whenever there is a series $\{X_1, X_2, \dots\}$ of request sequences and corresponding initial trees $\{T_1, T_2, \dots\}$ such that $\lim_{n\to\infty} \text{cost}_{\mathcal{A}}(X_n,T_n)/\,\text{OPT}(X_n,T_n) = \infty$.

The next theorem shows that if *any* sequence of instances separates Splay's cost from OPT then Splay

[4] Note that this does not eliminate the possibility of $O(n \log n)$ additive overhead for Splay, since we can have a request sequence whose dynamic finger bound is $O(n \log n)$ yet whose optimal execution cost is $O(n)$. See, for example, [14].

[5] Iacono uses a multiplicative weights update method where each "expert" is a member from a certain class of BST algorithms, ensuring that if any member of this class is optimal then so is his algorithm.

[6] We can, if so desired, simply absorb this into OPT (see Remark 2.2), giving an overhead equal to zero. The observations of Section 7 will provide natural reasons to maintain this as a separate overhead.

cannot be optimal for any transient bound. More precisely:

THEOREM 4.1. *Suppose we are given request sequences* $\{X_1, X_2, \dots\}$ *and corresponding initial trees* $\{T_1, T_2, \dots\}$. *If*

$$\lim_{n\to\infty} \frac{\text{cost}(X_n, T_n)}{\text{OPT}(X_n, T_n)} = \infty$$

then there is no *transient bound* $g : \mathbb{N} \to \mathbb{N}$ *and corresponding constant* c *so that for* all *instances* X *and corresponding initial trees* T *we have* $\text{cost}(X, T) \leq c(\text{OPT}(X, T) + g(|T|))$.

Before beginning the proof, Theorem 4.1 is worth translating back into the language of additive overheads. Suppose, hypothetically, that there is a sequence of instances $\{(X_1, T_1), (X_2, T_2), \dots\}$ for which $|T_n| = n$ such that $\text{cost}(X_n, T_n) = \Theta(n \log n)$ while $\text{OPT}(X_n, T_n) = \Theta(n)$. *A priori* this is compatible with the possibility that $\text{cost}(X, T) = \Theta(\text{OPT}(X, T) + |T| \log |T|)$. But as we shall see, this sequence of instances would allow us to construct request sequences of *any* length that cost Splay $\Theta(\log n)$-times as much to execute as OPT.

Proof. By contradiction. Suppose Splay does indeed have such a transient bound g with constant c. By Remarks 2.2 and 4.1, we have for all sequences X of length greater than $g(|T|)$ that

$$\begin{aligned} 2c\,\text{OPT}(X, T) &\geq c(\text{OPT}(X, T) + g(|T|)) \\ &\geq \text{cost}(X, T), \end{aligned}$$

or equivalently,

$$\frac{\text{cost}(X, T)}{\text{OPT}(X, T)} \leq 2c.$$

We will construct sequences of length greater than $g(|T|)$ violating the above bound.

Let V_n be the the tree produced by splaying the keys X_n starting from T_n, and define the augmented sequence $U_n = X_n \oplus \mathbb{T}(V_n, T_n)$, where $\mathbb{T}(V_n, T_n)$ is the transformation sequence that makes Splay transform V_n into V_n. Denote by $k * U_n$ the sequence U_n repeated k times. We make the following observations about the behavior of Splay:

- $\text{cost}(U_n, T_n) \geq \text{cost}(X_n, T_n)$, since U_n merely consists of requests *appended* to X_n.
- $\text{cost}(k * U_n, T_n) = k\,\text{cost}(U_n, T_n)$, where the equality follows by construction of U_n, which resets the tree to T_n, making each repetition an identical execution.

Next, we need bounds for the optimal costs of two sets of instances, both of which are established by first constructing executions that satisfy the bounds, and then noting that OPT has cost lower than any execution. The bounds are:

- $\text{OPT}(U_n, T_n) \leq \text{OPT}(X_n, T_n) + 81|T_n|$. The upper bounding execution is built by appending to a sequence of transition trees for $\text{OPT}(X_n, T_n)$ the trees formed by splaying the paths to the nodes of $\mathbb{T}(V_n, T_n)$, except with the first such path replaced by the tree formed by splaying the first node of $\mathbb{T}(V_n, T_n)$ in V_n. The total lengths of the remaining paths are at most $80|T_n|$ by the transformation property (Theorem 3.4).
- $\text{OPT}(k * U_n, T_n) \leq k * (\text{OPT}(U_n, T_n) + |T_n|)$. The execution is constructed as follows. Let B_n denote a sequence of transition trees for $\text{OPT}(U_n, T_n)$. Define B'_n to be identical to B_n, except the first transition tree of B'_n includes the *entirety* of the tree formed by substituting the first transition tree in B_n into T_n. The execution comprises of B_n followed by $k - 1$ repetitions of B'_n. The cost of B'_n is at most $|T_n|$ greater than B_n, hence the inequality.

Combining the two bounds on OPT gives

$$\begin{aligned} \text{OPT}(k * U_n, T_n) &\leq k(\text{OPT}(X_n, T_n) + 82|T_n|) \\ &\leq 83k\,\text{OPT}(X_n, T_n), \end{aligned}$$

where the last inequality follows from Remark 2.2.

Finally, we can define a lower bound function h on the ratio between the execution costs of Splay and OPT for $k * U_n$ by

$$\frac{\text{cost}(k * U_n, T_n)}{\text{OPT}(k * U_n, T_n)} \geq \frac{1}{83} \frac{\cancel{k}\,\text{cost}(X_n, T_n)}{\cancel{k}\,\text{OPT}(X_n, T_n)} =: h(n).$$

Notice that h is *independent* of k. Since $h(n) \to \infty$ as $n \to \infty$, we can choose n_0 so that $h(n_0) > 2c$. Pick some $k_0 > g(n_0)$. The request sequence $k_0 * U_{n_0}$ with starting tree T_{n_0} violates the presumed transient bound. □

REMARK 4.2. Kurt Mehlhorn has pointed out that this theorem's contrapositive is perhaps more readily understandable: a proof showing Splay is optimal with *some* additive overhead suffices to show optimality for *every* instance. We have retained our framing of Theorem 4.1 as a negative result because it is more suggestive of the ideas in 7.5.

4.3 Necessity of the Subsequence Property. We can *almost* immediately use 4.1 to show that Splay *must* have the subsequence property in order to be dynamically

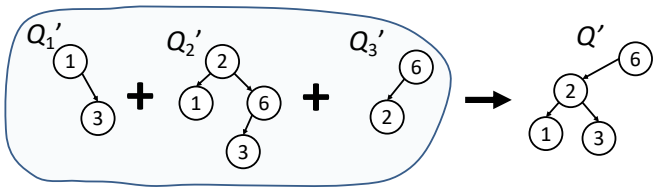

Figure 7: Example of eliding three transition trees.

optimal. However, we note the critical fact that Splay cannot have the subsequence property if OPT does *not* have the subsequence property. Fortunately, this is one of a select few properties that is relatively easy to prove about optimum BST executions.

THEOREM 4.2. *Let Y be a strict subsequence of $X = (x_1, \ldots, x_m)$. For all valid initial trees T containing the keys in X, $\mathrm{OPT}(Y, T) < \mathrm{OPT}(X, T)$.*

Proof. 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 $\mathrm{OPT}(X, T)$. 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 $j \le i \le k$ (see Figure 7). 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 $|Q| \le |Q_j| + \cdots + |Q_k| - (k - j)$. 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 subsequent contiguous subsequence of requests in X that are missing from Y to form a valid execution for Y costing less than X. □

THEOREM 4.3. *If Splay does not have the subsequence property then it is not dynamically optimal.*

Proof. Suppose we have request sequences $\{X_1, X_2, \ldots\}$ with starting trees $\{T_1, T_2, \ldots\}$ and corresponding subsequences $\{Y_1, Y_2, \ldots\}$ such that

$$\lim_{n\to\infty} \frac{\mathrm{cost}(Y_n, T_n)}{\mathrm{cost}(X_n, T_n)} = \infty.$$

Because OPT has the subsequence property by Theorem 4.2, we have $\mathrm{OPT}(Y_n, T_n) \le \mathrm{OPT}(X_n, T_n) \le \mathrm{cost}(X_n, T_n)$, hence

$$\lim_{n\to\infty} \mathrm{cost}(Y_n, T_n)/\,\mathrm{OPT}(Y_n, T_n) = \infty.$$

By Theorem 4.1, Splay is not dynamically optimal. □

Of course since OPT has the subsequence property, *any* dynamically optimal algorithm must as well, in an asymptotic sense at minimum. Theorem 4.3, in a manner very similar to Theorem 4.1, provides the additional assurance that Splay must obey the subsequence property in a very strictly.

5 Extensions

5.1 Competitive Ratios. The subsequence property 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.[7] Similarly if, hypothetically, there is some sequence of instances $\{(X_1, T_1), (X_2, T_2), \ldots\}$ for which $\mathrm{cost}(X_n, T_n) = \Omega(\log\log n\,\mathrm{OPT}(X_n, T_n))$, then $\mathrm{cost}(X_n, T_n)/\,\mathrm{cost}(\mathcal{S}(\mathrm{OPT}(X_n, T_n)), T_n) = \Omega(\log\log n)$, 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: if $f(n) = \sup_{|T|=n, Y\subseteq X} \mathrm{cost}(Y, T)/\,\mathrm{cost}(X, T)$ and $g(n) = \sup_{|T|=n,X} \mathrm{cost}(X, T)/\,\mathrm{OPT}(X, T)$ then $f(n)/g(n) = \Theta(1)$.[8] The subsequence property and dynamic optimality are very much equivalent.

5.2 Insertion and Deletion. We can extend the BST model to cover insertions and deletions. To do so, we divide the tree into two sets of nodes: *visible* and *invisible*. The visible nodes always form a connected subtree of the root. The nodes of the starting tree all begin as visible. The keys which are not in the initial tree but will be inserted some time later are "pre-inserted" into the initial tree without adjustment as invisible nodes, by leaf insertion in insertion order.

To delete a visible node, an execution produces a transition tree where the node's successor is the root, its predecessor is the root's left child, and the node to

[7] This itself would be a major result, improving over the best-known bound, $\mathrm{cost}(X, T) = O(\log|T|\,\mathrm{OPT}(X, T))$, that was first derived in [27].

[8] Here, $Y \subseteq X$ denotes Y is a subsequence of X.

be deleted is its predecessor's right child (if one of the successor or predecessor is not present then the deleted node becomes the child of the other one, otherwise it becomes the root).[9] After substituting the subtree, the deleted node becomes invisible.

The execution cannot incorporate an invisible node into a transition tree until it is made visible by an insertion. To execute an insertion, an invisible child of a visible node is made visible, and then incorporated into a transition tree in the usual way. We assume without loss of generality that an inserted node is always a child of a visible node, or is at the root (otherwise we can create an alternate request sequence that satisfies this property and has an equivalent execution).

The representations of *insertion* and *deletion* are carefully constructed so that if Splay has the subsequence property in the model of Definition 2.1 then it will also be constant competitive with OPT on request sequences that allow these operations. We implement these operations as requests that impose certain restrictions on the transition trees employed by OPT. Since Splay can simulate arbitrary transition trees, it can simulate transition trees of restricted form as well. We thus never need to add insertions or deletions to the simulation embeddings for Splay to account for these operations.[10]

5.3 The Deque Conjecture. Consider the following problem. Start with tree T having integers 1 through n as elements, and consider a sequence of m of the following operations: delete($\min\{T\}$), delete($\max\{T\}$), insert($\min\{T\}-1$), and insert($\max\{T\}+1$). (We call these *deque* operations.) We can use our implementations of insertion and deletion to perform these operations in $O(n+m)$ amortized time in the BST model (see [4, Lemma 36] for details). In [30], Tarjan considered whether the total cost of performing deque operations via *splaying* costs $O(m+n)$, calling this the *deque conjecture*. If deque operations are implemented with our versions of insertion and deletion then by Theorems 3.6 and 4.1, the subsequence property implies that Splay executes these deque operations with cost $O(m+n)$.[11]

[9] Our definition of deletion is unusually strong: normally deletion is implemented by accessing the successor *or* predecessor, but not both. Our definition ensures that the subtree rooted at the deleted item contains no visible node, and can be implemented using a subtree transformation. We leave the analysis of standard deletion algorithms as an open problem.

[10] An extended version of this paper will feature a detailed proof.

[11] Tarjan's original method for using Splay to implement deque operations is slightly different [30]. We leave determining whether the subsequence property also implies that the *original* implementations satisfy the deque conjecture as an open problem.

5.4 The Traversal Conjecture. A corollary of Theorem 4.1 is that if Splay has the subsequence property (and hence is dynamically optimal) then it satisfies the *traversal conjecture*, which is as follows. Let T_1 and T_2 be two binary search trees on the same set of n keys, and let P be the sequence listing T_2's keys in *preorder* (the item in the root of T_2 first, followed by the items in the left subtree of T_2 in preorder, followed by the items in the right subtree of T_2 in preorder). The traversal conjecture [27] is that $\text{cost}(T_1, P) = O(n)$. The traversal conjecture would be an immediate consequence of Splay being dynamically optimal with linear transient bound, but would *not* necessarily follow from Splay being optimal with *super-linear* transient bound. Theorem 4.1 removes the difference.

5.5 Path-Based Algorithms. A reexamination of §3 and §4 reveals that all of our results apply to any algorithm that

1. has transition trees that are strict functions of the *binary encodings* of the search paths for the requested nodes, and
2. whose transition graph G_n is strongly connected for some $n \geq 3$.

These two properties guarantee 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.

We note that the transition graph of "Simple Splay," described in [27], is also strongly connected. Hence if Simple Splay has the subsequence property then it is dynamically optimal. These results do *not* immediately apply to *top-down* splaying, either regular or "simple" variant, as they do not have the transformation property (proof omitted). Interestingly enough, these results *do* apply to Move-to-Root, since we can use Move-to-Root to transform between three-node trees (proof omitted). It is easy to show that Move-to-Root is *not* optimal. Hence by Theorem 3.6 Move-to-Root does not have the subsequence property. (We leave finding the relevant family of sequence/subsequence pairs to the reader.) If nothing else, this indicates Theorem 4.3 could be a useful tool for showing algorithms are *not* optimal.

We also observe in particular that any *template based* algorithm of the types discussed in [28] and [12] whose transition graph G_n is strongly connected for some $n \geq 3$ is dynamically optimal if and only if it satisfies the subsequence property. We suspect, but have not tried to prove, that 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.

6 Wilber's Bound

In this section we describe *Wilber's second lower bound* in terms of the crossing nodes of Move-to-Root's execution, and characterize Move-to-Root as the unique algorithm that maintains the *treap* of keys with *heap-order* defined by *most recent access time*. This allows us to prove that Wilber's bound has the *subsequence property* with constant factor of 1.

6.1 Treaps and Move-to-Root.

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 *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 priorities and keys: the root is the node of greatest priority, and the left and right subtrees comprise the treaps of keys respectively smaller and greater than the root. Jean Vuillemin constructed treaps from permutations, setting each key's priority to its position in the the permutation [31]. The term "treap" is due to Seidel and Aragon [24].

Move-to-Root is the *unique* algorithm such that, after each access, the keys so far requested are arranged as a treap in which a key's priority is the most recent time at which it was requested. The action of Move-to-Root is equivalent to setting the requested key's priority to one greater than that of all other nodes, and then restoring the treap order. (The first item is accessed at time 1, and for initial tree T each node $x \in T$ is given initial priority of $r(x) - |T|$, where $r(x)$ is the index at which x appears in T's postorder.[12])

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 order. That Move-to-Root is the unique algorithm restoring heap order stems from the uniqueness of the treap for the given priorities.

[12]The postorder of T is a sequence comprising the items in the left subtree of T in postorder, followed by the items in the right subtree of T in postorder, followed by the item in the root of T.

6.2 Crossing Costs.

The lower bound ignobly named *Wilber 2* has never had more than supporting roles in various lemmas and corollaries scattered throughout the literature. That the "independent rectangle bound" from the geometric view of the BST model subsumes Wilber's seems to have further eroded its popularity (see [10] for details). But we think Wilber's outwardly inscrutable lower bound on the cost of binary search tree executions will yet play a crucial part in the story of the elusive dynamic optimality conjecture, and we give it a new debut. Note that we will have no use for the *first* lower bound defined in [32]. Hence from this point forward, *Wilber's bound* will always mean his *second* lower bound.

We reformulate Wilber's bound, as a count of critical keys chosen by their positions in the treap maintained by Move-to-Root. This formulation, implicit in Iacono's work [15] and alluded to in [16], is far more amenable to our purposes in §6.3 than the original.[13]

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 are 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 in $\ell_T(x)$.

Definition 6.1. (Crossing Cost) Let $X = (x_1, \ldots, x_m)$ be a request sequence with initial tree $T = T_0$, and let $T_1, \ldots, T_m$ be the after-trees of an execution E for this instance. The *crossing cost* of execution E is $\sum_{i=1}^{m} \ell_{T_{i-1}}(x_i)$.

Wilber's bound for instance (X, T), denoted by $\Lambda(X, T)$, is the crossing cost of Move-to-Root's execution of this instance.[14]

Theorem 6.1. (From [32].) $\Lambda(X, T) \leq \mathrm{OPT}(X, T)$.

Remark 6.1. Crossing nodes can be computed visually. Let $\mathrm{root}(T) = p_1, p_2, \ldots, p_k = x$ be the nodes of the access path for x in T. The first crossing node is p_1. Draw a line vertically from x up to infinity.[15] For $1 < i \leq k$, if the edge connecting p_{i-1} to p_i touches this line then p_i is a crossing node. See Figure 8 for an example of graphically calculating Wilber's bound.

[13]Proving equivalence between Wilber's and Iacono's formulations of Wilber's bound is, in our opinion, not entirely trivial, and appears to be folklore. We include a proof in an extended version of this paper.

[14]We have slightly changed the original bound to account for initial trees.

[15]A node's horizontal coordinate is its key's position in symmetric order; its vertical coordinate is the negative of its depth.

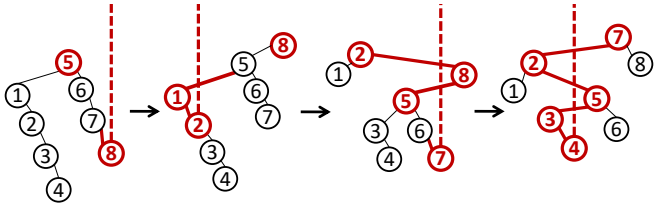

Figure 8: Execution of $X = (8, 2, 7, 4)$ starting from T by Move-to-Root. There are 14 total crossing nodes. Hence $\Lambda(X, T) = 14$.

6.3 Λ has the Subsequence Property. The treap-based view of Wilber's bound enables us to change the problem of proving the subsequence property from one of executing different request sequences starting from the same tree to one of executing the same request sequence starting from different trees. In detail, let Q be an initial tree for request sequence U, and let V be a subsequence of U formed by removing an access to a single request x from U, so that $U = W \oplus (x) \oplus Z$ and $V = W \oplus Z$ for some subsequence W of U. Let S be the tree formed by executing W with Move-to-Root starting from Q, and let T be formed by moving x to the root of S via Move-to-Root.

LEMMA 6.1. $\Lambda(Z, S) - \Lambda(Z, T) \leq \ell_S(x)$.

This innocuous looking lemma turns out to be very tedious to prove. The basic idea is to analyze how the structure of S and T evolve as requested items in Z are moved to the root using Move-to-Root. This requires both the *treap* and *rotation-based* views of Move-to-Root. Our proof, which involves a "brute-force" case analysis of how more than a dozen families of nodes change crossing depths depending on what is accessed,[16] is not very elegant.

The gory details will not be of immediate interest to our investigation, so we relegate them to §8 for reference. We caution that even the lengthy proof in §8 is merely an outline, as many of the observations contained within it are not trivial, and deserve proofs in their own right.

LEMMA 6.2. $\Lambda(U, Q) \geq \Lambda(V, Q)$.

Proof. The execution of both U and V by Move-to-Root starting from Q is identical for every request in W, hence $\Lambda(U, Q) = \Lambda(W, Q) + l_S(x) + \Lambda(Z, T)$ while $\Lambda(V, Q) = \Lambda(W, Q) + \Lambda(Z, S)$. The theorem follows from Lemma 6.1. □

[16] The families are the crossing nodes on the left and right side of x, the nodes between these crossing nodes on the path to x, the subtrees of each of these families, and corner cases at the root and the subtrees of x.

THEOREM 6.2. *Wilber's bound has the subsequence property with overhead one.*

Proof. The result follows from Lemma 6.2, by induction on the request deletions used to form Y from X. □

REMARK 6.2. After reviewing our paper, Kurt Mehlhorn kindly supplied us with a vastly simplified proof of Theorem 6.2 in the geometric model [10]. This simplified proof will be featured in an extended version of this paper. Similarly, a reviewer has pointed out that it is even easier to prove that the stronger Independent Rectangle Bound has the subsequence property. We have chosen to focus on Wilber's weaker lower bound because we suspect Theorem 6.2 can be used as a template for showing Splay has the subsequence property, as we discuss at length in §7. In addition, we strongly believe that Splay is not amenable to geometric analysis, hence we retain our longer non-geometric proof of Theorem 6.2.

7 The Path Forward

In this section, we put forward the following plan for structuring a proof that Splay is dynamically optimal, which builds on the tools developed in §3 - 6. First, we argue for separately analyzing the number of zig-*zags* (or *crossings*) and zig-*zigs* in Splay's executions. We then supply evidence of a strong relationship between Splay's crossing costs and Wilber's bound, and outline two avenues for making use of this relationship. The first option is to directly bound Splay's crossing costs by those of Move-to-Root. The second option is to adapt the proof of the subsequence property from Wilber's bound to Splay. We discuss the advantages and disadvantageous of each approach. Both approaches seem likely to require *potential functions* in their analysis, and we speculate on what this potential might look like. For the last part of the proof, we claim Splay's zig-zig costs should be bounded by its crossing costs, and offer our thoughts on how to prove this.

Disclaimer. Our in-depth suggestions are entirely speculative, and *our speculation could be partially or entirely wrong.* However, the suppositions *are* informed by extensive *in*formal evidence of some variety. This includes folklore knowledge gleaned from discussions with colleagues, comments from other papers, notebooks filled with the first author's failed attempts to prove various theorems, and most of all, numerical experiments run using the first author's (*thoroughly* tested) personal implementations of the algorithms discussed herein. The results of our numerical experiments in particular have been corroborated in large part by John Iacono. The authors of this paper are confident that, at a minimum,

any incorrect statements in the following section can only be wrong for *interesting reasons*.

7.1 Separating Λ' and ζ. We believe proving that Splay is dynamically optimal will require separately counting the number of zig-*zig* and zig-*zag* steps that it performs. First, we cover some formalities. In the sequel, for the purposes of both typographic clarity and conceptual harmony with our terminology for Wilber's bound, we refer to zig-*zags* as *crossings*, while retaining the terminology for zig-*zigs*.

Let $\Lambda'(X, T)$ denote the *crossing cost* (as in Definition 6.1) of *splaying* the nodes of X starting from tree T. Define the *zig-zig nodes* of an access as the nodes of the access path that are *not* crossing nodes. The *zig-zig cost* of an execution is the total number of zig-zig nodes along all the access paths encountered. We will use $\zeta(X, T)$ to denote the zig-zig costs of splaying the items requested in X starting from T. Note that, by definition, $\text{cost}(X, T) = \Lambda'(X, T) + \zeta(X, T)$.

Numerical experiments indicate that $\text{cost}(X, T) \leq 3(\Lambda(X, T) + |T|)$ for all request sequences X and corresponding T, giving preliminary impetus for exploring how Splay and Wilber's bound are related. But we have also observed a much stronger connection. We believe Splay's crossing costs are *equal* to Move-to-Root's up to lower order additive terms, and conjecture:

$$|\Lambda'(X, T) - \Lambda(X, T)| = O(|X| + |T|).^{17}$$

Note that up to an additive $O(|X|)$ term, $\Lambda'(X, T)$ is equal to the number of zig-*zag* splay steps enacted.

Empirically, ζ exhibits far more variability than Λ' with respect to Wilber's bound, providing at least some motivation to analyze these two quantities in different ways. This proposal also agrees well with Iacono's remarks from [16]. He notes that one way to prove Wilber's bound is tight is by designing an algorithm that can search for elements in time proportional to the number of crossing nodes on the path to that key in the treap maintained by Move-to-Root. His remarks align with our suspicion (see §7.7) that the BST model's true power stems from the existence of crossing nodes, and that zig-zig nodes are, in some sense, merely incidental "bookkeeping" nodes. Splay is of course *defined* by separating the zig-zig and zig-zag cases, lending further credence to the suggestion for distinct analyses.

Our basic proposal for analyzing ζ is to upper bound it by Λ'. We defer further discussion of ζ to §7.6, and focus our immediate attention on Λ'.

[17] The absolute value signs are *not* accidental. For a minimum example, let $X = (3, 1, 4, 2)$, and T the corresponding treap.

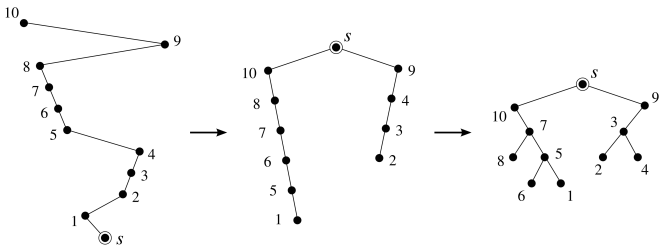

Figure 9: A global view of Splay trees. The transformation from the left to the middle illustrates Move-to-Root. The transformation from the left to the right illustrates Splay trees. (Figure and caption from [3]).

7.2 Connecting Λ and Λ'. As discussed in §6, Wilber's bound is defined by the crossing costs of Move-to-Root. That Splay's and Move-to-Root's crossing costs might be intimately tied becomes less surprising by looking at Splay in the *global view* of template algorithms. We quote the description of Splay from [3] verbatim. "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 9. 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}."

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 creates a "violation" in the heap ordering that Move-to-Root maintains with respect to most recent access time. As the executions of both Splay and Move-to-Root proceed for a given instance (X, T), 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. Tracking the creation and destruction of heap-order violations in Splay's executions rapidly becomes complicated, yet there is a clear intuitive basis for suspecting the Splay and Move-to-Root never become "too far" out of sync. As we shall see in §7.5, this strongly suggests the use of a *potential function* [29] for tracking these violations.

Of course, we have not specified what one might use this potential *for*. This question brings us to a fork in our path. In the next two sections, we discuss what each road forward entails.

7.3 Road I: Bounding Λ' by Λ. The obvious way forward is attempting to directly prove the empirically derived conjecture of the previous section that Splay's

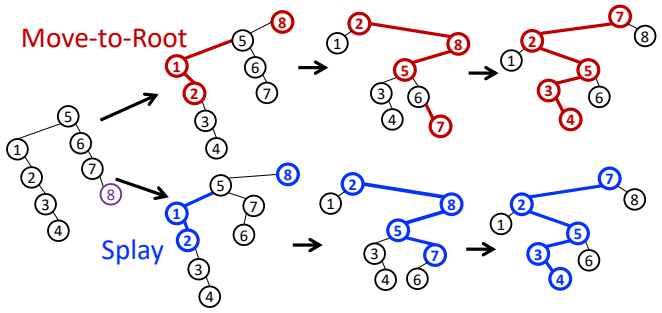

Figure 10: Comparison of crossing nodes for Splay and Move-to-Root when $X = (8, 2, 7, 4)$.

crossing costs equal those of Move-to-Root up to an additive linear overhead. Note that if our suspicion that $\zeta(X,T) = O(\Lambda'(X,T))$ in §7.6 is correct, then $\Lambda'(X,T) = \Theta(\Lambda(X,T))$ is in fact a *stronger* result than dynamic optimality: it would imply that Wilber's lower bound is *tight*.

The authors have tried at length to prove the above conjecture directly. Suffice it to say, this paper would have included a proof had we 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, we offer our comments on where the difficulties 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 10), albeit sometimes "offset" from each other in symmetric order by a small amount. *However*, the crossing nodes in P_i do not always appear directly in P'_i. 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, depending on the particular request sequence. Likely, this is due to the differing extents to which nodes violate the heap order in Splay 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 [20]. We believe this 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.

We think this matter merits further investigation. It would certainly be an elegant end to the conjecture if Splay's crossing costs essentially equal Wilber's bound, and it is quite possible the above impediments to proving this are a mere product of the authors' lack of imagination.

7.4 Road II: Λ' and Subsequences. We see only one other viable route to analyzing Λ': show, for any request sequence X, starting tree T, and subsequence Y of X, that $\Lambda'(Y,T) \leq \Lambda'(X,T) + O(|X| + |T|)$.[18]

There are several points in favor of this alternative approach. First, if $\Lambda'(X,T)$ *is* bounded by $\Lambda(X,T)$ up to additive terms then Λ' must *necessarily* have the subsequence property in this sense, from the subsequence property for Wilber's bound. Since numerical experiments indicate the former statement is true, we expect the latter to be true as well. Second, Λ' having the subsequence property is an (a priori) *weaker* statement than equating Λ' to Wilber's bound, hence it can still be true even if our numerical experiments comparing Λ and Λ' have misled us. Third, and most importantly (and unlike the situation in Road I) *we already have a reasonable starting point*: we can try to prove Λ' has the subsequence property by adapting the machinery of §6.3 from Wilber's bound to Splay. This critical task requires a key modification to Theorem 6.2: we must change the *form* of the induction.

The proof of Theorem 6.2 used in §6.3 has a specific structure: begin by establishing that the removal of a *single* item from a request sequence *always* decreases Λ (Lemma 6.2), and then induct on item removals to show that $\Lambda(X,T) \geq \Lambda(Y,T)$ for any subsequence Y formed by successively removing requests from X. This structure enables establishing the subsequence property for Wilber's bound *without* having to "compare" how Move-to-Root executes two *different* request sequences (key to Lemma 6.1).

This trick is unlikely to work for Splay. Numerical evidence strongly indicates that removing a request

[18] In fact, we merely require that $\Lambda'(Y,T) = O(\Lambda'(X,T) + |X| + |T|)$. The constant in front of $\Lambda'(X,T)$ does *not* need to be 1 for our proofs to carry through. We also note that there are cases in which Λ' does not obey the subsequence property in the strictest possible sense. For example, form T by inserting keys in the order $(5, 3, 1, 4, 2)$ into an empty tree, let $X = (2, 3, 5, 1, 4)$, and $Y = (3, 5, 1, 4)$. Here, $\Lambda'(Y,T) > \Lambda'(X,T)$

for an item with crossing depth κ in the splayed tree can *increase* Splay's crossing costs by $\omega(\kappa)$.[19] If true, the inductive step will fail. Instead, we must "directly" demonstrate that removing an *arbitrary* subset of requests increases Splay's crossing costs by at most $O(|X| + |T|)$ *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 $1 \leq d_1 < \cdots < d_k \leq |X|$. For $1 \leq i \leq m$, define $X_i = (x_1, \ldots, x_i)$, $Y_0 = ()$ and $Y_i = \big(y_1, \ldots, y_{i-\max\{j|d_j \leq i\}}\big)$. What we want is a double induction on both i and k that establishes $\Lambda'(Y_i, T) \leq \Lambda'(X_i, T) + O(i + |T|)$. This style of induction requires comparing how Splay executes *different* request sequences starting from *different* trees.

Prudence dictates reproving the subsequence property for Wilber's bound via the above-mentioned double induction *before* moving on to analyzing *Splay's* crossing costs. The proof of Theorem 6.2 via Lemma 6.1 is already unpleasantly complicated, so the more intricate version is, frankly, not going to be pretty. The good news is that since we *already know* the theorem is true, this alternate method of establishing it is, at least in principle, achievable.

We must still adapt this new proof from Λ to Λ'. 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. We discuss this matter in the next section.

We 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.[20]

7.5 A Potential 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 move-to-root tree, we still believe that $\Lambda'(X, T)$ and $\Lambda(X, T)$ are always tightly coupled (§7.3). Similarly, despite evidence that the removal of any *individual* request may greatly increase Λ', we remain convinced that removing an arbitrary subset of requests will not increase Λ' too much *in total* (7.4). Both of these conjectures are ripe for analysis via a potential function.

Those who have designed potential functions understand by experience that it can be a tricky and subtle business. We are fairly 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 perhaps the biggest remaining roadblock to proving dynamic optimality left open by this work. We can however, infer something very important. As noted in [17], 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. In our case, however, we have found Splay's additive overhead via Theorem 4.1, which informs us that the potential's range should not exceed $O(|X| + |T|)$, and in fact we strongly suspect that the maximum value of this potential is $O(|T|)$.

We 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. The authors 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* (or *level*) in the splayed tree and in the treap maintained by Move-to-Root. Russo's potential from [23] may be a good starting point for gleaning inspiration. But any further speculation that we provide on the form of this second type of potential is more likely than not to simply make readers liable for the authors' ignorance.

We strongly recommend those that who build on the present work exhaust all attempts at using the *first* potential function before moving on the second, for two reasons. First, it is a natural analogue of the potential function used to show that *Move-to-Front*, the algorithm that motivated *splay trees*, is constant-

[19] Sadly, we have not come up with any set of examples showing this.

[20] With apologies to Robert Frost.

competitive in list-based caching models [26]. Second, the minutia of reweighing the second kind of potential (due to level changes in subtrees of rotated nodes) is likely to considerably complicate the analysis. In a sense, the first potential function is the simplest one that conceivably can work. Time will tell if it does.

7.6 Bounding ζ by Λ'. Should either of the results conjectured in §7.3 or in §7.4 hold true, then bounding Splay's zig-zig costs by its crossing costs would suffice to affirmatively settle the conjecture. More formally

THEOREM 7.1. *If $\zeta(X,T) = O(\Lambda'(X,T) + |T|)$, and there is some constant c so that either*

- $\Lambda'(X,T) \leq c(\Lambda(X,T) + |T|)$ *or*
- $\Lambda'(Y,T) \leq c(\Lambda'(X,T) + |T|)$ *for all subsequences Y*

then Splay is dynamically optimal.

We also note that *if* our numerical experiments indicating $\Lambda'(X,T) + |T| = \Theta(\Lambda(X,T) + |T|) = \Theta(\text{cost}(X,T))$ are correct, then $\zeta(X,T) = O(\Lambda'(X,T) + |T|)$ follows by *necessity*.

We can gain some intuition on why this bound might hold by comparing the trees resulting from splaying paths with many zig-zigs to those resulting from splaying paths with many crossings. Essentially, 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 is why we suggest employing some sort of potential function to do so.

We believe an appropriate potential function will act as a proxy for "the number of zig-zigs" 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. There does seem to be some more subtlety in appropriately *defining* the "number of zig-zigs" in a binary search tree than one might expect at first. Our investigations into this matter are only preliminary, so we decline to comment further on what the potential might look like.

We postulate that $\zeta(X,T) \leq 2(\Lambda'(X,T) + |T|)$, and that the maximum value of the zig-zig potential should be $2|T|$. We surmise the factor of two from the lower bound on Splay's subsequence overhead, derived in §3.5 by splaying nodes at the end of a left path. Indeed, notice from that example that all of the $|T| - o(|T|)$ extra splay steps introduced by removing requests from X were zig-*zigs*, lending credence to our suspicion that Splay's subsequence overhead stems from its zig-zig costs.

7.7 Why This Path? Our ideas stem from two convictions: that Splay is dynamically optimal, and that the subsequence property is the way to prove it. Having covered the formal and empirical bases for these convictions, we conclude by offering a bit of intuition.

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 [26]. 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, while 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 (i.e. *crossings*, or *zig-zags*). Since the power of the BST model arises from zig-zags, it is 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 items' prior access times [32]. From Definition 2.1, when an item is accessed, a BST algorithm must bring it to the root. This requires relocating items that were brought to the root by previous requests. Wilber's bound essentially counts the number of such items which 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.

Move-to-Root is only non-optimal due to sometimes having high zig-zig costs. Splay tweaks Move-to-Root by breaking apart zig-zig edges in the side-arms of the after-tree. Splay thus seems to be precisely the modification needed to make Move-to-Root optimal.

Reducing dynamic optimality to the subsequence property 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 set 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 the subsequence property*) at other points in the space to its behavior in the subset.

We rest our case: the subsequence property is a new path from Splay to Dynamic Optimality.

8 Proof of Lemma 6.1

Let $Z = Z' \oplus (z)$. We denote by S' and T' the trees formed by executing the request sequence Z'

starting respectively from S and T using Move-to-Root. We denote by S'' and T'' the trees formed respectively by moving z to the root of T' and S'. Conceptually, this proof is a complicated induction on the number of requests in Z. Recall that we wish to show $\Lambda(Z,S) - \Lambda(Z,T) < \ell_S(x)$. The base case, when Z is the empty sequence, is trivial. For the inductive step we must show that if the lemma holds for request sequence Z' then it also holds for $Z' \oplus z$.

For $y \in Q$ we define $\Delta(y) = \ell_S(y) - \ell_T(y)$, $\Delta'(y) = \ell_{S'}(y) - \ell_{T'}(y)$, and $\Delta''(y) = \ell_{S''}(y) - \ell_{T''}(y)$. The proof will focus on parametrizing the functions Δ, Δ', and Δ'', since only the *difference* between z's crossing depth in S' and T' will contribute to any discrepancy between $\Lambda(Z,S)$ and $\Lambda(Z,T)$. Let u be the greatest key less than or equal to x in Z', and v the smallest key greater than or equal to x in Z'. (Initially $u = -\infty$ and $v = \infty$.) Observe:[21]

1. Let y be on the path from the root of a binary search tree Π to $z \in \Pi$, and let Π_y denote the subtree rooted at y in Π. The tree move-to-root(Π, z) is identical to move-to-root(Π', z), where Π' is formed by replacing Π_y with move-to-root(Π_y, z) in Π.[22]

2. The nodes in $(-\infty, u] \cup [v, \infty)$ form a connected subtree of the root in both S' and T'. This subtree, I, is identical in both S' and T'.

3. For $z \in I$, $\ell_{S'}(z) = \ell_{T'}(z)$.

4. For $y \in Q \setminus I$, both u and v appear on the search path for y in both S' and T'. If u was accessed more recently than v in Z' then the set of keys in v's left subtree is exactly $Q \setminus I$, otherwise the set of keys in u's right subtree is exactly $Q \setminus I$.[23]

5. Let J be subtree containing keys $Q \setminus I$ in S', and K the subtree with keys $Q \setminus I$ in T'. $K =$ move-to-root(J, x).

6. The crossing depth of a node y in tree Π can be determined as follows. If y is the root, then $\ell_\Pi(y) = 1$. If y's parent, p is the root, then $\ell_\Pi(y) = 2$. If y and p are both left children or both right children then $\ell_\Pi(y) = \ell_\Pi(p)$. Otherwise $\ell_\Pi(y) = \ell_\Pi(p) + 1$.

7. Let J_+ and K_+ be subtrees J and K, augmented with their parent in I. From Observation 6, $\Delta'(y) = \ell_{J_+}(y) - \ell_{K_+}(y)$ for $y \in Q \setminus I$.

[21] Proofs omitted for brevity.

[22] Splay does not share this "overlapping structure" property due to the zig-zig steps.

[23] We omit details for cases when $v - u = \infty$.

Observation 3 ensures that $\Delta'(z) = 0$ for $z \in I$. Observation 7 suggests computing $\Delta'(z)$ for $z \in Q \setminus I$ in terms of $\ell_{J_+}(z) - \ell_{K_+}(z)$. Both ℓ_{J_+} and ℓ_{K_+} can be determined entirely from knowing u, v, the original configurations of S and T, and which of u or v was more recently accessed in Z', as follows.

Without loss of generality, assume that the root of S is less than x and that x is a left child in S (the other cases are similar). Define the indicator variable δ to be 1 if x is a right child in T' and 0 otherwise. Similarly define φ to be 1 if x is a left child in T' and 0 otherwise. The quantity $\delta + \varphi$ is an indicator of whether x has a parent in T'. Define the indicator variable ψ to be 1 if the root of J_+ is greater than x, and 0 otherwise. Similarly define σ to be 1 if the root of J_+ is less than x, and 0 otherwise. Finally, let j be the number of crossing nodes for x in J with keys greater than x. Note x has $j - \psi$ crossing nodes in J with keys less than x.

Let P be the nodes of the path from the root of J to x, *excluding* x, descending downwards. Let L be the nodes of the rightward path descending downward from the root of x's left subtree, R the nodes on the leftward path descending from the root of x's right subtree, and define $P_+ = P \cup L \cup R \cup \{x\}$.

Let $a_1, \ldots, a_{j-\psi}$ be the crossing nodes encountered on the left side of x descending down the path from the root of J to x (if $j - \psi = 0$ then none are defined). Similarly, let $b_1, \ldots, b_j$ be the crossing nodes so encountered on the right side of x. Additionally, define $a_{j+1-\psi}$ to be the root of x's left subtree in J, and b_{j+1} the root of its right subtree. For convenience, define $a_{j+2-\psi} = b_{j+2} = x$.

For $1 \le i \le j+1-\psi$, define c_i to be the smallest node in P_+ lying strictly between a_i and a_{i+1} in symmetric order, and for $1 \le i \le j+1$ let d_i be the largest node in P_+ lying strictly between b_i and b_{i+1} in symmetric order. Finally, let A_i and C_i be the left subtrees of a_i and c_i, and B_i and D_i the right subtrees of b_i and d_i. We make the following further observations:

8. For $1 \le i \le j$ and any node $y \in P_+$ lying between b_i and b_{i+1} in symmetric order, $\ell_{J_+}(y) = \ell_{J_+}(d_i)$. A similar statement holds for $\ell_{J_+}(c_i)$. This justifies the use of c_i and d_i as stand-ins for all nodes in P_+ in respective intervals (a_i, a_{i+1}) and (b_i, b_{i+1}).

9. For $y \in P_+ \setminus \{x\}$, let Σ_y denote its left subtree in J if $y < x$ and its right subtree in J if $y > x$. Let Σ'_y be defined similarly in K. Then $\Sigma_y = \Sigma'_y$.

10. Denote by $\ell_\Pi(\Sigma)$ the level of the *root* of subtree Σ in Π. For any node $y \in A_i$, $\Delta'(y) = \Delta'(A_i)$, and for any node y in the left subtree of node $f \in P_+$ such that $a_i < f < a_{i+1}$, $\Delta'(y) = \Delta'(C_i)$. A similar statement holds for B_i and D_i.

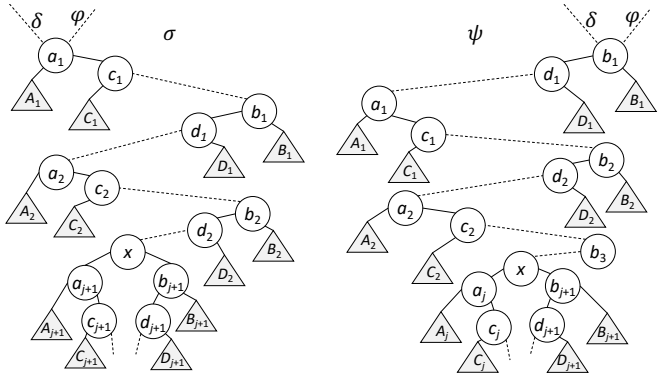

Figure 11: Schematic of tree J_+.

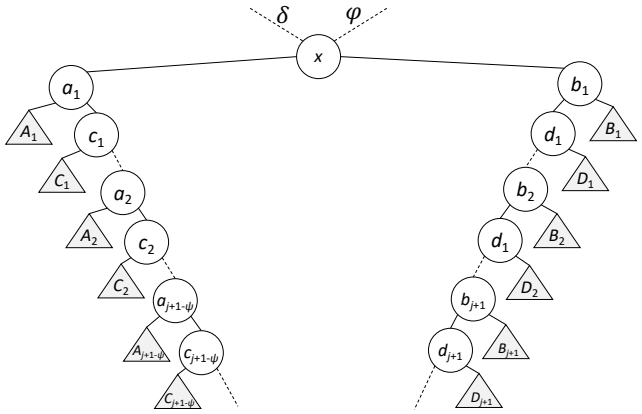

Figure 12: Schematic of tree K_+.

11. When $j > 1$, he following hold (see Figure 11):

$$\ell_{J_+}(x) = 2j + \varphi\sigma + \sigma + \delta\psi$$

$$\ell_{J_+}(a_i) = \begin{cases} 1 + \delta + \varphi + \delta\psi, & i = 1 \\ 2i - 1 + \varphi\sigma + \psi + \delta\psi, & 1 < i \le j - \psi \\ \ell_{J_+}(x), & i = j + 1 - \psi \end{cases}$$

$$\ell_{J_+}(b_i) = \begin{cases} 2 + \varphi\sigma, & i = 1 \\ 2i - 1 + \varphi\sigma + \sigma + \delta\psi, & 1 < i \le j \\ \ell_{J_+}(x) + 1, & i = j + 1 \end{cases}$$

$$\ell_{J_+}(c_i) = \ell_{J_+}(b_{i+\psi}), \quad 1 \le i \le j + 1 - \psi$$

$$\ell_{J_+}(d_i) = \begin{cases} \ell_{J_+}(a_{i+\sigma}), & 1 \le i \le j \\ \ell_{J_+}(x) + 2, & i = j + 1 \end{cases}$$

$$\ell_{J_+}(A_i) = \begin{cases} 2 + \delta, & i = 1 \\ \ell_{J_+}(a_i), & 1 < i \le j + 1 - \psi \end{cases}$$

$$\ell_{J_+}(B_i) = \begin{cases} 2 + \varphi, & i = 1 \\ \ell_{J_+}(b_i), & 1 < i \le j + 1 \end{cases}$$

$$\ell_{J_+}(C_i) = \ell_{J_+}(c_i) + 1, \quad 1 \le i \le j + 1 - \psi$$

$$\ell_{J_+}(D_i) = \ell_{J_+}(d_i) + 1, \quad 1 \le i \le j + 1$$

12. The following can be extrapolated from Figure 12:

$$\ell_{K_+}(x) = 1 + \delta + \varphi$$

$$\ell_{K_+}(a_i) = \begin{cases} 2 + \delta, & i = 1 \\ 3 + \delta, & 1 < i \le j + 1 - \psi \end{cases}$$

$$\ell_{K_+}(b_i) = \begin{cases} 2 + \varphi, & i = 1 \\ 3 + \varphi, & 1 < i \le j + 1 \end{cases}$$

$$\ell_{K_+}(c_i) = 3 + \delta, \quad 1 \le i \le j + 1 - \psi$$

$$\ell_{K_+}(d_i) = 3 + \varphi, \quad 1 \le i \le j + 1$$

$$\ell_{K_+}(A_i) = \begin{cases} 2 + \delta, & i = 1 \\ 4 + \delta, & 1 < i \le j + 1 - \psi \end{cases}$$

$$\ell_{K_+}(B_i) = \begin{cases} 2 + \varphi, & i = 1 \\ 4 + \varphi, & 1 < i \le j + 1 \end{cases}$$

$$\ell_{K_+}(C_i) = 4 + \delta, \quad 1 \le i \le j + 1 - \psi$$

$$\ell_{K_+}(D_i) = 4 + \varphi, \quad 1 \le i \le j + 1$$

The value $\Delta'(z)$ can now be computed from $\ell_{J_+}(z) - \ell_{K_+}(z)$. Using this, we note that for $z \in Q$:

(a) $\Delta(x) = \ell_S(x) - 1$,

(b) $\Delta'(x) \ge \max\{\Delta'(z), 0\}$,

(c) $\Delta''(x) \le \Delta'(x) - \max\{\Delta'(z), 0\}$.

Items (a) and (b) are found by maximizing Δ' for each of the sixteen families of nodes discussed in Observations 11 and 12. Inequality (c) follows by a more tedious analysis, using previous observations, of how moving z with level difference $\Delta'(z)$ to the root affects the parameters δ, φ, σ, ψ and j, and thereby $\Delta''(x)$. We omit these analyses.

We are now ready to complete the proof of the lemma. Recall the original goal: we wish to show that $\Lambda(Z, S) - \Lambda(Z, T) < \ell_S(x)$. Let $Z = (z_1, \ldots, z_m)$. Let $T_0 = T$, $S_0 = S$, and for $1 \le i < m$ let $T_i = \text{move-to-root}(T_{i-1}, z_i)$ and $S_i = \text{move-to-root}(S_{i-1}, z_i)$. Let $\Delta^i(y) = \ell_{S_{i-1}}(y) - \ell_{T_{i-1}}(y)$. Note that $\Delta^1 = \Delta$.

By definition, $\Lambda(Z, S) - \Lambda(Z, T) = \sum_{i=1}^{m} \Delta^i(z_i)$. By Observation (b), $\sum_{i=1}^{m} \Delta^i(z_i) \le \Delta^m(x) + \sum_{i=1}^{m-1} \Delta^i(z_i)$.[24] By Observation (c), $\Delta^i(x) \le \Delta^{i-1}(x) - \max\{\Delta^{i-1}(z_{i-1}), 0\}$, hence for $1 \le k \le m$,

$$\Delta^i(x_i) + \sum_{i=1}^{k-1} \Delta^i(z_i) \le \Delta^{i-1}(x) + \sum_{i=1}^{k-2} \Delta^i(z_i).$$

When $k = 1$, we have $\sum_{i=1}^{m} \Delta^i(z_i) \le \Delta(x) < \ell_S(x)$. The last inequality follows from Observation (a). □

[24] If $k < j$ then we define $\sum_{i=j}^{k}$ to be zero.

9 Acknowledgements

We thank Luís Russo for suggestions that much improved Figure 6, Kurt Mehlhorn for supplying alternative proofs to Theorems 4.1 and 6.2, Amit Halevi for providing comments that greatly clarified Definition 2.1, and Siddhartha Sen for editorial feedback. We are indebted to John Iacono for his guidance in understanding the equivalence between Wilber's bound and the crossing nodes of treaps, along with corroborating our empirical comparisons between the behaviors of Splay and Wilber's bound. We are also grateful to David Galles for making his "Data Structure Visualizations" available on his website.[25] The first author in particular spent many hours prototyping the ideas featured in this paper using Galles' visualizer. This would have been a lesser work without this publicly available tool.

Final Remarks

The dynamic optimality conjecture remains standing despite our best efforts, and the path forward does not appear easy. Our hope is that we have at least brought this problem from the realm of the unyielding to the realm of what is merely difficult. We anticipate that, in the long run, the most important influence of this work will be the application of simulation embeddings for analyzing approximation algorithms generally. For now, the fate of this fascinating tale lies in the hands of our audience.

References

[1] B. Allen and I. Munro, *Self-Organizing Binary Search Trees*, J. ACM, 25, (1978), pp. 526–535.

[2] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak, *Pattern-Avoiding Access in Binary Search Trees*, FOCS, (2015), pp. 410–423.

[3] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak, *Self-Adjusting Binary Search Trees: What Makes Them Tick?*, ESA, (2015), pp. 300–312.

[4] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak, *The Landscape of Bounds for Binary Search Trees*, ArXiv, (2016), pp. 1–32.

[5] S. Cleary, *Restricted Rotation Distance between Binary Trees*, Inform. Process. Lett., 84(6), (2002) pp. 333–338.

[6] S. Cleary and J. Taback, *Bounding Restricted Rotation Distance*, Inform. Process. Lett., 88(5), (2003), pp. 251–256.

[7] R. Cole, *On the Dynamic Finger Conjecture for Splay Trees*, STOC, (1990), pp. 8–17.

[8] R. Cole, B. Mishra, J. Schmidt, and A. Siegel, *On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting* $\log n$*-Block Sequences*, SICOMP, 30(1), (2000), pp. 1–43.

[9] R. Cole, *On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof*, SCICOMP, 30(1), (2000), pp. 44–85.

[10] E. Demaine, D. Harmon, J. Iacono, D. Kane, and M. Pătraşcu, *The Geometry of Binary Search Trees*, SODA, (2009), pp. 496–505.

[11] E. Demaine, D. Harmon, J. Iacono, and M. Pătraşcu, *Dynamic Optimality – Almost*, SICOMP, 37, (2007), pp. 240–251.

[12] G. Georgakopoulos and D. McClurkin, *Generalized Template Splay: A Basic Theory and Calculus*, The Computer Journal, (47)1, (2004), pp. 10–19.

[13] D. Harmon, *New Bounds on Optimal Binary Search Trees*, MIT, 2006.

[14] J. Howat, J. Iacono, and P. Morin, *The Fresh-Finger Property*, arXiv, (2013), pp. 1–11.

[15] J. Iacono, *Key-Independent Optimality*, Algorithmica, 42(1), (2005), pp. 3–10.

[16] J. Iacono, *In Pursuit of the Dynamic Optimality Conjecture*, LNCS, (2013), pp. 236–250.

[17] J. Iacono and M. Yagnatinsky, *A Linear Potential Function for Pairing Heaps*, COCOA, (2016), pp. 489–504.

[18] L. Kozma, *Binary Search Trees, Rectangles and Patterns*, Saarland University, 2016.

[19] J. Lucas, *A Direct Algorithm for Restricted Rotation Distance*, Inform. Process. Lett., 90(3), (2004), pp. 129–134.

[20] J. Lucas, *Canonical Forms for Competitive Binary Search Tree Algorithms*, Rutgers University Technical Report, (1988).

[21] J. Lucas, *The Rotation Graph of Binary Trees Is Hamiltonian*, J. Algorithms, 8(4), (1987), pp. 503–535.

[22] J. Lucas, D. Roelants van Baronaigien, and F. Ruskey, *On Rotations and the Generation of Binary Trees*, J. Algorithms, 15(3), (1993), pp. 343–366.

[23] L. Russo, *A Study on Splay Trees*, arXiv, (2015).

[24] R. Seidel and C. Aragon, *Randomized Search Trees*, Algorithmica, (16), (1996), pp. 464–497.

[25] S. Sleator and R. Tarjan, *Self-Adjusting Binary Trees*, STOC, 15, (1983), pp. 235–245.

[26] S. Sleator and R. Tarjan, *Amortized Efficiency of List Update and Paging Rules*, Comm. ACM, 28(2), (1985), pp. 202–208.

[27] S. Sleator and R. Tarjan, *Self-Adjusting Binary Search Trees*, J. ACM, 32, (1985), pp. 652–686.

[28] A. Subramanian, *An Explanation of Splaying*, J. Algorithms, 20(3), (1996), pp. 512–525.

[29] R. Tarjan, *Amortized Computational Complexity*, SIAM J. Alg. Disc. Math., 6(2), (1985), pp. 306–318.

[30] R. Tarjan, *Sequential Access in Splay Trees Takes Linear Time*, Combinatorica, 5(4), (1985), pp. 367–378.

[31] J. Vuillemin, *A Unifying Look at Data Structures*, Commun. ACM, (23), (1980), pp. 652–686.

[32] R. Wilber, *Lower Bounds for Accessing Binary Search Trees with Rotations*, SICOMP, 18, (1989), pp. 56–67.

[25] https://www.cs.usfca.edu/~galles/visualization/Algorithms.html