Dynamic Optimality—Almost*

Erik D. Demaine†‡ Dion Harmon* John Iacono§† Mihai Pătraşcu*

Abstract

We present an $O(\lg \lg n)$-competitive online binary search tree, improving upon the best previous (trivial) competitive ratio of $O(\lg n)$. This is the first major progress on Sleator and Tarjan's dynamic optimality conjecture of 1985 that $O(1)$-competitive binary search trees exist.

1 Introduction

Binary search trees (BSTs) are one of the most fundamental data structures in computer science. Despite decades of research, the most fundamental question about BSTs remains unsolved: what is the asymptotically best BST data structure? This problem is unsolved even if we focus on the case where the BST stores a static set and does not allow insertions and deletions.

1.1 Model

To make precise the notion of "asymptotically best BST", we now define the standard notions of BST data structures and dynamic optimality. Our definition is based on the one by Wilber [Wil89], which also matches the one used implicitly by Sleator and Tarjan [ST85].

BST data structures. We consider BST data structures supporting only searches on a static universe of keys $\{1, 2, \ldots, n\}$. We consider only successful searches, which we call *accesses*. The input to the data structure is thus a sequence X, called the *access sequence*, of keys $x_1, x_2, \ldots, x_m$ chosen from the universe.

A BST data structure is defined by an algorithm for serving a given access x_i, called the *BST access algorithm*. The BST access algorithm has a single pointer to a node in the BST. At the beginning of an access to a given key x_i, this pointer is initialized to the root of the tree. The algorithm may then perform any sequence of the following unit-cost operations such that the node containing x_i is eventually the target of the pointer.

1. Move the pointer to its left child.
2. Move the pointer to its right child.
3. Move the pointer to its parent.
4. Perform a single rotation on the pointer and its parent.

Whenever the pointer moves to or is initialized to a node, we say that the node is *touched*. The time taken by a BST to execute a sequence X of accesses to keys $x_1, x_2, \ldots, x_m$ is the number of unit-cost operations performed, which is at least the number of nodes it touches. There are several possible variants of this definition that can be shown to be equivalent up to constant factors. For example, in one such variant, the pointer begins a new operation where it finished the previous operation, rather than at the root [Wil89].

*A preliminary version of this paper appeared in FOCS 2004 [DHIP04].
†MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, {edemaine,dion, mip}@mit.edu
‡Supported in part by NSF grant CCF-0430849.
§Department of Computer and Information Science, Polytechnic University, 5 MetroTech Center, Brooklyn, NY 11201, USA, jiacono@poly.edu

An *online BST data structure* augments each node in a BST with additional data. Every unit-cost operation can change the data in the new node pointed to by the pointer. The access algorithm's choice of the next operation to perform is a function of the data and augmented data stored in the node currently pointed to. In particular, the algorithm's behavior depends only on the past. The amount of augmented information at each node should be as small as possible. For example, red-black trees use one bit [CLRS01, chapter 13] and splay trees do not use any [ST85]. Any online BST that uses only $O(1)$ augmented words per node has a running time in the RAM model dominated by the number of unit-cost operations in the BST model.

Optimality. Given any particular access sequence X, there is some BST data structure that executes it optimally. Let $\mathrm{OPT}(X)$ denote the number of unit-cost operations made by this fastest BST data structure for X. In other words, $\mathrm{OPT}(X)$ is the fastest any *offline* BST can execute X, because the model does not restrict how a BST access algorithm chooses its next move, so in particular it may depend on the future accesses to come. Standard balanced BSTs establish that $\mathrm{OPT}(X) = O(m \lg n)$. Wilber [Wil89] proved that $\mathrm{OPT}(X) = \Theta(m \lg n)$ for some classes of sequences X.

A BST data structure is *dynamically optimal* if it executes all sequences X in time $O(\mathrm{OPT}(X))$. It is not known whether such a data structure exists. More generally, a BST data structure is *c-competitive* if it executes all sufficiently long sequences X in time at most $c\mathrm{OPT}(X)$.

The goal of this line of research is to design a dynamically optimal ($O(1)$-competitive) online BST data structure that uses $O(1)$ augmented bits per node. The result would be a single, asymptotically best BST data structure.

1.2 Previous Work

Much of the previous work on the theory of BSTs centers around splay trees of Sleator and Tarjan [ST85]. Splay trees are an online BST data structure that use a simple restructuring heuristic to move the accessed node the root. Splay trees are conjectured in [ST85] to be dynamically optimal. This conjecture remains unresolved.

Upper bounds. Several upper bounds have been proved on the performance of splay trees: the working-set bound [ST85], the static finger bound [ST85], the sequential access bound [Tar85], and the dynamic finger bound [CMSS00, Col00]. These bounds show that splay trees execute certain classes of access sequences in $o(m \lg n)$ time, but they all provide $O(m \lg n)$ upper bounds on access sequences that actually take time $\Theta(m)$ time to execute on splay trees. There are no known upper bounds on any BST that are superior to these splay tree bounds. Thus, no BST is known to be better than $O(\lg n)$-competitive against the offline optimal BST data structure.

There are several related results in different models. The unified structure [Iac01, BD04] has an upper bound on its runtime that is stronger than all of the proved upper bounds on splay trees. However, this structure is not a BST data structure, augmenting with additional pointers, and it too is no better than $O(\lg n)$-competitive against the offline optimal BST data structure.

Lower bounds. There are two known lower bounds for the BST model, both due to Wilber [Wil89]. Given an access sequence X, they provide lower bounds on the cost of any BST data structure to execute X. Neither bound is simply stated; they are both complex functions of X. We use a variation on the first bound extensively in this paper, and describe it in detail in Section 2.

Optimality. Several restricted optimality results have been proved for BSTs.

The first result is the "optimal BST" of Knuth [Knu71]. Given an access sequence X over the universe $\{1, 2, \ldots, n\}$, let f_i be the number of accesses in X to key i. Optimal BSTs execute X in the entropy bound $O(\sum_{i=1}^{n} f_i \lg(m/f_i))$. This bound is expected to be $O(\mathrm{OPT}(X))$ if the accesses are drawn independently at random from a fixed distribution matching the frequencies f_i. The bound is not optimal if the accesses are dependent or not random. Originally, these trees required the f values for construction, but this requirement is lifted by splay trees, which share the asymptotic runtime of the older optimal trees.

The second result is key-independent optimality [Iac02]. Suppose $Y = \langle y_1, y_2, \ldots y_m \rangle$ is a sequence of accesses to a set S of n unordered items. Let b be a uniform random bijection from S to $\{1, 2, \ldots, n\}$. Let $X = \langle f(x_1), f(x_2), \ldots, f(x_m) \rangle$. The key-independent optimality result proves that splay trees, and any data structure with the working-set bound, executes X in time $O(E[\mathrm{OPT}(X)])$. In other words, if key values are assigned arbitrarily (but consistently) to unordered data, splay trees are dynamically optimal. This result uses the second lower bound of Wilber [Wil89].

The third result [BCK02] shows that there is an online BST data structure whose search cost is $O(\mathrm{OPT}(X))$ given *free* rotations between accesses. This data structure is heavily augmented and uses exponential time to decide what BST operations to perform next.

1.3 Our Results

In summary, splay trees are conjectured to be $O(1)$-competitive for all access sequences, but no online BST data structure is known to have a competitive factor better than the trivial $O(\lg n)$, no matter how much time or augmentation they use to decide the next BST operation to perform. In fact, no polynomial-time offline BST is known to exist either. (Offline and with exponential time, one can of course design a dynamically optimal structure by simulating all possible offline BST structures that run in time at most $2m \lg n$ to determine the best one, before executing a single BST operation.)

We present an online BST data structure, called Tango, that is $O(\lg \lg n)$-competitive against the optimal offline BST data structure on every access sequence. Tango uses $O(\lg \lg n)$ bits (less than 1 word) of augmentation per node, and the bookkeeping cost to determine the next BST operation is constant amortized.

1.4 Overview

Our results are based on a slight variation of the first lower bound of Wilber [Wil89], called the *interleave lower bound*. This lower bound considers applying the access sequence to a fixed lower-bound tree which, in our case, is a perfect binary tree P. The bound counts the number of "interleaves", i.e., switches from accessing the left subtree of a node to accessing the right subtree of a node, or vice versa. Section 2 defines the lower bound precisely, and Appendix A gives a proof. Our results show that this lower bound is within an $O(\lg \lg n)$ factor of being tight against the offline optimal; we also show in Section 3.5 that the lower bound is no tighter in the worst case.

Tango simulates the behavior of the lower-bound tree P using a tree of balanced BSTs (but represented as a single BST). Specifically, before any access, we can conceptually decompose P into *preferred paths* where, at each node, a path proceeds to the child subtree that was most recently accessed. Other than a startup cost of $O(n)$, the interleave lower bound is the sum, for each access, of the number of edges that connect different preferred paths. The Tango BST matches this bound up to an $O(\lg \lg n)$ factor by representing each preferred path as a balanced BST, called an *auxiliary tree*. Because the perfect tree P has height $O(\lg n)$, each auxiliary tree contains $O(\lg n)$ nodes, so it costs $O(\lg \lg n)$ to visit each preferred path. The technical details are in showing how to maintain the auxiliary trees as the preferred paths change, conceptually using $O(1)$ split and concatenate operations per change, while maintaining the invariant that at all times the entire structure is a single BST sorted by key. Overall, the Tango BST runs any access sequence X in $O((\mathrm{OPT}(X) + n)\,(1 + \lg \lg n))$ time, which is $O(\mathrm{OPT}(X)\,(1 + \lg \lg n))$ provided $m = \Omega(n)$. Section 3 gives a detailed description and analysis of Tango.

The Tango BST is similar to link-cut trees of Sleator and Tarjan [ST83, Tar83]. Both data structures maintain a tree of auxiliary trees, where each auxiliary tree represents a path of a represented tree (in Tango, the lower-bound tree P). Some versions of link-cut trees also define the partition into paths in the same way as preferred paths, except that the represented tree is dynamic. The main distinction is that Tango is a BST data structure, so the "tree of auxiliary trees" must be stored as a single BST sorted by key value. In constrast, the auxiliary trees in link-cut trees are sorted by depth, making it easier to splice preferred paths as necessary. We show that this difficulty is surmountable with only a little extra bookkeeping.

1.5 Further Work

Since the conference version of this paper [DHIP04], Wang, Derryberry, and Sleator [WDS06] have developed a variant of Tango trees, called the *multi-splay tree*, in which auxiliary trees are splay trees. (Interestingly, this idea is also used in one version of link-cut trees [Tar83].) In this variant, they establish $O(\lg \lg n)$-competitiveness, an $O(\lg n)$ amortized time bound per operation, and an $O(\lg^2 n)$ worst-case time bound per operation. Furthermore, they generaralize the lower-bound framework and data structure to handle insertions and deletions on the universe of keys.

We note that a more complicated but easier-to-analyze variant of Tango executes any access sequence X in $O\big((\mathrm{OPT}(X) + n)\,(1 + \lg \frac{\lg n}{\mathrm{OPT}(X)/n})\big)$ time, which is always $O(n \lg n)$. Namely, we can replace the auxiliary tree data structure with a balanced BST supporting search, split, and concatenate operations in the dynamic finger bound, $O(1 + \lg r)$ time where r is 1 plus the rank difference between the accessed element and the previously accessed element. For example, one such data structure maintains the previously accessed element at the root, and has roughly exponentially increasing subtrees on either side hanging off the left and right spines. Following the analysis in Section 3.4, the cost of accessing an element in the tree of auxiliary trees then becomes $O\big(\sum_{i=1}^{k}(1 + \lg r_i)\big)$, where $r_1, r_2, \ldots, r_k$ are the numbers of nodes along the k preferred paths we visit, so $r_1 + r_2 + \cdots + r_k = \Theta(\lg n)$. This cost is maximized up to constant factors when $r_1 = r_2 = \cdots = r_k = \Theta\big(\frac{\lg n}{k}\big)$, for a cost of $O\big(k(1 + \lg \frac{\lg n}{k})\big)$. Summing over all accesses, the k's sum to the interleave lower bound plus an $O(n)$ startup cost, so the total cost is maximized up to constant factors when each access cost is approximately equal, for a total cost of $O\big((\mathrm{OPT}(X) + n)\,(1 + \lg \frac{\lg n}{\mathrm{OPT}(X)/n})\big)$. Indeed, this variant guarantees $O(\lg n)$ worst-case time per access.

2 Interleave Lower Bound

The *interleave bound* is a lower bound on the time taken by any BST data structure to execute an access sequence X, dependent only on X. The particular version of the bound that we use is a slight variation of the first bound of Wilber [Wil89]. (Specifically, our lower-bound tree P has a key at every node, instead of just at the leaves; we also fix P to be the complete tree for the purposes of upper bounds; and our bound differs by roughly a factor of two because we do not insist that the search algorithm brings the desired element to the root.) Our lower bound is also similar to lower bounds that follow from partial sums in the semigroup model [HF98, PD04]. Given the similarity to previous lower bounds, we just state the bound in this section, and delay the proof to Appendix A.

We maintain a perfect binary tree, called the *lower-bound tree* P, on the keys $\{1, 2, \ldots, n\}$. (If n is not one less than a power of two, the tree is complete, not perfect.) This tree has a fixed structure over time.

For each node y in P, define the *left region* of y to consist of y itself plus all nodes in y's left subtree in P; and define the *right region* of y to consist of all nodes in y's right subtree in P. The left and right regions of y partition y's subtree in P and are temporally invariant. For each node y in P, we label each access x_i in the access sequence X by whether x_i is in the left or right region of y, discarding all accesses outside y's subtree in P. The *amount of interleaving through* y is the number of alternations between "left" and "right" labels in this sequence. The *interleave bound* $\mathrm{IB}(X)$ is the sum of these interleaving amounts over all nodes y in P.

The exact statement of the lower bound is as follows:

Theorem 2.1 $\mathrm{IB}(X)/2 - n$ *is a lower bound on* $\mathrm{OPT}(X)$*, the cost of the optimal offline BST that serves access sequence* X*.*

3 BST Upper Bound

3.1 Overview of the Tango BST

We now define a specific BST access algorithm, called *Tango*. Let T_i denote the state of the Tango BST after executing the first i accesses $x_1, x_2, \ldots, x_i$. We define T_i in terms of an *augmented* lower-bound tree P.

As in the interleave lower bound, P is a perfect binary tree on the same set of keys, $\{1, 2, \ldots, n\}$. We augment P to maintain, for each internal node y of P, a *preferred child* of either the left or the right, specifying whether the last access to a node within y's subtree in P was in the left or right region of y. In particular, because y is in its left region, an access to y sets the preferred child of y to the left. If no node within y's subtree in P has yet been accessed (or if y is a leaf), y has no preferred child. The state P_i of this augmented perfect binary tree after executing the first i accesses $x_1, x_2, \ldots, x_i$ is determined solely by the access sequence, independent of the Tango BST.

The following transformation converts a state P_i of P into a state T_i of the Tango BST. Start at the root of P and repeatedly proceed to the preferred child of the current node until reaching a node without a preferred child (e.g., a leaf). The nodes traversed by this process, including the root, form a *preferred path*. We compress this preferred path into an "auxiliary tree" R. (Auxiliary trees are BSTs defined below.) Removing this preferred path from P splits P into several pieces; we recurse on each piece and hang the resulting BSTs as children subtrees of the auxiliary tree R.

The behavior of the Tango BST is now determined: at each access x_i, the state T_i of the Tango BST is given by the transformation described above applied to P_i. We have not yet defined how to efficiently obtain T_i from T_{i-1}. To address this algorithmic issue, we first describe auxiliary trees and the operations they support.

3.2 Auxiliary Tree

The *auxiliary tree* data structure is an augmented BST that stores a subpath of a root-to-leaf path in P (in our case, a preferred path), but ordered by key value. With each node we also store its fixed *depth* in P. Thus, the depths of the nodes in an auxiliary tree form a subinterval of $[0, \lg(n+1))$. We call the shallowest node the *top* of the path, and the deepest node the *bottom* of the path. We require the following operations of auxiliary trees:

1. *Searching* for an element by key in an auxiliary tree.
2. *Cutting* an auxiliary tree into two auxiliary trees, one storing the path of all nodes of depth at most a specified depth d, and the other storing the path of all nodes of depth greater than d.
3. *Joining* two auxiliary trees that store two disjoint paths where the bottom of one path is the parent of the top of the other path.

We require that all of these operations take time $O(\lg k)$ where k is the total number of nodes in the auxiliary tree(s) involved in the operation. Note that the requirements of auxiliary trees (and indeed their solution) are similar to Sleator and Tarjan's link-cut trees [ST83, Tar83]; however, auxiliary trees have the additional property that the nodes are stored in a BST ordered by key value, not by depth in the path.

An auxiliary tree is implemented as an augmented red-black tree. In addition to storing the key value and depth, each node stores the minimum and maximum depth over the nodes in its subtree. This auxiliary data can be trivially maintained in red-black trees with a constant-factor overhead; see, e.g., [CLRS01, chapter 14].

The additional complication is that the nodes which would normally lack a child in the red-black tree (e.g., the leaves) can nonetheless have child pointers which point to other auxiliary trees. In order to distinguish auxiliary trees within this tree-of-auxiliary-trees decomposition, we mark the root of each auxiliary tree.

Recall that red-black trees support search, split, and concatenate in $O(\lg k)$ time [CLRS01, Problem 13-2]. In particular, this allows us to search in an augmented tree in $O(\lg k)$ time. We use the following specific forms of split and concatenate phrased in terms of a tree-of-trees representation instead of a forest representation:

1. *Split* a red-black tree at a node x: Re-arrange the tree so that x is at the root, the left subtree of x is a red-black tree on the nodes with keys less than x, and the right subtree of x is a red-black tree on the nodes with keys greater than x.
2. *Concatenate* two red-black trees whose roots are children of a common node x: Re-arrange x's subtree to form a red-black tree on x and the nodes in its subtree.

Both operations do not descend into marked nodes, where other auxiliary trees begin, treating them as external nodes (i.e., ignoring the existence of marked subtrees but preserving the pointers to them automatically during rotations). It is easy to phrase existing split and concatenate algorithms in this framework.

Now we describe how to support cut and join using split and concatenate.

To cut an augmented tree A at depth d, first observe that the nodes of depth greater than d form an interval of key space within A. Using the augmented maximum depth of each subtree, we can find the node ℓ of minimum key value that has depth greater than d in $O(\lg k)$ time, by starting at the root and repeatedly walking to the leftmost child whose subtree has maximum depth greater than d. Symmetrically, we can find the node r of maximum key value that has depth greater than d. We also compute the predecessor ℓ' of ℓ and the successor r' of r.

With the interval $[\ell, r]$, or equivalently the open interval (ℓ', r'), defining the range of interest, we manipulate the trees using split and concatenate as shown in Figure 1. First we split A at ℓ' to form two subtrees

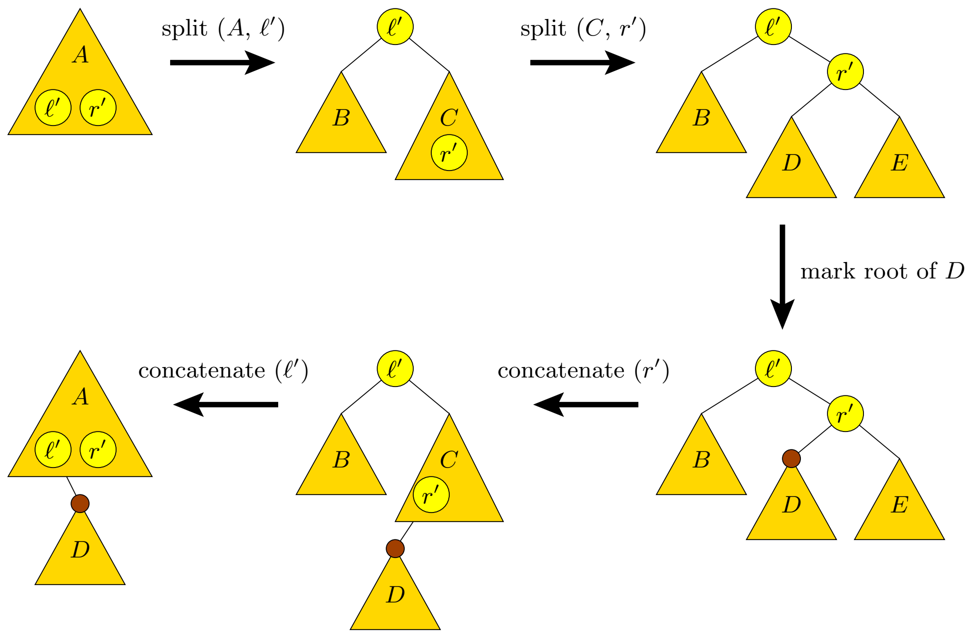

Figure 1: Implementing cut with split, mark, and concatenate.

B and C of ℓ' corresponding to key ranges $(-\infty, \ell')$ and (ℓ', ∞). (We skip this step, and the subsequent concatenate at ℓ', if $\ell' = -\infty$.) Then we split C at r' to form two subtrees D and E of r' corresponding to key ranges (ℓ', r') and (r', ∞). (We skip this step, and the subsequent concatenate at r', if $r' = \infty$.) Now we mark the root of D, effectively splitting D off from the remaining tree. The elements in D have keys in the range (ℓ', r'), which is equivalent to the range $[\ell, r]$, which are precisely the nodes of depth greater than d. Next we concatenate at r', which to the red-black tree appears to have no left child; thus the concatenation simply forms a red-black tree on r' and the nodes in its right subtree. Finally we concatenate at ℓ', effectively merging all nodes except those in D. The resulting tree therefore has all nodes of depth at most d.

Joining two augmented trees A and B is similar, except that we unmark instead of mark. First we determine which tree stores nodes of depth larger than all nodes in the other tree by comparing the depths of the roots of A and B. Suppose by relabeling that A stores nodes of larger depth. Symmetric to cuts, observe that the nodes in B have key values that fall in between two adjacent keys ℓ' and r' in A. We can find these keys by searching in A for the key of B's root. Indeed, if we split A at ℓ' and then r' (skipping a split and the subsequent concatenate in the case of $\pm\infty$), the marked root of B becomes the left child of r'. Then we unmark the root of B, concatenate at r', and then concatenate at ℓ'. The result is a single tree containing all elements from A and B.

3.3 Tango Algorithm

Now we describe how to construct the new state T_i of the BST given the previous state T_{i-1} and the next access x_i. The access algorithm follows a normal BST walk in T_{i-1} toward the query key x_i. Accessing x_i changes the necessary preferred children to make a preferred path from the root to x_i, sets the preferred child of x_i to the left, and does not change any other preferred children. Except for the last change to x_i's preferred child, the points of change in preferred children correspond exactly to where the BST walk in T_{i-1} crosses from one augmented tree to the next, i.e., where the walk visits a marked node. Thus, when the walk visits a marked node x, we cut the auxiliary tree containing the parent of x, cutting at depth 1 less than the minimum depth of nodes in the auxiliary tree rooted at x; and then we join the resulting top path with the augmented tree rooted at x. Finally, when we reach x_i, we cut its auxiliary tree at the depth of x_i, and join the resulting top path with the auxiliary tree rooted at the predecessor marked node of x_i.

3.4 Analysis

Lemma 3.1 *The running time of an access x_i is $O((k+1)\,(1+\lg\lg n))$, where k is the number of nodes whose preferred child changes during access x_i.*

Proof: The running time consists of two parts: the cost of searching for x_i and the cost of re-arranging the structure from state T_{i-1} into state T_i.

The search visits a root-to-x_i path in T_{i-1}, which we partition into subpaths according to the auxiliary trees visited. The transition between two auxiliary trees corresponds one-to-one to the edge between a node and its nonpreferred child in the root-to-x_i path in P, at which a node's preferred child changes because of this access. Thus the search path in T_{i-1} partitions into at most $k+1$ subpaths in $k+1$ auxiliary trees. The cost of the search within a single auxiliary tree is $O(\lg\lg n)$ because each auxiliary tree stores $O(\lg n)$ elements, corresponding to a subpath of a root-to-leaf path in P. Therefore the total search cost for x_i is $O((k+1)\,(1+\lg\lg n))$.

The update cost is the same as the search cost up to constant factors. For each of the at most $k+1$ auxiliary trees visited by the search, we perform one cut and one join, each costing $O(\lg\lg n)$. We also pay $O(\lg\lg n)$ to find the predecessor marked node of x_i. The total cost is thus $O((k+1)\,(1+\lg\lg n))$. □

Define the *interleave bound* $\mathrm{IB}_i(X)$ *of access* x_i to be the interleave bound on the prefix $x_1, x_2, \ldots, x_i$ of the access sequence minus the interleave bound on the shorter prefix $x_1, x_2, \ldots, x_{i-1}$. In other words, the interleave bound of access x_i is the number of additional interleaves introduced by access x_i.

Lemma 3.2 *The number of nodes whose preferred child changes from left to right or from right to left during an access x_i is equal to the interleave bound $\mathrm{IB}_i(X)$ of access x_i.*

Proof: The preferred child of a node y in P changes from left to right precisely when the previous access within y's subtree in P was in the left region of y and the next access x_i is in the right region of y. Symmetrically, the preferred child of node y changes from right to left precisely when the previous access within y's subtree in P was in the right region of y and the next access x_i is in the left region of y. Both of these events correspond exactly to interleaves. Note that these events do not include when node y previously had no preferred child and the first node within y's subtree in P is accessed. □

Theorem 3.3 *The running time of the Tango BST on an sequence X of m accesses over the universe $\{1, 2, \ldots, n\}$ is $O((\mathrm{OPT}(X)+n)\,(1+\lg\lg n))$ where $\mathrm{OPT}(X)$ is the cost of the offline optimal BST servicing X.*

Proof: Lemma 3.2 states that the total number of times a preferred child changes from left to right or from right to left is at most $\mathrm{IB}(X)$. There can be at most n first preferred child settings (i.e., changes from no preferred child to a left or right preference). Therefore the total number of preferred child changes is at most $\mathrm{IB}(X)+n$. Combining this bound with Lemma 3.1, the total cost of Tango is $O((\mathrm{IB}(X)+n+m)\,(1+\lg\lg n))$. On the other hand, Lemma 2.1 states that $\mathrm{OPT}(X) \geq \mathrm{IB}(X)/2 - n$. A trivial lower bound on all access sequences X is that $\mathrm{OPT}(X) \geq m$. Therefore, the running time of Tango is $O((\mathrm{OPT}(X)+n)\,(1+\lg\lg n))$. □

Corollary 3.4 *When $m = \Omega(n)$, the running time of the Tango BST is $O(\text{OPT}(X)\,(1 + \lg\lg n))$.*

3.5 Tightness of Approach

Observe that we cannot hope to improve the competitive ratio beyond $\Theta(\lg\lg n)$ using the current lower bound. At each moment in time, the preferred path from the root of P contains $\lg(n+1)$ nodes. Regardless of how the BST is organized, one of these $\lg(n+1)$ nodes must have depth $\Omega(\lg\lg n)$, which translates into a cost of $\Omega(\lg\lg n)$ for accessing that node. On the other hand, accessing any of these nodes increases the interleave bound by at most 1. Suppose we access node x along the preferred path from the root of P. The preferred children do not change for the nodes below x in the preferred path, nor do they change for the nodes above x. The preferred child of only x itself may change, in the case that the former preferred child was the right child, because we defined the preferred child of a just-accessed node x to be the left child. In conclusion, at any time, there is an access that costs $\Omega(\lg\lg n)$ in any fixed BST data structure, yet increases the interleave lower bound by at most 1, for a ratio of $\Omega(\lg\lg n)$.

Acknowledgments

We thank Richard Cole, Martin Farach-Colton, Michael L. Fredman, and Stefan Langerman for many helpful discussions.

References

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

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

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CMSS00] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[DHIP04] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality—almost. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 484–490, Rome, Italy, October 2004.

[HF98] Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.

[Iac01] John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, Washington, D.C., January 2001.

[Iac02] John Iacono. Key independent optimality. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 25–31, Vancouver, Canada, November 2002.

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

[PD04] Mihai Pătraşcu and Erik D. Demaine. Tight bounds for the partial-sums problem. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 20–29, New Orleans, Louisiana, January 2004.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 24(3):362–391, June 1983.

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

[Tar83] Robert Endre Tarjan. Linking and cutting trees. In *Data Structures and Network Algorithms*, chapter 5, pages 59–70. Society for Industrial and Applied Mathematics, 1983.

[Tar85] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, September 1985.

[WDS06] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. $O(\log \log n)$-competitive dynamic binary search trees. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, Miami, Florida, January 2006. To appear.

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

A Proof of Interleave Lower Bound

In this appendix, we prove Theorem 2.1, stated below as Theorem A. We assume a fixed but arbitrary BST access algorithm, and argue that the time it takes is at least the interleave bound. Let T_i denote the state of this arbitrary BST after the execution of accesses $x_1, x_2, \ldots, x_i$.

Consider the interleaving through a node y in P. Define the *transition point* for y at time i to be the minimum-depth node z in the BST T_i such that the path from z to the root of T_i includes a node from the left region of y and a node from the right region of y. (Here we ignore nodes not from y's subtree in P.) Thus the transition point z is in either the left or the right region of y, and it is the first node of that type seen along this root-to-node path. Intuitively, any BST access algorithm applied both to an element in the left region of y and to an element in the right region of y must touch the transition point for y at least once.

First we show that the notion of transition point is well-defined:

Lemma A.1 *For any node y in P and any time i, there is a unique transition point for y at time i.*

Proof: Let ℓ be the lowest common ancestor of all nodes in T_i that are in the left region of y. Because the lowest common ancestor of any two nodes in a binary search tree has a key value nonstrictly between these two nodes, ℓ is in the left region of y. Thus ℓ is the unique node of minimum depth in T_i among all nodes in the left region of y. Similarly, the lowest common ancestor r of all nodes in T_i in the right region of y must be in the right region of y and the unique such node of minimum depth in T_i. Also, the lowest common ancestor in T_i of all nodes in the left and right regions of y must be in either the left or right region of y (because they are consecutive in key space), and among such nodes it must be the unique node of minimum depth, so it must be either ℓ or r (whichever has smaller depth). Assume by symmetry that it is ℓ, so that ℓ is an ancestor of r. Thus r is a transition point for y in T_i, because the path in T_i from the root to r visits at least one node (ℓ) from the left region of y in P, and visits only one node (r) from the right region of y in P because it has minimum depth among such nodes. Furthermore, any path in T_i from the root must visit ℓ before any other node in the left or right region of y, because ℓ is an ancestor of all such nodes, and similarly it must visit r before any other node in the right region of y because it is an ancestor of all such nodes. Therefore r is the unique transition point for y in T_i. □

Second we show that the transition point is "stable", not changing until it is accessed:

Lemma A.2 *If the BST access algorithm does not touch a node z in T_i for all i in the time interval $[j, k]$, and z is the transition point for a node y at time j, then z remains the transition point for node y for the entire time interval $[j, k]$.*

Proof: Define ℓ and r as in the proof of the previous lemma, and assume by symmetry that ℓ is an ancestor of r in T_j, so that r is the transition point for y at time j. Because the BST access algorithm does not touch r, it does not touch any node in the right region of y, and thus r remains the lowest common ancestor of these nodes. On the other hand, the algorithm may touch nodes in the left region of y, and in particular the lowest common ancestor $\ell = \ell_i$ of these nodes may change with time (i). Nonetheless, we claim that ℓ_i remains an ancestor of r. Because nodes in the left region of y cannot newly enter r's subtree in T_i, and y is initially outside this subtree, some node ℓ'_i in the left region of y must remain outside this subtree in T_i. As a consequence, the lowest common ancestor a_i of ℓ'_i and r cannot be r itself, so it must be in the left region of y. Thus ℓ_i must be an ancestor of a_i, which is an ancestor of r, in T_i. □

Next we prove that these transition points are different over all nodes in P, enabling us to charge to them:

Lemma A.3 *At any time i, no node in T_i is the transition point for multiple nodes in P.*

Proof: Consider any two nodes y_1 and y_2 in P, and define ℓ_j and r_j in terms of y_j as in the proof of Lemma A.1. Recall that the transition point for y_j is either ℓ_j or r_j, whichever is deeper. If y_1 and y_2 are not ancestrally related in P, then their left and right regions are disjoint from each other, so ℓ_1 and r_1 are distinct from ℓ_2 and r_2, so the transition points for y_1 and y_2 are distinct. Otherwise, suppose by symmetry that y_1 is an ancestor of y_2 in P. If the transition point for y_1 is not in y_2's subtree in P (e.g., it is y_1, or it is in the left or right subtree of y_1 in P while y_2 is in the opposite subtree of y_1 in P), then it differs from ℓ_2 and r_2 and thus the transition point for y_2. Otherwise, the transition point for y_1 is the lowest common ancestor of all nodes in y_2's subtree in P, and thus it is either ℓ_2 or r_2, whichever is less deep. On the other hand, the transition point for y_2 is either ℓ_2 or r_2, whichever is deeper. Therefore the two transition points differ in all cases. □

Finally we prove that the interleave bound is a lower bound:

Theorem 2.1 $\mathrm{IB}(X)/2 - n$ *is a lower bound on* $\mathrm{OPT}(X)$*, the cost of the optimal offline BST that serves access sequence X.*

Proof: Instead of counting the entire cost incurred by the (optimal offline) BST, we just count the number of transition points it touches (which can be only smaller). By Lemma A.3, we can count the number of times the BST touches the transition point for y, separately for each y, and the sum these counts. Define ℓ and r as in the proof of Lemma A.1, so that the transition point for y is always either ℓ or r, whichever is deeper. Consider a maximal ordered subsequence $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ of accesses to nodes that alternate between being in the left and right regions of y. Thus p is the amount of interleaving through y. Assume by symmetry that the odd accesses $x_{i_{2j-1}}$ are nodes in the left region of y, and the even accesses $x_{i_{2j}}$ are nodes in the right region of y. Consider each j with $1 \leq j \leq \lfloor p/2 \rfloor$. Any access to a node in the left region of y must touch ℓ, and any access to a node in the right region of y must touch r. Thus, for both accesses $x_{i_{2j-1}}$ and $x_{i_{2j}}$ to avoid touching the transition point for y, the transition point must change from r to ℓ in between, which by Lemma A.2 requires touching the transition point for y. Thus the BST access algorithm must touch the transition point for y at least once during the time interval $[i_{2j-1}, i_{2j}]$. Summing over all j, the BST access algorithm must touch the transition point for y at least $\lfloor p/2 \rfloor \geq p/2 - 1$ times. Summing over all y, the amount p of interleaving through y adds up to the interleave bound $\mathrm{IB}(X)$, so the number of transition points touched adds up to at least $\mathrm{IB}(X)/2 - n$. □