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.

∗ 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

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. 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].

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 a constant number of bits 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. 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.

The goal of this line of research is to design a dynamically optimal 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 the first bound extensively in this paper, and describe it in detail in Section 2. This first bound also follows by noting that any BST offers an upper bound for the partial-sums problem in the semigroup model, and using the lower bounds from [HF98, PD04] for the latter problem.

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 in 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 data structure on every access sequence. Tango uses $O(1)$ bits of augmentation per node, and the book-keeping cost to determine the next BST operation is constant amortized.

The rest of the paper proceeds as follows. In Section 2 we present the first bound on Wilber, which we call the interleave lower bound. This lower bound is the basis of our competitive ratio. In Section 3 we describe Tango and prove that it is $O(\lg \lg n)$-competitive.

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 original by Wilber [Wil89], and is identical to the lower bounds that follow from partial sums in the semigroup model [HF98, PD04]. For self-containment, we include a proof of the lower bound in Appendix A.

We maintain a perfect binary tree, called P, on the keys $\{1, 2, \ldots, n\}$. (Assume that n is one less than a power of two so that we do not have to worry about imbalance.) This tree has a fixed structure over time. However, we additionally augment each internal node of the tree P to specify a *preferred child* of either left or right. Specifically, the preferred child of a node y in P corresponds to the child subtree containing the most recently accessed node within y's subtree; in the special case that the most recently accessed node is y itself, we define the preferred child to be the left child. Let P_i denote the state of this augmented structure after accesses $x_1, x_2, \ldots, x_i$.

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; and define the *right region* of y to consist of all nodes in y's right subtree. The left and right regions of y partition y's subtree 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. The *interleaving through* y is the number of alternations between "left" and "right" labels in this sequence. The total *interleave bound* $\mathrm{IB}(X)$ is the sum of these interleaving counts over all nodes y in P.

The exact statement of the lower bound is as follows:

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

Again, the interested reader is referred to Appendix A for a proof.

3. BST Upper Bound

3.1. Overview of Tango BST

We now define a specific BST access algorithm, Tango. Let T_i denote the state of the Tango BST after executing accesses $x_1, x_2, \ldots, x_i$. Again we maintain a perfect binary tree P on the same keys, augmented to store the preferred child for each node (whose subtree contains the most recently accessed item), as in the previous section. Thus the state P_i of this augmented perfect binary tree at any time 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. Follow the preferred child of the root of P, and the preferred child of that child, etc., until we reach a leaf. The nodes traversed by this process form a root-to-leaf path, called 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 off of auxiliary tree R as children.

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, 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; 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 maximum depth of a node 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.

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 walk 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 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 B and C of ℓ' corresponding to key ranges $(-\infty, \ell')$ and (ℓ', ∞). Then we split C at r' to form two subtrees D and E of r' corresponding to key ranges (ℓ', r') and (r', ∞). 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 larger depth 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', the marked root of B becomes the left child of r'. Then we unmark the root of B, merge at r', and then merge 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. Accessing x_i changes the necessary preferred children to make a preferred path from the root to x_i, and does not change any other preferred children. These points of change correspond exactly to where the search algorithm crosses from one augmented tree to the next, i.e., where it hits a marked node. For each such change in a preferred child, say changing the preferred child of x from left to right, we cut the augmented tree of the old path containing x at the left child of x, and then join the resulting top path (which contains x) with the augmented tree of the right child of x.

3.4. Analysis

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 3.1 *The number k of nodes whose preferred child changes during an access x_i is proportional to the interleave bound $\mathrm{IB}_i(X)$ of access x_i.*

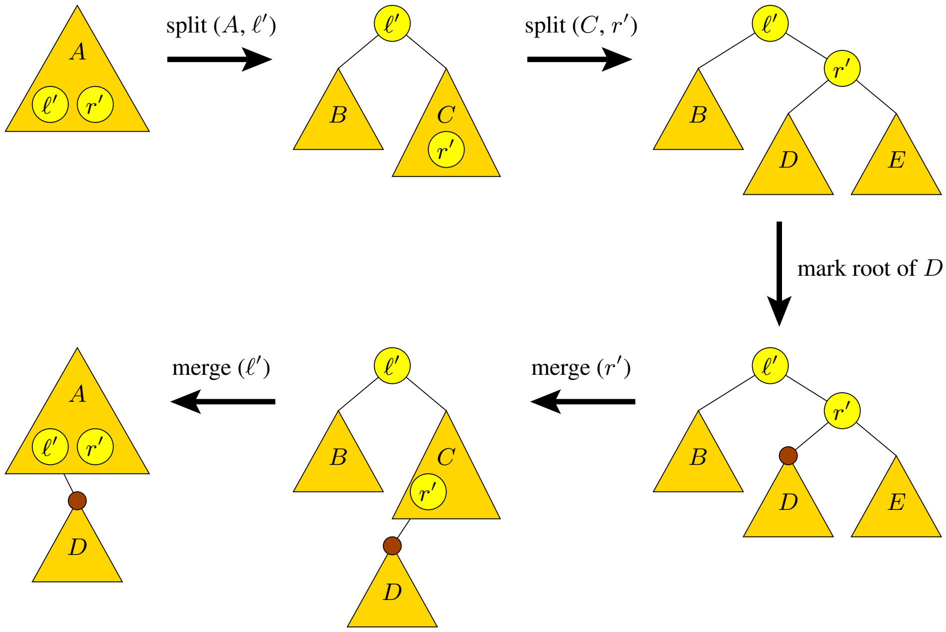

Figure 1. Implementing cut with split, mark, and merge.

Proof: The preferred child of a node y in P changes from left to right precisely when the previous access within y's subtree 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 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. □

Lemma 3.2 *The running time of an access x_i is $O(k\,(1 + \lg\lg n))$, 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, which is precisely where a node's preferred child changes because of this access. Thus the search path in T_{i-1} partitions into exactly $k+1$ 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 $O(k\,(1 + \lg\lg n))$. The update cost is the same as the search cost up to constant factors. For each of the $k+1$ auxiliary trees visited by the search, we perform one cut and one join, each costing $O(\lg\lg n)$, for a total cost of $O(k\,(1 + \lg\lg n))$. □

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

Proof: Lemmas 3.1 and 3.2 together states that Tango's cost per access x_i is $O(\mathrm{IB}_i(X)\,(1 + \lg\lg n))$. Summing over all i, the total cost of Tango is $O(\mathrm{IB}(X)\,(1 + \lg\lg n))$. On the other hand, Lemma 2.1 states that $\mathrm{OPT}(X) \geq \mathrm{IB}(X) - O(n)$, i.e., $\mathrm{IB}(X) \leq \mathrm{OPT}(X) + O(n)$. Therefore, the running time of Tango is $O((\mathrm{OPT}(X) + O(n))\,(1 + \lg\lg n))$. □

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

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 (arbitrarily) 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*, Buenos Aires, Argentina, April 2004. To appear.

[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.

[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.

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

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

[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.4. 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 highest 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.

First we show that this definition is well-defined:

Lemma A.1 *The transition point z in T_i for a node y in P is unique.*

Proof: Let l be the lowest common ancestor of all of the nodes in T_i that are in the left region of y in P. Because the lowest common ancestor of any two nodes in a binary search tree has a key value between these two nodes, l is in the left region of y in P. Similarly, define r be the lowest common ancestor of all of the nodes in T_i that are in the right region of y in P. The lowest common ancestor of l and r in T_i must be either l or r, because they are adjacent in key space. Assume by symmetry that it is l. We claim that r is the unique transition point for y in z. This claim follows because r has at least one ancestor in the left region of y in P, namely l, and because all other nodes in T_i in the right region of y in P are in r's subtree. Thus r is the unique first node on any path containing elements from both the left and right regions of y in P. □

Second we show that the transition point is "stable":

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

Proof: Using the same definition of l and r as in the previous lemma, and we assume that, at time j, l is an ancestor of r in the tree. The transition point can not change in $[j, k]$ because all elements of the right region of y in P will remain in the subtree of r, and no elements of the left region of y in P can move into the subtree of r without r being touched. Thus least one element of the left region of y is P must be an ancestor of r (this will be l at time j but may change), and therefore r remains the unique first element of every path containing elements from the left and right regions of y in P. □

Next we prove that these transition points are different for all the nodes in P:

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

Proof: The proof is by contradiction. Suppose a node x in T_i is a transition point for two distinct nodes r and s in P. Because x is in either the left or right region of both r and s in P, both r and s are ancestors of x in P. Assume by symmetry that r is above s in P. The transition point of s is the lowest common ancestor in T_i of either the left or the right region of s in P. It cannot be the lowest common ancestor of all of the nodes in the subtree of s in P. The transition point of r is the lowest common ancestor in T_i of either the left or the right region of r in P. Because r is an ancestor of s in P, one of these two regions is disjoint from s's subtree and the other region contains s's subtree. Thus the transition points for r and s are distinct. □

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

Theorem A.4 $\mathrm{IB}(X)/2 - O(n)$ *is a lower bound on* $\mathrm{OPT}(X)$*, the cost of the optimal offline BST that serves access sequence X.*

Proof: We define an adversarial game between the lower bound and the BST access algorithm, involving the distribution of marbles.

The lower bound plays as follows. Consider any left-to-right interleave through node y in P, that is, two adjacent accesses x_i and x_j to y's subtree such that $x_i \leq y < x_j$ and $i < j$. (Thus x_i is in the left region of y, x_j is in the right region of y, and every access between x_i and x_j is outside y's subtree.) Then, immediately after the execution of access x_i, the lower bound places a marble on the transition point for node y.

The BST access algorithm plays as follows. Whenever it touches a node in a tree T_i, it discards any marbles on that node.

First we claim that there is at most one marble on any node in T_i at any time i. By Lemma A.3, the marbles from different nodes y and y' in P do not interfere. For any node y in P, we place a marble only after accessing an element in the left region of y and when the next access to an element in y's subtree is in the right region of y. This event will not occur again until the element in the right region of y has been accessed, which requires that it be touched, which requires that the transition point for y be touched.

Whenever the algorithm takes a marble, it is paying a unit of cost to touch the node storing the marble. Therefore the number of marbles picked up is a lower bound on the running time of the algorithm. The number of marbles picked up is at least the number of marbles placed minus n, because every node stores at most one marble. As argued above, the number of marbles placed is exactly the number of left-to-right interleaves, which is at least one half of the interleave bound minus n, one for each node y in P. □