Combining Binary Search Trees*

Erik D. Demaine[1], John Iacono[2**], Stefan Langerman[***3], and Özgür Özkan[2†]

[1] Massachusetts Institute of Technology
[2] Polytechnic Institute of New York University
[3] Université Libre de Bruxelles

Abstract. We present a general transformation for combining a constant number of binary search tree data structures (BSTs) into a single BST whose running time is within a constant factor of the minimum of any "well-behaved" bound on the running time of the given BSTs, for any online access sequence. (A BST has a *well-behaved* bound with $f(n)$ overhead if it spends at most $\mathcal{O}(f(n))$ time per access and its bound satisfies a weak sense of closure under subsequences.) In particular, we obtain a BST data structure that is $\mathcal{O}(\log\log n)$ competitive, satisfies the working set bound (and thus satisfies the static finger bound and the static optimality bound), satisfies the dynamic finger bound, satisfies the unified bound with an additive $\mathcal{O}(\log\log n)$ factor, and performs each access in worst-case $\mathcal{O}(\log n)$ time.

1 Introduction

Binary search trees (BSTs) are one of the most fundamental and well-studied data structures in computer science. Yet, many fundamental questions about their performance remain open. While information theory dictates the worst-case running time of a single access in an n node BST to be $\Omega(\log n)$, which is achieved by many BSTs (e.g., [2]), BSTs are generally not built to execute a single access, and there is a long line of research attempting to minimize the overall running time of executing an online access sequence. This line of work was initiated by Allen and Munro [1], and then by Sleator and Tarjan [15] who invented the splay tree. Central to splay trees and many of the data structures in the subsequent literature is the BST model. The BST model provides a precise model of computation, which is not only essential for comparing different BSTs, but also allows the obtaining of lower bounds on the optimal offline BST.

In the BST model, the elements of a totally ordered set are stored in the nodes of a binary tree and a BST data structure is allowed at unit cost to manipulate the tree by following the parent, left-child, or right-child pointers at each node or rotate the node with its parent. We give a formal description of

* See [9] for the full version.

** Research supported by NSF Grant CCF-1018370

*** Directeur de Recherches du F.R.S.-FNRS. Research partly supported by the F.R.S.-FNRS and DIMACS.

† Research supported by GAANN Grant P200A090157 from the US Department of Education.

the model in Section 1.1. A common theme in the literature since the invention of splay trees concerns proving various bounds on the running time of splay trees and other BST data structures [4, 6, 7, 12, 15]. Sleator and Tarjan [15] proved a number of upper bounds on the performance of splay trees. The *static optimality* bound requires that any access sequence is executed within a constant factor of the time it would take to execute it on the best static tree for that sequence. The *static finger* bound requires that each access x is executed in $\mathcal{O}(\log d(f, x))$ amortized time where $d(f, x)$ is the number of keys between any fixed finger f and x. The *working set* bound requires that each access x is executed in $\mathcal{O}(\log w(x))$ amortized time where $w(x)$ is the number of elements accessed since the last access to x. Cole [6] and Cole et al. [7] later proved that splay trees also have the *dynamic finger* bound which requires that each access x is executed in $\mathcal{O}(\log d(y, x))$ amortized time where y is the previous item in the access sequence. Iacono [14] introduced the *unified* bound, which generalizes and implies both the dynamic finger and working set bounds. Bose et al. [4] presented layered working set trees, and showed how to achieve the unified bound with an additive cost of $\mathcal{O}(\log \log n)$ per access, by combining them with the skip-splay trees of Derryberry and Sleator [12].

A BST data structure satisfies the dynamic optimality bound if it is $\mathcal{O}(1)$-competitive with respect to the best offline BST data structure. Dynamic optimality implies all other bounds of BSTs. The existence of a dynamically optimal BST data structure is a major open problem. While splay trees were conjectured by Sleator and Tarjan to be dynamically optimal, despite decades of research, there were no online BSTs known to be $o(\log n)$-competitive until Demaine et al. invented Tango trees [8] which are $\mathcal{O}(\log \log n)$-competitive. Later, Wang et al. [17] presented a variant of Tango trees, called multi-splay trees, which are also $\mathcal{O}(\log \log n)$-competitive and retain some bounds of splay trees. Bose et al. [3] gave a transformation where given any BST whose amortized running time per access is $\mathcal{O}(\log n)$, they show how to deamortize it to obtain $\mathcal{O}(\log n)$ worst-case running time per access while preserving its original bounds.

Results and Implications. In this paper we present a structural tool to combine bounds of BSTs from a certain general class of BST bounds, which we refer to as well-behaved bounds. Specifically, our method can be used to produce an online BST data structure which combines well-behaved bounds of all known BST data structures. In particular, we obtain a BST data structure that is $\mathcal{O}(\log \log n)$ competitive, satisfies the working set bound (and thus satisfies the static finger bound and the static optimality bound), satisfies the dynamic finger bound, satisfies the unified bound with an additive $\mathcal{O}(\log \log n)$, and performs each access in worst-case $\mathcal{O}(\log n)$ time. Moreover, we can add to this list any well-behaved bound realized by a BST data structure.

Note that requiring the data structures our method produces to be in the BST model precludes the possibility of a trivial solution such as running all data structures in parallel and picking the fastest.

Our result has a number of implications. First, it could be interpreted as a weak optimality result where our method produces a BST data structure which is $\mathcal{O}(1)$-competitive with respect to a constant number of given BST data structures whose actual running times are well-behaved. In comparison, a dynamically optimal BST data structure, if one exists, would be $\mathcal{O}(1)$-competitive with respect to all BST data structures. On the other hand, the existence of our method is a necessary condition for the existence of a dynamically optimal BST. Lastly, techniques introduced in this paper (in particular the simulation of multiple fingers in Section 2) may be of independent interest for augmenting a BST in nontrivial ways, as we do here. Indeed, they are also used in [5].

1.1 Preliminaries

The BST model. Given a set S of elements from a totally ordered universe, where $|S| = n$, a BST data structure T stores the elements of S in a rooted tree, where each node in the tree stores an element of S, which we refer to as the key of the node. The node also stores three pointers pointing to its parent, left child, and right child. Any key contained in the left subtree of a node is smaller than the key stored in the node; and any key contained in the right subtree of a node is greater than the key stored in the node. Each node can store data in addition to its key and the pointers.

Although BST data structures usually support insertions, deletions, and searches, in this paper we consider only successful searches, which we call *accesses*. To implement such searches, a BST data structure has a single pointer which we call the finger, pointed to a node in the BST T. The finger initially points to the root of the tree before the first access. Whenever a finger points to a node as a result of an operation o we say the node is *touched*, and denote the node by $N(o)$. An access sequence $(x_1, x_2, \ldots, x_m)$ satisfies $x_i \in S$ for all i. A BST data structure executes each access i by performing a sequence of unit-cost operations on the finger—where the allowed unit-cost operations are following the left-child pointer, following the right-child pointer, following the parent pointer, and performing a rotation on the finger and its parent—such that the node containing the search key x_i is touched as a result of these operations. Any augmented data stored in a node can be modified when the node is touched during an access. The running time of an access is the number of unit-cost operations performed during that access.

An *offline* BST data structure executes each operation as a function of the entire access sequence. An *online* BST data structure executes each operation as a function of the prefix of the access sequence ending with the current access. Furthermore, as coined by [3], a *real-world* BST data structure is one which can be implemented with a constant number of $\mathcal{O}(\log n)$ bit registers and $\mathcal{O}(\log n)$ bits of augmented data at each node.

The Multifinger-BST model. The Multifinger-BST model is identical to the BST model with one difference: in the Multifinger-BST model we have access to a constant number of fingers, all initially pointing to the root.

We now formally define what it means for a BST data structure to simulate a Multifinger-BST data structure.

Definition 1. *A BST data structure S* simulates *a Multifinger-BST data structure M if there is a correspondence between the ith operation $op_M[i]$ performed by M and a contiguous subsequence $op_S[j_i], op_S[j_i+1], \ldots, N(op_S[j_{i+1}-1])$ of the operations performed by S, for $j_1 < j_2 < \cdots$, such that the touched nodes satisfy $N(op_M[i]) \in \{N(op_S[j_i]), N(op_S[j_i+1]), \ldots, N(op_S[j_{i+1}-1])\}$.*

For any access sequence X, there exists an offline BST data structure that executes it optimally. We denote the number of unit-cost operations performed by this data structure by $\mathrm{OPT}(X)$. An online BST data structure is c-competitive if it executes all sequences X of length $\Omega(n)$ in time at most $c \cdot \mathrm{OPT}(X)$, where n is the number of nodes in the tree. An online BST that is $\mathcal{O}(1)$-competitive is called *dynamically optimal.*

Because a BST data structure is a Multifinger-BST data structure with one finger, the following definitions apply to BST data structures as well.

Definition 2. *Given a Multifinger-BST data structure A and an initial tree T, let $\mathcal{T}(A, T, X) = \sum_{i=1}^{|X|} \tau(A, T, X, i) + f(n)$ be an upper bound on the total running time of A on any access sequence X starting from tree T, where $\tau(A, T, X, i)$ denotes an amortized upper bound on the running time of A on the ith access of access sequence X, and $f(n)$ denotes the overhead. Define $\mathcal{T}_{X'}(A, T, X) = \sum_{i=1}^{|X'|} \tau(A, T, X, \pi(i))$ where X' is a contiguous subsequence of access sequence X and $\pi(i)$ is the index of the ith access of X' in X. The bound τ is **well-behaved** with overhead $f(n)$ if there exists constants C_0 and C_1 such that the cost of executing any single access x_i is at most $C_1 \cdot f(n)$, and for any given tree T, access sequence X, and any contiguous subsequence X' of X, $\mathcal{T}(A, T, X') \leq C_0 \cdot \mathcal{T}_{X'}(A, T, X) + C_1 \cdot f(n)$.*

1.2 Our Results

Given k online BST data structures $\mathcal{A}_1, \ldots, \mathcal{A}_k$, where k is a constant, our main result is the design of an online BST data structure which takes as input an online access sequence $(x_1, \ldots, x_m)$, along with an initial tree T; and executes, for all j, access sequence $(x_1, \ldots, x_j)$ in time $\mathcal{O}\left(\min_{i \in \{1,\ldots,k\}} \mathcal{T}(\mathcal{A}_i, T, (x_1, \ldots, x_j))\right)$ where $\mathcal{T}(\mathcal{A}_i, T, X)$ is a well-behaved bound on the running time of $\mathcal{A}_i$. To simplify the presentation, we let $k = 2$. By combining k BSTs two at a time, in a balanced binary tree, we achieve an $O(k)$ (constant) overhead.

Theorem 3. *Given two online BST data structures $\mathcal{A}_0$ and $\mathcal{A}_1$, let $\mathcal{T}_{X'}(\mathcal{A}_0, T, X)$ and $\mathcal{T}_{X'}(\mathcal{A}_1, T, X)$ be well-behaved amortized upper bounds with overhead $f(n) \geq n$ on the running time of $\mathcal{A}_0$ and $\mathcal{A}_1$, respectively, on a contiguous subsequence X' of any online access sequence X from an initial tree T. Then there exists an online BST data structure,* Combo-BST $=$ Combo-BST$(\mathcal{A}_0, \mathcal{A}_1, f(n))$ *such that*

$$\mathcal{T}_{X'}(\textsf{Combo-BST}, T, X) = \mathcal{O}(\min(\mathcal{T}_{X'}(\mathcal{A}_0, T, X), \mathcal{T}_{X'}(\mathcal{A}_1, T, X)) + f(n)).$$

If $\mathcal{A}_0$ and $\mathcal{A}_1$ are real-world BST data structures, so is Combo-BST.

Corollary 4. *There exists a BST data structure that is $\mathcal{O}(\log \log n)$-competitive, satisfies the working set bound (and thus satisfies the static finger bound and the static optimality bound), satisfies the dynamic finger bound, satisfies the unified bound[4] with an additive $\mathcal{O}(\log \log n)$, all with additive overhead $\mathcal{O}(n \log n)$, and*

[4] The Cache-splay tree [11] was claimed to achieve the unified bound. However, this claim has been rescinded by one of the authors at the 5th Bertinoro Workshop on Algorithms and Data Structures.

performs each access in worst-case $\mathcal{O}(\log n)$ time.

Proof. We apply Theorem 3 to combine the bounds of the splay tree, the multi-splay tree [17], and the layered working set tree [4]. The multi-splay tree is $\mathcal{O}(\log\log n)$-competitive. Observe that $\mathrm{OPT}(X)$ is a well-behaved bound with overhead $\mathcal{O}(n)$ because any tree can be transformed to any other tree in $\mathcal{O}(n)$ time [16]. Therefore, $\mathcal{O}(\log\log n)$-competitiveness of multi-splay trees is a well-behaved bound with overhead $\mathcal{O}(n\log\log n)$. On the other hand, the multi-splay tree also satisfies the working set bound. The working set bound is a well-behaved bound with $\mathcal{O}(n\log n)$ overhead because only the first instance of each item in a subsequence of an access sequence has a different working set number with respect to that subsequence and the log of each such difference is upper bounded by $\log n$. The working set bound implies the static finger and static optimality bounds with overhead $O(n\log n)$ [13]. The splay tree satisfies the the dynamic finger bound [6, 7], which is a well-behaved bound with $\mathcal{O}(n)$ overhead because the additive term in the dynamic finger bound is linear and only the first access in a subsequence may have an increase in the amortized bound which is at most $\log n$. The layered working set tree [4] satisfies the unified bound with an additive $\mathcal{O}(\log\log n)$. Similar to the working set bound, the unified bound is a well-behaved bound with $\mathcal{O}(n\log n)$ overhead because only the first instance of each item in a subsequence of an access sequence has a different unified bound value with respect to that subsequence and each such difference is at most $\log n$. Therefore, because the $\mathcal{O}(\log\log n)$ term is additive and is dominated by $\mathcal{O}(\log n)$, the unified bound with an additive $\mathcal{O}(\log\log n)$ is a well-behaved bound with $\mathcal{O}(n\log n)$ overhead. Lastly, because the multi-splay tree performs each access in $\mathcal{O}(\log n)$ worst-case time and because $\mathcal{O}(\log n)$ is a well-behaved bound with no overhead, we can apply the transformation of Bose et al. [3] to our BST data structure to satisfy all of our bounds while performing each access in $\mathcal{O}(\log n)$ worst-case time. □

To achieve these results, we present OneFinger-BST, which can simulate any Multifinger-BST data structure in the BST model in constant amortized time per operation. We will present our Combo-BST data structure as a Multifinger-BST data structure in Section 4 and use OneFinger-BST to transform it into a BST data structure.

Theorem 5. *Given any Multifinger-BST data structure A, where $op_A[j]$ is the jth operation performed by A,* OneFinger-BST(A) *is a BST data structure such that, for any k, given k operations $(op_A[1], \ldots, op_A[k])$ online,* OneFinger-BST(A) *simulates them in $C_2 \cdot k$ total time for some constant C_2 that depends on the number of fingers used by A. If A is a real-world BST data structure, then so is* OneFinger-BST(A).

Organization and Roadmap. The rest of the paper is organized as follows. We present a method to transform a Multifinger-BST data structure to a BST data structure (Theorem 5) in Section 2. We show how to load and save the state of the tree in $\mathcal{O}(n)$ time in the Multifinger-BST model using multiple fingers in Section 3. We present our BST data structure, the Combo-BST, in Section 4. We analyze Combo-BST and prove our Theorem 3 in Section 5.

2 Simulating Multiple Fingers

In this section, we present OneFinger-BST, which transforms any given Multifinger-BST data structure T into a BST data structure OneFinger-BST(T).

Structural Terminology. First we present some structural terminology defined by [10], adapted here to the BST model; refer to Figure 1.

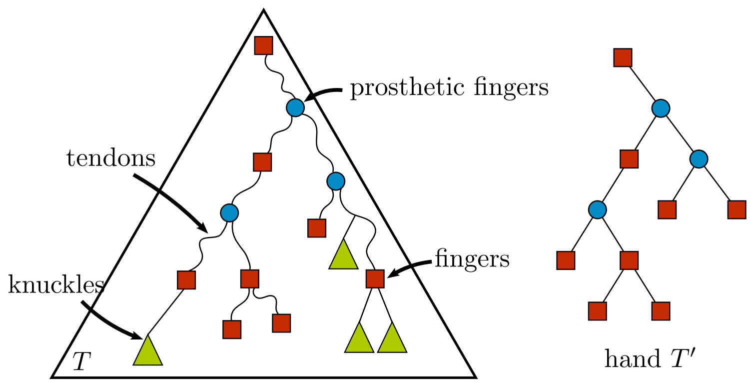


Fig. 1: A tree T with a set of fingers, and the corresponding hand structure.

Given any Multifinger-BST T with a set F of fingers $f_1, \ldots, f_{|F|}$, where $|F| = \mathcal{O}(1)$, let $S(T, F)$ be the be the Steiner tree with terminals f_i, that is, the union of shortest paths in T between all pairs of fingers[5] F. We define *prosthetic fingers*, denoted by $P'(T, F)$, to be the set of nodes with degree 3 in $S(T, F)$ that are not in F. Then, we define the set of *pseudofingers*, denoted by $P(T, F)$, to be $P(T, F) = F \cup P'(T, F)$. Note that $|P(T, F)| \leq 2|F| = \mathcal{O}(1)$. The *hand* $H(T, F)$ is the compressed Steiner tree obtained from the Steiner tree $S(T, F)$ by contracting every vertex not in $P(T, F)$ (each of degree 2). A *tendon* $\tau_{x,y}$ is the shortest path in $S(T, F)$ connecting two pseudofingers x and y (excluding nodes x and y), where x is an ancestor of y and x and y are adjacent in $H(T, F)$. We refer to x as the top of $\tau_{x,y}$ and y as the bottom of $\tau_{x,y}$. A *knuckle* is a connected component of T after removing all of its pseudofingers and tendons.

To avoid confusion, we use $\mathsf{parent}_T(x)$, $\mathsf{left\text{-}child}_T(x)$, and $\mathsf{right\text{-}child}_T(x)$ to denote the pointers of a node x in the Multifinger-BST T, and use $\mathsf{parent}(x)$, $\mathsf{left\text{-}child}(x)$, and $\mathsf{right\text{-}child}(x)$ to denote the pointers of a node x in OneFinger-BST(T).

Our Approach. To simulate a Multifinger-BST data structure, OneFinger-BST needs to handle the movement and rotation of multiple fingers. To accomplish this, OneFinger-BST maintains the hand structure. We discuss how this is done at a high level in Section 2.2. However, a crucial part of maintaining the hand

[5] For convenience, we also define the root of the tree T to be a finger.

is an efficient implementation of tendons in the BST model where the distance between any two fingers connected by a tendon is at most a constant. We will implement a tendon as a pair of double-ended queues (deques). The next lemma lets us do this by showing that a tendon consists of an increasing and a decreasing subsequence. See Figure 2a.

Lemma 6. *A tendon can be partitioned into two subsets of nodes $T_>$ and $T_<$ such that the level-order key values of nodes in $T_>$ are increasing and the level-order key values of nodes in $T_<$ are decreasing, where the maximum key value in $T_>$ is smaller than the minimum key value in $T_<$.*

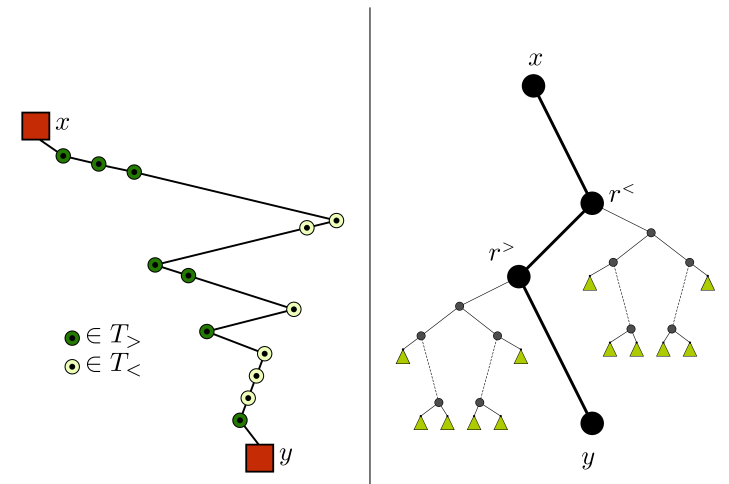
Proof. Letting $T_>$ to be the set of all nodes in tendon $\tau_{x,y}$ whose left-child is also in the tendon, and $T_<$ to be the set of all nodes in tendon $\tau_{x,y}$ whose right-child is also in the tendon yields the statement of the lemma. ☐

Deque-BST. It is straightforward to implement double ended queues (deques) in the BST model. We implement the deques for storing $T_>$ and $T_<$ symmetrically. LeftDeque-BST is a BST data structure storing the set of nodes in $T_>$ and RightDeque-BST is a BST data structure symmetric to LeftDeque-BST storing the set of nodes in $T_<$. We denote the roots of LeftDeque-BST and RightDeque-BST by $r^>$ and $r^<$ respectively. Because they are symmetric structures, we only describe LeftDeque-BST. Any node x to be pushed into a LeftDeque-BST is given as the parent of the $r^>$. Similarly, whenever a node is popped from LeftDeque-BST, it becomes the parent of $r^>$. LeftDeque-BST supports the following operations in constant amortized time: $\mathsf{push\text{-}min}(T_>, x)$, $\mathsf{push\text{-}max}(T_>, x)$, $\mathsf{pop\text{-}min}(T_>)$, $\mathsf{pop\text{-}max}(T_>)$.

2.1 Tendon-BST

We now present a BST data structure, Tendon-BST, which supports the following operations on a given tendon $\tau_{x,y}$, where $x' = \mathsf{parent}_T(x)$ and $y' = \mathsf{parent}_T(y)$, $\tau_{x',y} \leftarrow \mathsf{AddTop}(\tau_{x,y})$, $\tau_{x,y} \leftarrow \mathsf{AddBottom}(\tau_{x,y'}, y)$, $\tau_{x,y} \leftarrow \mathsf{RemoveTop}(\tau_{x',y})$, $\tau_{x,y'} \leftarrow \mathsf{RemoveBottom}(\tau_{x,y})$.

Implementation. We implement the Tendon-BST operations using LeftDeque-BST and RightDeque-BST. See Figure 2b. Nodes x, $r^>$, $r^<$, and y form a path in Tendon-BST where node x is an ancestor of nodes $r^>$, $r^<$, and y; and node y is a descendant of nodes x, $r^>$, and $r^<$. There are four such possible paths and the particular one formed depends on the key values of x and y, and the relationship between node y and its parent in T. These invariants imply that the distance between x and y is 3. When we need to insert a node into the tendon, we perform a constant number of rotations to preserve the invariants and position the node appropriately as the parent of $r^>$ or $r^<$. We then call the appropriate LeftDeque-BST or RightDeque-BST operation. Removing a node from the tendon is symmetric. Because deques (LeftDeque-BST, RightDeque-BST) can be implemented in constant amortized time per operation, and Tendon-BST performs a constant number of unit-cost operations in addition to one deque operation, it supports all of its operations in constant amortized time.



(a) $\tau_{x,y}$ and sets $T_>$ and $T_<$ (b) Tendon-BST storing $\tau_{x,y}$.

Fig. 2: A tendon in Multifinger-BST T and how its stored using Tendon-BST.

2.2 OneFinger-BST

At a high level, the OneFinger-BST data structure maintains the hand $H(T, F)$, which is of constant size, where each node corresponds to a pseudofinger in $P(T, F)$. For each pseudofinger x the parent pointer, parent(x), either points to another pseudofinger or the bottom of a tendon, and the child pointers, left-child(x), right-child(x), each point to either another pseudofinger, or the root of a knuckle, or the top of a tendon.

Intuitively, the Tendon-BST structure allows us to *compress* a tendon down to constant depth. Whenever an operation is to be done at a finger, OneFinger-BST *uncompresses* the 3 surrounding tendons until the elements at distance up to 3 from the finger are as in the original tree, then performs the operation. OneFinger-BST then reconstructs the hand structure locally, possibly changing the set of pseudofingers if needed, and *recompresses* all tendons using Tendon-BST. This is all done in amortized $\mathcal{O}(1)$ time because Tendon-BST operations used for decompression and recompression take amortized $\mathcal{O}(1)$ time and reconfiguring any constant size local subtree into any shape takes $\mathcal{O}(1)$ time in the worst case.

ADT. OneFinger-BST is a BST data structure supporting the following operations on a Multifinger-BST T with a set F of fingers $f_1, \ldots, f_{|F|}$, where $|F| = \mathcal{O}(1)$:

- MoveToParent(f_i): Move finger f_i to its parent in T, parent$_T(f_i)$.
- MoveToLeftChild(f_i): Move finger f_i to its left-child in T, left-child$_T(f_i)$.
- MoveToRightChild(f_i): Move finger f_i to its right-child in T, right-child$_T(f_i)$.
- RotateAt(f_i): Rotate finger f_i with its parent in T, parent$_T(f_i)$.

Implementation. We augment each node with a $\mathcal{O}(|F|)$ bit field to store the type of the node and the fingers currently on the node.

All the OneFinger-BST operations take as input the finger they are to be performed on. We first do a brute force search using the augmented bits mentioned above to find the node pointed to by the input finger. Note that all such fingers will be within a $\mathcal{O}(1)$ distance from the root . We then perform the operation as well as the relevant updates to the tree to reflect the changes in $H(T, F)$. Specifically, in order to perform the operation, we extract the relevant nodes from the surrounding tendons of the finger by calling the appropriate Tendon-BST functions. We perform the operation and update the nodes to reflect the structural changes to the hand structure. Then we insert the tendon nodes back into their corresponding tendons using the appropriate Tendon-BST functions.

Theorem 7. *Given any Multifinger-BST data structure A, where* $op_A[j]$ *is the* j*th operation performed by A,* OneFinger-BST(A) *is a BST data structure such that, for any* k*, given* k *operations* $(op_A[1], \ldots, op_A[k])$ *online,* OneFinger-BST(A) *simulates them in* $C_2 \cdot k$ *total time for some constant* C_2 *that depends on the number of fingers used by A. If A is a real-world BST data structure, then so is* OneFinger-BST(A).

Proof. Note that before $op_M[1]$, all the fingers are initialized to the root of the tree and therefore the potentials associated with the deques of the tendons is zero. All finger movements and rotations are performed using OneFinger-BST operations. Each operation requires one finger movement or rotation and at most a constant number of Tendon-BST operations, as well as the time it takes to traverse between fingers. Because the time spent traversing between a constant number of fingers is at most a constant, this implies that OneFinger-BST(M) simulates operations $(op_M[1], \ldots, op_M[k])$ in $C_2 \cdot k$ time for any k. □

3 Multifinger-BST with Buffers

In our model, each node in the tree is allowed to store $\mathcal{O}(\log n)$ bits of augmented data. In this section, we show how to use this $\mathcal{O}(n \log n)$ collective bits of data to implement a traversable "buffer" data structure. More precisely, we show how to augment any Multifinger-BST data structure into a structure called Buf-MFBST supporting buffer operations.

Definition 8. *A* buffer *is a sequence* $b_1, b_2, \ldots, b_n$ *of cells, where each cell can store* $\mathcal{O}(\log n)$ *bits of data. The buffer can be traversed by a constant number of buffer-fingers, each initially on cell* b_1*, and each movable forwards or backwards one cell at a time.*

ADT. In addition to Multifinger-BST operations, Buf-MFBST supports the following operations on any buffer-finger bf of a buffer: bf.PreviousCell(), bf.NextCell(), bf.ReadCell(), bf.WriteCell(d).

Implementation. We store the jth buffer cell in the jth node in the in-order traversal of the tree. $\mathsf{PreviousCell}()$ and $\mathsf{NextCell}()$ are performed by traversing to the previous or next node respectively in the in-order traversal of the tree.

Lemma 9. *Given a tree T, traversing it in-order (or symmetrically in reverse-in-order) with a finger, interleaved with r rotation operations performed by other fingers, takes $\mathcal{O}(n + r)$ time.*

Proof. The cost of traversing the tree in-order is at most $2n$. Each rotation performed in between the in-order traversal operations can increase the total length of any path corresponding to a subsequence of the in-order traversal by at most one. Thus, the cost of traversing T in-order, interleaved with r rotation operations, is $\mathcal{O}(n + r)$. □

Tree state. We present an augmentation in the Multifinger-BST model, which we refer to as TSB, such that given any Multifinger-BST data structure M, TSB augments M with a tree state buffer. The following operations are supported on a tree state buffer: $\mathsf{SaveState}()$: save the current state of the tree on the tree state buffer, $\mathsf{LoadState}()$: transform the current tree to the state stored in the tree state buffer. Let the encoding of the tree state be a sequence of operations performed by a linear time algorithm $\mathsf{LeftifyTree}(T)$ that transforms the tree into a left path. There are numerous folklore linear time implementations of such an algorithm. We can save the state of T by calling $\mathsf{LeftifyTree}(T)$ and recording the performed operations in the tree state buffer. To load a tree state in the tree state buffer, we call $\mathsf{LeftifyTree}(T)$ then undo all the operations in the tree state buffer. Note that TSB maintains up to n cells but we may need to store more data. We can either pack more data into each cell or use multiple copies of TSB. We can also apply TSB to itself to allow for multiple buffers.

Lemma 10. *Given any Multifinger-BST data structure M with k fingers, $\mathsf{TSB}(M)$ is a Multifinger-BST data structure with $\mathcal{O}(k)$ fingers such that the number of operations performed to execute $\mathsf{SaveState}()$ or $\mathsf{LoadState}()$ is $\mathcal{O}(n)$.*

Proof. Because $\mathsf{LeftifyTree}(T)$ runs in linear time, there can be only $\mathcal{O}(n)$ rotations, and by Lemma 9 the running time of both operations is $\mathcal{O}(n)$. □

4 Combo-MFBST and Combo-BST

Given two online BST data structures $\mathcal{A}_0$ and $\mathcal{A}_1$, let $\mathcal{T}_{X'}(\mathcal{A}_0, T, X)$ be any well-behaved upper bound with overhead $f(n) \geq n$ on the running time of $\mathcal{A}_0$, and let $\mathcal{T}_{X'}(\mathcal{A}_1, T, X)$ be any well-behaved upper bound with overhead $f(n) \geq n$ on the running time of $\mathcal{A}_1$ on a contiguous subsequence X' of X, for any online access sequence X and initial tree T. Then $\mathsf{Combo\text{-}MFBST}$ = $\mathsf{Combo\text{-}MFBST}(\mathcal{A}_0, \mathcal{A}_1, f(n))$ is defined as follows. It uses a tree state buffer ST_* implemented as a TSB. It stores the initial tree state T in ST_* by calling $\mathsf{SaveState}()$ before executing any accesses. Then, $\mathsf{Combo\text{-}MFBST}$ executes any online access sequence in rounds by alternating between emulating $\mathcal{A}_0$ and $\mathcal{A}_1$. Specifically, each round consists of $C_3 \cdot f(n)$ operations that execute the access sequence using BST data structure $\mathcal{A}_\mu$, for $\mu \in \{0,1\}$, always starting from the initial tree T. When the operation limit $C_3 \cdot f(n)$ gets reached, say in the middle of executing access x_i, Combo-MFBST transforms the tree back to its initial state T by calling LoadState() on ST_*; and toggles the active BST data structure by setting μ to $1-\mu$. The next round re-runs access x_i, this time on the opposite BST data structure. By picking a suitably large C_3, we ensure that every round completes at least one access, and thus no access gets executed by the same BST data structure in more than one round.

Lemma 11. *Given two BST data structures, $\mathcal{A}_0$ and $\mathcal{A}_1$,* Combo-MFBST $=$ Combo-MFBST$(\mathcal{A}_0, \mathcal{A}_1, f(n))$ *for any $f(n)$ is a Multifinger-BST data structure with $\mathcal{O}(1)$ fingers. Furthermore, if $\mathcal{A}_0$ and $\mathcal{A}_1$ are real-world BST data structures, then so is* Combo-MFBST.

Proof. Combo-MFBST$(\mathcal{A}_0, \mathcal{A}_1, f(n))$ has one tree state buffer which has $\mathcal{O}(1)$ fingers by Lemma 10. Because TSB augments each node with at most $\mathcal{O}(\log n)$ bits and Combo-MFBST uses only a constant number of registers each of size $\mathcal{O}(\log n)$ bits, the lemma follows. □

Definition 12. *Given two online BST data structures $\mathcal{A}_0$ and $\mathcal{A}_1$,* Combo-BST$(\mathcal{A}_0, \mathcal{A}_1, f(n)) =$ OneFinger-BST(Combo-MFBST$(\mathcal{A}_0, \mathcal{A}_1, f(n))$).

5 Analysis

Theorem 13. *Given two online BST data structures $\mathcal{A}_0$ and $\mathcal{A}_1$, let $\mathcal{T}_{X'}(\mathcal{A}_0, T, X)$ and $\mathcal{T}_{X'}(\mathcal{A}_1, T, X)$ be well-behaved amortized upper bounds with overhead $f(n) \geq n$ on the running time of $\mathcal{A}_0$ and $\mathcal{A}_1$, respectively, on a contiguous subsequence X' of X for any online access sequence X and initial tree T Then there exists an online BST data structure,* Combo-BST $=$ Combo-BST$(\mathcal{A}_0, \mathcal{A}_1, f(n))$ *such that*

$$\mathcal{T}_{X'}(\textsf{Combo-BST}, T, X) = \mathcal{O}(\min(\mathcal{T}_{X'}(\mathcal{A}_0, T, X), \mathcal{T}_{X'}(\mathcal{A}_1, T, X)) + f(n)).$$

If $\mathcal{A}_0$ and $\mathcal{A}_1$ are real-world BST data structures, so is Combo-BST.

Proof. Let $\mathcal{A}_{\min} = \mathcal{A}_0$ if $\mathcal{T}_{X'}(\mathcal{A}_0, T, X) \leq \mathcal{T}_{X'}(\mathcal{A}_1, T, X)$, and $\mathcal{A}_{\min} = \mathcal{A}_1$ otherwise. Let $X'' = X'_1 \cdot \ldots \cdot X'_k$ be the subsequence of X' executed by $\mathcal{A}_{\min}$. If the Combo-MFBST terminates after at most $2k+1$ rounds (k of them performed by $\mathcal{A}_{\min}$), then taking into account the TSB traversal at every round which takes $\mathcal{O}(n) = \mathcal{O}(f(n))$ time by Lemma 10 we have

$$\mathcal{T}_{X'}(\textsf{Combo-MFBST}, T, X) = \mathcal{O}(k \cdot f(n) + k \cdot n). \tag{1}$$

Now we need to bound k. Each round but the last one runs for $C_3 \cdot f(n)$ steps exactly, and in particular, $\mathcal{T}(\mathcal{A}_{\min}, T, X'_j) \geq C_3 \cdot f(n)$ for all $j < k$, that is, it might need more steps to complete the last access of X'_j. Summing over all j, we get $C_3(k-1)f(n) \leq \sum_{j=1}^{k-1} \mathcal{T}(\mathcal{A}_{\min}, T, X'_j) \leq C_0 \sum_{j=1}^{k-1} \mathcal{T}_{X'_j}(\mathcal{A}_{\min}, T, X) + C_1(k-1)f(n) \leq C_0 \mathcal{T}_{X'}(\mathcal{A}_{\min}, T, X) + C_1(k-1)f(n)$ by well-behavedness, the definition of $\mathcal{T}$ and the fact that X'_js are disjoint subsets of X'. Therefore, setting $C_3 > C_1$

yields $k - 1 \leq \mathcal{O}(\mathcal{T}_{X'}(\mathcal{A}_{\min}, T, X)/f(n))$. Combining with Equation 1, we obtain the desired bound for Combo-MFBST. By Lemma 11, Combo-MFBST is a real-world Multifinger-BST data structure. Applying OneFinger-BST (Theorem 5) yields our result. □

References

1. Brian Allen and J. Ian Munro. Self-organizing binary search trees. *Journal of the ACM*, 25(4):526–535, 1978.
2. Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
3. Prosenjit Bose, Sébastien Collette, Rolf Fagerberg, and Stefan Langerman. De-amortizing binary search trees. In *Proceedings of the 39th International Colloquium Automata, Languages, and Programming (ICALP)*, pages 121–132, 2012.
4. Prosenjit Bose, Karim Douïeb, Vida Dujmovic, and John Howat. Layered working-set trees. *Algorithmica*, 63(1-2):476–489, 2012.
5. Prosenjit Bose, Karim Douïeb, John Iacono, and Stefan Langerman. The power and limitations of static binary search trees with lazy finger. arXiv:1304.6897, 2013.
6. Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.
7. Richard Cole, Bud Mishra, Jeanette P. 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.
8. Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality — almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.
9. Erik D. Demaine, John Iacono, Stefan Langerman, and Özgür Özkan. Combining binary search trees. arXiv:1304.7604, 2013.
10. Erik D. Demaine, Stefan Langerman, and Eric Price. Confluently persistent tries for efficient version control. *Algorithmica*, 57(3):462–483, 2010.
11. Jonathan Derryberry. *Adaptive Binary Search Tree.* PhD thesis, CMU, 2009.
12. Jonathan Derryberry and Daniel Dominic Sleator. Skip-splay: Toward achieving the unified bound in the BST model. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS)*, pages 194–205, 2009.
13. John Iacono. Improved upper bounds for pairing heaps. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 32–45, 2000.
14. 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 (SODA)*, pages 516–522, 2001.
15. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
16. Daniel Dominic Sleator, Robert Endre Tarjan, and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC)*, pages 122–135, 1986.
17. 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 (SODA)*, pages 374–383, 2006.