The Geometry of Tree-Based Sorting

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA

Magdalen Dobson
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We study the connections between sorting and the binary search tree (BST) model, with an aim towards showing that the fields are connected more deeply than is currently appreciated. While any BST can be used to sort by inserting the keys one-by-one, this is a very limited relationship and importantly says nothing about parallel sorting. We show what we believe to be the first formal relationship between the BST model and sorting. Namely, we show that a large class of sorting algorithms, which includes mergesort, quicksort, insertion sort, and almost every instance-optimal sorting algorithm, are equivalent in cost to offline BST algorithms. Our main theoretical tool is the geometric interpretation of the BST model introduced by Demaine et al. [18], which finds an equivalence between searches on a BST and point sets in the plane satisfying a certain property. To give an example of the utility of our approach, we introduce the log-interleave bound, a measure of the information-theoretic complexity of a permutation π, which is within a $\lg \lg n$ multiplicative factor of a known lower bound in the BST model; we also devise a parallel sorting algorithm with polylogarithmic span that sorts a permutation π using comparisons proportional to its log-interleave bound. Our aforementioned result on sorting and offline BST algorithms can be used to show existence of an offline BST algorithm whose cost is within a constant factor of the log-interleave bound of any permutation π.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases binary search trees, sorting, dynamic optimality, parallelism

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.26

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2110.11836`

Funding This work was supported by the National Science Foundation grants CCF-1901381, CCF-1910030, CCF-1919223, DGE1745016, and DGE2140739.

Acknowledgements We thank Kanat Tangwongsan for helpful discussions. We thank the anonymous reviewers for their useful comments.

1 Introduction

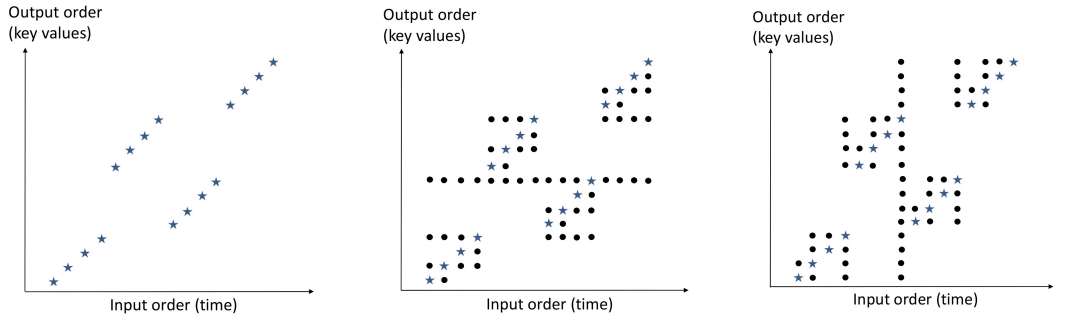
Comparison-based sorting and searching on a BST are among the most elementary, important, and well-studied algorithmic topics in all of theoretical computer science. It has long been observed that they are closely related: both enjoy better performance on sequences that are information-theoretically simpler, such as reversals of sorted lists, sequences with long runs of consecutive keys, or sequences composed from simple shuffles of sorted lists. Indeed, their respective searches for instance optimality have yielded the independent discovery of almost identical results [34]. Despite the extensive number of similar results throughout the literature, there is comparatively very little known about the *formal* relationship between sorting and the binary search tree (BST) model. In this paper, we present what we believe to be the first formal relation between the sorting cost model and the BST model. Our result shows that a large class of sorting algorithms, which includes mergesort, quicksort, insertion

sort, and most adaptive sorting algorithms, are equivalent in cost to offline algorithms in the BST model. As this class is large and contains many well-studied algorithms, we find it interesting that we are able to show any kind of new and nontrivial theoretical result based on this characterization.

Binary Search Trees. The binary search tree is a fundamental data structure that stores an ordered universe of keys in a dynamic tree. A search for a key begins with a pointer to the root of the tree and at each step performs one of two unit-cost actions: move the pointer to a parent or child node, or perform a rotation. Rotations are key to understanding the power of the model, because they allow frequently queried elements to be kept close to the root of the tree as well as exploiting other kinds of order in the sequence of queried keys. The power of rotations is related to the concept of *instance optimality*, a general term which refers to the fact that an algorithm can enjoy improvements on its worst-case complexity on well-defined sets of "easy" inputs. Many BST algorithms perform better than their worst-case complexity (i.e. $\log n$ per operation) on various kinds of input [37, 19, 18, 3]. Beyond these specific improvements, the hope for a more general kind of instance optimality is crisply expressed by the *dynamic optimality conjecture* of Sleator and Tarjan [35], which states that there exists a binary search tree whose performance on any online sequence of searches is constant factor competitive with the best offline algorithm for that sequence. The dynamic optimality conjecture remains open. Another equally important open question is whether there is an offline efficient algorithm for calculating even an approximately optimal number of rotations for a given input sequence. These problems have been the subject of extensive work both in the past [38, 16, 15, 19, 21, 8] and at the present moment [26, 13, 7, 25, 23].

Sorting. Similarly to the BST model, where rotations are used to achieve instance optimality for particular classes of inputs, in comparison-based sorting an *adaptive sorting algorithm* performs fewer comparisons when the input is "closer" to sorted by some measure. In this field a *measure of disorder* for a list L is paired with an algorithm which is *optimal* for this measure. Here, optimal roughly means that sorting L only requires the number of comparisons needed to distinguish it from all other lists which are more presorted than L [34]. An accompanying notion is that a measure of disorder may be *superior* (inferior) to another measure – that is, always requires fewer comparisons for any given permutation. Mannila first formalized these ideas [31]. After this, many researchers devised new measures of disorder and corresponding optimal algorithms [17, 22, 24, 28, 29, 27, 33, 34]; furthermore, there was also interest in work-optimal parallel versions of optimal sorting algorithms [11, 30, 14]. It remains an open problem whether there exists a measure which is provably superior to any possible measure of disorder.

Arborally Satisfied Point Sets and Sorting. One of our most important tools in connecting BSTs and sorting is the geometric interpretation of BSTs [18, 20]. In this interpretation, an access sequence of n keys is represented as an $n \times n$ grid with time order (input order) on one axis (here the x axis) and key order (output order) on the other axis. Points are added to the grid to account for all keys that must be visited when searching or inserting the keys one at a time from left to right. Demaine et al. [18], and Derryberry, Sleator, and Wang [20] show that for any BST algorithm, the accesses plotted in the plane must satisfy the property that for every pair of points p, q (both original and added points), there is a monotonic path (e.g. up and right) from p to q consisting of horizontal and vertical segments with a point at



(a) The input sequence.

(b) Sequence arborally satisfied using quicksort.

(c) Sequence arborally satisfied using mergesort.

Figure 1 On the left, the input sequence plotted in the plane. In the middle, the input sequence arborally satisfied using accesses corresponding to quicksort. On the right, the input sequence arborally satisfied using accesses corresponding to mergesort.

each corner[1]. Demaine et al. refer to such a set of points as being *arborally satisfied*, and show that any such set of size m implies the sequence of n keys can be searched or inserted in cost $O(m)$ in the BST model.

We observe that the geometric approach is also useful for sorting, since unlike the BST model, it does not directly enforce an order of insertion. As some evidence of the utility of the geometric approach, consider the following two algorithms that can be used to arborally satisfy a set of accesses. The first algorithm starts by choosing a random point, then adding accesses to that point across its entire row of the point set. Then, it recurses above and below the row, and in each partition it picks a random point and adds new points to all locations in its row for which there is a key in the partition. It is not hard to verify the points added in this way are arborally satisfied: any point p can get to a point q by going up (or down) to the row that separated them, then across to column of q and up (down) to q. Second, consider another algorithm that adds points across the middle column, and then for the left and right, add points along their middle columns for all points in those halves. Recursing to the base case again gives an arborally satisfied set. See Figure 1 for an example of both of these algorithms. The attentive reader may have noticed that the accesses added in the first algorithm correspond to the comparisons made by the quicksort algorithm, and the accesses added in the second algorithm correspond to comparisons made by the mergesort algorithm. We will extend these ideas to more interesting algorithms in this paper.

We note that in addition to being of significant theoretical interest, taking advantage of locality in key sequences is widely practical for both search trees and sorting. Sleator and Tarjan won the ACM Kannelakis Theory and Practice award for their work on splay trees and its applications for reducing key search time in several widely used applications. Adaptive sorting algorithms are widely adopted in practice, including *timsort*, which is implemented as built-in libraries for Python, Java, Swift, and Rust, among other languages [2].

1.1 Our Results

In this paper we present specific results relating sorting and BSTs using arborally satisfied sets. Our first result is an explicit relation between the cost models of a broad set of sorting algorithms and BST algorithms. The specific set of algorithms, which we refer to as *tree-based*

[1] The papers have their own preferred definition, but the definitions are all equivalent.

sorting algorithms and which are formally defined in Section 3, are divided into two classes: *BST mergesorts* are sorting algorithms based on recursive merges of the input, where the keys being merged are stored in binary search trees; *BT partition sorts* are sorting algorithms based on recursive partitions of the input based on key ordering, where the keys are stored in a binary tree. In Section 3, we show that these two types of algorithms are "dual" to each other in the following way: a BST mergesort $\mathcal{A}$ sorting permutation π with cost $\mathcal{A}(\pi)$ implies the existence of binary tree (BT) partition sort $\mathcal{B}$ sorting π^{-1} with cost $\mathcal{A}(\pi)$, and vice versa.

The category of tree-based sorting algorithms is large. Both quicksort and mergesort on lists fall into the category of tree-based sorting algorithms, as lists are simply a special case of trees. Sorting by insertion into a BST is also a BST mergesort, since the merges can be carried out in any order. McIlroy's adaptive sorting algorithms – namely, insertion sort with exponential search and mergesort with exponential search – are BST mergesorts [32]. These algorithms were influential in the development of timsort [2], a mergesort algorithm that breaks the input into runs of increasing or decreasing keys and merges them based on certain ordering criteria; since it is also a mergesort on lists, timsort is a BST merge. In their capstone paper on adaptive sorting, Petersson and Moffat cover the three most powerful known adaptive sorting algorithms – local insertion sort, historical insertion sort, and regional insertion sort. Local insertion sort is a BST mergesort as it inserts into a BST with a single additional pointer, which can be converted to the BST model with constant overhead [13]. Historical and regional insertion sort are not BST mergesorts as presented by the authors, but independently discovered data structures would yield BST mergesorts with the same bounds [35, 21].

Theorem 1 shows that for any tree-based sorting algorithm that sorts access sequence π using $\mathcal{A}(\pi)$ accesses, there exists an offline algorithm in the BST model which searches for each key in π using $O(\mathcal{A}(\pi))$ accesses. The proof of Theorem 1 is subtle and nontrivial and requires several new insights relating to the geometric interpretation of the BST model. In the following statement, $\mathrm{OPT}_{\mathrm{BST}}(\pi)$ refers to the cost of the best offline algorithm.

▶ **Theorem 1.** *Let $\mathcal{A}$ be a tree-based sorting algorithm which sorts permutation π using $\mathcal{A}(\pi)$ accesses. Then $OPT_{BST}(\pi) \in O(\mathcal{A}(\pi))$.*

As some evidence for the utility of our approach, we introduce the log-interleave bound, a measure of the information-theoretic complexity of a permutation π. The log-interleave bound is an upper bound on the number of bits needed to encode π; it can also be understood from an algorithmic perspective as a mergesort with a more efficient merge step. Our main results on the log-interleave bound illustrate the connections between sorting and the BST model. In the statements of the following results, we use the notation $\mathrm{LIB}(\pi)$ to refer to the log-interleave bound of a permutation π. This will be defined formally in Section 4.

The first result is a proof that the log-interleave bound is within a $\lg \lg n$ multiplicative factor of the optimal offline BST algorithm on any permutation. Somewhat similarly to Demaine et al.'s proof of the closeness to optimality of tango trees [19], our proof shows closeness to optimality by comparing the log-interleave bound with Wilber's interleave bound [38], a lower bound in the BST model.

▶ **Theorem 17.** *For any permutation π, $IB(\pi) \leq LIB(\pi) \in O(\lg \lg n \; IB(\pi))$.*

Next, we show that there is a work optimal *parallel* sorting algorithm related to the log-interleave bound. The next result is a parallel mergesort featuring a merge step which combines recent work on parallel split and join of BSTs [4] with a BST from [9] and an analysis which shows that with this new merge step, the mergesort sorts a sequence π in

$O(\text{LIB}(\pi))$ work. While the span of this algorithm is greater than the span of a typical parallel sort and indeed may be open to improvement, all existing parallel sorting algorithms present guarantees only for very weak measures of disorder [14, 30].

▶ **Theorem 21.** *There exists a parallel mergesort which for any permutation π performs $O(LIB(\pi))$ work with polylogarithmic span.*

Finally, a corollary of Theorem 6 shows that there is an offline BST algorithm that incurs cost $O(\text{LIB}(\pi))$, and thus that the log-interleave bound is an upper bound in the BST model.

▶ **Corollary 22.** *There exists an offline BST algorithm $\mathcal{A}$ such that $\mathcal{A}(\pi) = O(LIB(\pi))$.*

Model of Computation. Our results for the parallel algorithms are given for the binary-fork-join model [5]. In this model a process can fork two child processes, which work in parallel and when both complete, the parent process continues. Costs are measured in terms of the work (total number of instructions across all processes) and span (longest dependence path among processes). Any algorithm in the binary forking model with W work and S span can be implemented on a CRCW PRAM with P processors in $O(W/P + S)$ time with high probability [1, 6], so the results here are also valid on the PRAM, maintaining work efficiency.

1.2 Related Work

Upper and Lower Bounds in the BST Model. The pursuit of dynamic optimality led to a string of work in both upper and lower bounds on the cost of a sequence of searches on a BST. Three important upper bounds in the literature are the *dynamic finger bound* [35, 16, 15, 13, 8, 23], the *working set bound* [35], and the *unified bound* [3, 21], which respectively state that accessing an element is fast if its key is close to the key of the previous search, if its key has been searched recently, and a combination of the two. There has also been significant work in lower bounding the cost of an access sequence in the BST model. Two such lower bounds, the *interleave bound* and the *funnel bound*, were introduced by Wilber in [38]; a recent work by Lecomte and Weinstein [26] affirmatively settled the 30-year open question of whether the funnel bound was tighter than the interleave bound, proving a $\lg \lg n$ multiplicative separation in some cases. Another lower bound, the *rectangle bound*, was introduced by Demaine et al. in [18].

Progress on Dynamic Optimality. The BST which comes closest to dynamic optimality is the *tango tree* of Demaine et al. [19], which has a competitive ratio of $O(\lg \lg n)$ with respect to the best offline algorithm. Wilber's interleave bound was vital in the analysis of the competitive ratio, since the authors showed that on any access sequence x, the tango tree uses $O(\lg \lg n \ \text{IB}(x))$ accesses, where $\text{IB}(x)$ represents the interleave bound of the sequence. In [37], Wang et al. introduce the multi-splay tree, a BST which achieves $O(\lg \lg n)$ optimality with better worst-case guarantees than the tango tree. Prominent candidates for a dynamically optimal algorithm include the splay tree, which was presented by Sleator and Tarjan at the same time as the dynamic optimality conjecture [35], and the Greedy algorithm presented in Demaine et al.'s geometric interpretation of the BST model [18].

Other Data Structures and the BST model. The *min-heap*, which stores a set of keys and supports inserting arbitrary elements and extracting and deleting the minimum element. Recently, Kozma and Saranurak show an explicit relation between the BST model and the

heap cost model [25], as well as proposing an analogue of the dynamic optimality conjecture for heaps. Specifically, they show that for every heapsort algorithm (that is, an algorithm which sorts a permutation π with cost $\mathcal{A}(\pi)$ by inserting its keys into a heap and repeatedly extracting the minimum element) corresponds to an insertion sort into a BST algorithm which incurs cost $\mathcal{A}(\pi)$ on the inverse permutation π^{-1}. Their insight came from relating the rotation operation in a BST to the link operation in a heap, which allowed them to relate the corresponding cost models.

Adaptive Sorting in Parallel. During the period of interest in the adaptive sorting model, researchers were also interested in work-optimal parallel sorting algorithms with polylogarithmic span. Unsurprisingly, such algorithms exist for practical measures Runs and Inv [11, 14]; one also exists for Osc, a generalization of Inv [30] that is still theoretically weak. To our knowledge there are no results on parallel sorting algorithms which are optimal with respect to any stronger measures.

2 Preliminaries

Terminology. Throughout this paper, we will use the terms **list**, **permutation**, and **access sequence** interchangeably to refer to some ordering of the keys $1, 2, \ldots, n$. The term access sequence is used in the literature on BSTs to denote a sequence of queries to a BST; unless otherwise stated, an access sequence is presumed not to contain repeated keys.

The Binary Search Tree Model. A binary tree (BT) is either a *leaf* or a *node* consisting of a left binary tree, a key and a right binary tree. A binary seach tree (BST) is a binary tree where the keys have a total order, and for each node in the tree all keys in its left subtree are less than its key, and all keys in its right tree are greater.

The following definition of the BST model is drawn from [35, 38, 19]. The model assumes an initial BST with keys $[1, 2, \ldots, n]$ and an access sequence $[x_1, x_2, \ldots, x_m]$ of searches, where $x_i \in \{1, 2, \ldots, n\}$. Each search starts at the root and at each node it visits, it may perform one of the following actions: (a) move to the right child, left child, or parent, or (b) perform a rotation of the node and its parent. Each of these actions has unit cost and the search must visit its specified key. We refer to an algorithm that decides on what actions to perform for each search as a **BST algorithm**. A BST algorithm may be offline – meaning it can see the entire sequence of queries ahead of time – or online, meaning that queries are revealed one at a time.

Wilber's Interleave Bound. Wilber's interleave bound is a lower bound on the cost of accessing any sequence in the BST model. Given an access sequence π consisting of the keys $x_1, x_2, \ldots, x_n$, fix a static binary tree P (meaning it will never be rotated) with the keys of π at the leaves in the order they appear in π. Calculate the interleave bound of π as follows: query the keys in π in sorted order. For each vertex v_j, label each element i of the sequence with R or L, depending on whether accessing i in P goes through the right or the left subtree of v_j, respectively (if i is in neither subtree, give it no label). The interleave bound of v_j, denoted $\mathrm{IB}(v_j)$, is the number of switches between R and L in the labels if the keys are queried in sorted order. The interleave bound of the entire access sequence π is calculated by summing over the interleave bounds of each vertex, so $\mathrm{IB}(\pi) = \sum_{v \in P} \mathrm{IB}(v_i)$. As the lower bound holds for an arbitrary tree P, the interleave bound of the sequence is usually understood to refer to the maximum over all static trees. See Figure 3 for an example calculation.

Arborally Satisfied Sets. In [21], Derryberry et al. formalize a connection between binary search trees and points in the plane satisfying a certain property. An access sequence can be plotted in the plane where one axis represents key values and the other axis represents the ordering of the search sequence (that is, time). In the context of sorting, these axes can also be referred to as input order and output order. In this work, we use the horizontal axis for time and the vertical access for keyspace. See Figure 1 for an example of an arborally satisfied set.

In addition to plotting the search sequence on the plane, one can also plot the key values of the nodes which a BST algorithm accesses (for search or rotations) while searching for a node. When searching for an element x_i which is inserted at time i, the values of the nodes in the search path are plotted on the same vertical. Demaine et al. [18] proved that such a plot satisfies the following property:

▶ **Definition 2.** *Given a set P of points in the plane, P is **arborally satisfied** if for every two points $x, y \in P$ that are not on the same vertical or horizontal, the rectangle defined by x and y contains at least one point in addition to x and y.*

As mentioned in Section 1, a useful equivalent definition of arboral satisfaction is that there must be a monotonic path (e.g. consisting only of moves up and to the right) between x and y with a point at every corner. Note that any valid search on a BST will only touch nodes in a subtree τ_i of tree T, where τ_i includes the root of T. We sometimes refer to such a subtree as a **top tree** of T.

Demaine et al. show that a BST can be used to arborally satisfy an access sequence plotted in the plane. However, one can also use an algorithm that directly places points in the plane rather than using a BST.

▶ **Definition 3.** *Given a set of points in the plane corresponding to an access sequence π, an **offline arboral satisfaction algorithm** adds points to the plane to make an arborally satisfied set. An **online arboral satisfaction algorithm** also adds points in the plane to form an arborally satisfied set, but accesses are revealed one by one in input order and the algorithm must produce an arborally satisfied set at each time.*

Demaine et al. show that a BST algorithm is equivalent to an arboral satisfaction algorithm, but they also show a more surprising result: an arboral satisfaction algorithm is equivalent to a BST algorithm. Specifically, they show that an offline (online) arboral satisfaction algorithm requiring $f(\pi)$ accesses to arborally satisfy a search sequence π can be transformed to an offline (online) BST algorithm requiring $O(f(\pi))$ accesses to search for the elements of a sequence π.

3 Tree-based Sorting

Towards the goal of unifying the BST model and sorting, we ask the following question: when can the costs of a sorting algorithm be related to the costs of a BST algorithm? Clearly not every comparison-based sorting algorithm should be relatable to the BST model: as mentioned in Section 1, for example, an algorithm that guesses and checks could take $O(n)$ steps for an arbitrary permutation. Our investigation is therefore limited to sorting over binary trees, and in particular it considers a class of mergesort and partition sort (related to quicksort) algorithms on binary trees.

3.1 Mergesort

We first consider mergesorts in which the sequences to be merged are represented as BSTs. "Mergesort" is interpreted here as merging based on any split of the input sequence, not just splits into equally-sized parts; thus insertion sort using a sequence of insertions into a BST is a special case of mergesort. The cost of these algorithms is measured in terms of the number of accesses to the tree required during the mergesort, which will always be at least as great as the number of comparisons. We capture the idea of a BST mergesort more formally before giving the theorem statement.

A merge will interleave contiguous subsequences from its two inputs. We refer to each of these subsequences as *blocks* and we refer to the ends of each block as *block boundaries*. The block boundaries need to be accessed to even verify that the merge is correct. The following defines a merge that examines some top part of two trees to generate its output.

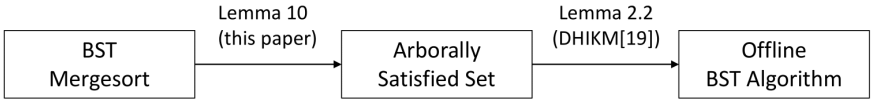
▶ **Definition 4.** *A* ***BST merge*** *takes two BSTs T_A and T_B, and for some top trees τ_a of T_A and τ_b of T_B, returns a BST T such that for some top tree τ of T, $\tau = \tau_a \cup \tau_b$, the subtrees of τ correspond to unchanged subtrees of τ_a and τ_b, and τ contains the block boundaries. The* number of accesses *used by the merge is $|\tau|$.*

▶ **Definition 5.** *A* ***BST mergesort*** *recursively splits the input sequence into two parts each of size at least 1, sorts each part with a BST mergesort, and executes a BST merge on the results. A mergesort on an input of size 1 returns its input. The* number of accesses *used by the mergesort is the sum of accesses across all merges.*

This leads to the main theorem.

▶ **Theorem 6.** *Let $\mathcal{A}$ be a BST mergesort algorithm which sorts permutation π using $\mathcal{A}(\pi)$ accesses. Then $OPT_{BST}(\pi) \in O(\mathcal{A}(\pi))$.*

Theorem 6 is proved by showing that for every BST mergesort algorithm, there is an offline BST algorithm that incurs the same cost as the BST mergesort within a constant factor. Our proof relies on the geometric interpretation of the BST model – instead of directly transforming a BST mergesort into an offline BST algorithm we use arborally satisfied sets as an intermediary. The key ingredient is a transformation of a BST merge algorithm to an offline arborally satisfied set algorithm, which is equivalent to an offline BST algorithm by Demaine et al.'s theorem [18]. The following graphic illustrates the chain of dependencies.

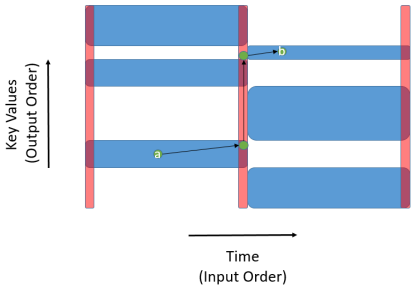


The main idea behind going from the mergesort to an arborally satisfied set is to transform a merge algorithm $\mathcal{M}$ into an algorithm that "merges" two arborally satisfied sets by concatenating them along the time axis (in this paper the x axis) and resolving any unsatisfied rectangles between the two sets, thus producing an arborally satisfied set. This "arboral" mergesort algorithm would by definition be an offline arborally satisfied set algorithm. Ideally this arboral merge would use the same number of accesses as a corresponding BST merge $\mathcal{M}$. The key idea behind our algorithm is to use the keys accessed during the tree merge to arborally satisfy the sets on the two sides, as well as add points to make it easier to satisfy the condition on future merges. In particular, the keys are added in

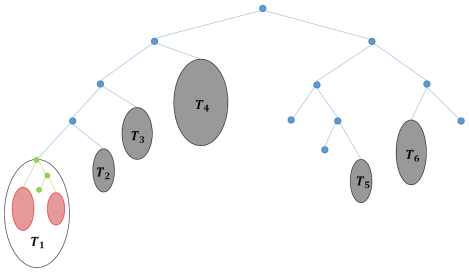
Algorithm 1 arboralMerge$(A, B, \mathcal{M})$
The arboral satisfaction algorithm; also illustrated in Figure 2a.

Input: Two arborally satisfied sets A, B; BST merge algorithm $\mathcal{M}$.
Output: An arborally satisfied set C consisting of the concatenation of A and B as well as additional accesses needed to arborally satisfy the concatenation.

```
1 if A == ∅ then return B ;
2 if B == ∅ then return A ;
3 C ← concatenate A and B on the time axis ;
4 S ← set of keys accessed when merging keys in A and B using M ;
5 access each key in S in the first, middle (rightmost column of A), and end columns of C ;
6 return C ;
```



(a) An illustration of an arboral merge.



(b) A BST separated into a top tree and auxiliary trees.

Figure 2 On the right, an illustration of the merging algorithm shown in Algorithm 1. The blue squares represent members of the two sets being merged, while the red columns (referred to as C_L, C_M, C_R in the proof of Theorem 6) illustrate the additional accesses necessary for the merge. A path drawn from $a \in A$ to $b \in B$ illustrates how the accesses in the middle column ensure the set is arborally satisfied. On the right, a BST with a top tree shown in blue and auxiliary trees T_1 through T_6, where the recursive structure is shown in T_1.

three columns: the leftmost column of the left set, the rightmost column of the right set, and a middle column – we will use the rightmost of the left set, although the leftmost of the right set would also work. Roughly, the accesses are placed down the middle to resolve unsatisfied rectangles, and they are placed down the leftmost and rightmost columns to restore invariants that are useful for future merges. These accesses must by definition form top trees of the two trees being merged. Algorithm 1 shows the merging routine, which is used as a sub-step in an arborally satisfied set algorithm which recursively splits the input set to singletons and then uses Algorithm 1 to merge sub-parts of the input; see Figure 2a for an illustration of the merge step. Since we are adding a constant number of points per access in the mergesort, the total points added is proportional to the cost of the mergesort, which in turn implies an offline BST algorithm with the same cost as the original BST mergesort.

The most difficult part of Theorem 6 is proving correctness of the arboral mergesort – that is, that each execution of the arboral merge routine produces an arborally satisfied set. This will require some more background on arborally satisfied sets.

▶ **Definition 7.** *A **treap** over a set of pairs S is a BST over the first coordinate of each $s \in S$ and a min-heap over the second coordinate. Ties over the second coordinate are permitted and may be broken arbitrarily.*

A treap with ties on the priorities can also be expressed as a **multi-treap** (for multi-node treap), where ties are stored in a **multi-node** that may have more than two children. Child relations must obey an underlying BST structure on the nodes stored within a multi-node; hence a multi-node may have at most one more child than the number of keys in the multi-node. A key component of our proof is that we will define a multi-treap with respect to each side of an arborally satisfied set and then relate these to the BST the mergesort will generate.

▶ **Definition 8.** *Given an arborally satisfied set A, let the left (right) priority be the distance from the left (right) boundary of the first point in the row (closer has higher priority). The* left (right) multi-treap *of A is the multi-treap defined by the (row, priority) pairs.*

Since there can be many points in any given column of an arborally satisfied set, multi-nodes of the multi-treaps can have more than two children.[2]

▶ **Definition 9.** *Given a BST T and an arborally satisfied set A with left (right) multi-treap H_A, T is* left (right) congruent with *A if there is some valid BST structure on H_A (forming a binary subtree within each multi-node) such that the tree structure on H_A is equal to T. T is* doubly *congruent with A if it is both left and right congruent.*

Note that many BSTs can be left (equivalently right) congruent with the same arborally satisfied set A due to equal priorities. Also many arborally satisfied sets can be left (right) congruent with the same BST T. It may seem unlikely that a BST is doubly congruent with a arborally satisfied set, but in our construction we will maintain double congruence, and in particular we will show that the point sets created by Algorithm 1 are doubly congruent to the corresponding tree.

The following observation will be useful for the proof of Theorem 6. It follows from a similar argument to Lemma 2.1 of Demaine et al. [18].

▶ **Observation 10.** *Consider an arborally satisfied set A that is double-congruent with a tree T. Then for any top tree τ of T, if the keys in τ are accessed along either the left or right column of A, or one past the left or right column, the resulting set of points is arborally satisfied.*

Proof. Begin with the case where the keys of τ are accessed one past the leftmost or rightmost column of A. Assume for the sake of contradiction that there exists an unsatisfied rectangle – that is, a rectangle with two points at its corners and no points contained within it – between access $a \in A$ and access $b \in \tau$. Consider the least common ancestor c of a and b, whose key value must be between those of a and b. Since by our assumption, there are no keys accessed between the rectangle defined by a and b, this contradicts the fact that τ is a continuous subtree of T, since c must be accessed in τ to reach b.

This leaves the case where accesses to τ are instead placed on the leftmost or rightmost column of A – that is, in addition to accesses that were already there. Consider an arbitrary access $a \in A$ and any access $b \in \tau$. If the accesses in τ are placed on a new column past the rightmost (leftmost) column of A, there is a monotonic path from a to b with accesses at every corner. If the accesses in τ are imposed on the rightmost (leftmost) column of A instead, the same accesses still form a monotonic path, since this only causes the elimination of one right (left) move. ◀

[2] Demaine et al. [18] define a similar notion when proving that for any arborally satisfied set there is a BST execution with equivalent cost, but only with respect to one side, and only when sweeping column by column.

We now prove correctness of the arboral mergesort algorithm.

▶ **Lemma 11.** *The arboral mergesort algorithm is correct: that is, it returns an arborally satisfied set.*

Proof. We will use the following inductive hypothesis on the arboral merge algorithm (Algorithm 1) to show correctness: Algorithm 1 returns an arborally satisfied set which is double-congruent to the tree T returned by the corresponding BST mergesort algorithm.

Base Case. When the set is just a single point, both arboral satisfaction and double congruence follow trivially.

Inductive Step. The inductive step is broken into several claims, and some new notation is called for. The two arborally satisfied sets being merged are A and B, and by the inductive hypothesis are both arborally satisfied and double-congruent to trees T_A and T_B, respectively. The additional accesses specified by the mergesort are added to three columns. Let C_L, C_R, and C_M denote the set of points which the arboral merge adds along the left, right, and middle columns respectively; see Figure 2a for an illustration. It will also be useful to denote the subset of a C_i ($i \in \{L, R, M\}$) consisting only of accesses to keys in A or B. These subsets are denoted by $C_i(A)$ or $C_i(B)$.

The merge will break A and B into contiguous blocks that are interleaved in key order. As assumed in the model, the block boundaries must be accessed by the merge, and therefore included in the C_i. In general the C_i will include other points as well. The inductive step has to show both that the resulting set is arborally satisfied and is doubly congruent to the merged tree T.

Arboral Satisfaction. Points within A or B are satisfied by the inductive hypothesis. Points both in C_L, C_R, C_M are satisfied by the fact that they access precisely the same keys. This leaves the two more interesting cases: (1) pairs of points one from the previously exiting points (in A or B) and one from the new boundaries C_L, C_R, C_M, and (2) pairs of points one from A and one for B. For the first case, A is double-congruent to tree T_A (by the inductive hypothesis), and $C_i(A)$ is a top tree of A (by construction), so we can apply Observation 10 for points in A and $C_i(A)$. Now since the boundaries of each block of A must be in $C_i(A)$, we can get from a point in A to a point in $C_i(B)$ using a monotonic path by going to a boundary point in $C_i(A)$ and then up or down the column to the point in $C_i(B)$. Symmetrically points in B can get to points in $C_i(B)$ and $C_i(A)$ by a monotonic path. For case (2) consider any point $a \in A$ and point $b \in B$. There is a monotonic path between a and b by composing the monotonic path between a and a boundary point in p_a in $C_m(A)$, between b and a boundary point p_b in $C_m(B)$, and between p_a and p_b, which are in the same column; see Figure 2a.

Double congruency to T. We will show that the set AB returned by the arboral merge algorithm is right congruent to T. Left congruency is true by symmetry. The keys in C_R (those accessed by the merge) correspond to a top tree τ of T. Since all keys in the column C_R have the same highest priority we can organize the root multi-node to match the structure of τ making those nodes congruent. Now consider the subtrees not in τ. They properly are lower in the tree and have equal or lower priority (only equal if they happen to be in the last row). Furthermore the subtrees are separated by keys in τ. Each such subtree either comes completely from T_A or completely from T_B and have the same structure as before the merge (they were not touched by the merge). Furthermore the points from T_A (T_B) only appear in

A (B). This implies the relative priorities have not change for those points when merging into AB. By induction, the trees T_A (T_B) were congruent to A (B) before the merge so the subtrees were congruent and remain congruent after the merge (neither the relative priorities nor tree structure have changed). This implies the whole tree T is congruent with AB. ◀

The proof of Theorem 6 now follows easily.

Proof of Theorem 6. The theorem follows once the cost of the arboral mergesort is shown to be $O(\mathcal{A}(\pi))$. Since the cost of $\mathcal{A}$ is dominated by the cost of each merge execution, and that cost is at most multiplied by six in each call to the arboral merge, the cost of the arboral mergesort is at most $6\mathcal{A}(\pi)$. When the arboral mergesort is transformed into an offline BST algorithm, the cost remains the same for a total cost of $O(\mathcal{A}(\pi))$. ◀

3.2 Partition Sort

We now consider a class of sorting algorithms motivated by quicksort, which we refer to as partition sorts. As in the case of mergesort, we limit ourselves to working with binary trees. However, in this case the trees are not ordered by key, but instead are ordered by input order. The algorithm is like quicksort in that it picks a pivot, partitions the keys on the pivot and recurses. Since we are interested in lower bounds (i.e. showing the cost of partition sort is at least as great as optimal BSTs), we can assume an oracle picks the perfect pivot (e.g., the median). As with mergesort, to achieve better than trivial $O(n \log n)$ bounds it is important that the partition need not visit the whole tree it is partitioning, but rather just some top tree. This allows for sending whole subsequences to the lesser or greater/equal side without visiting all nodes. More precisely here are the definitions of partition and partition sort.

▶ **Definition 12.** *A **BT partition** takes a BT T and for some top tree τ of T returns two BTs T_A, T_B with distinct keys such that for some top trees τ_a of T_A and τ_b of T_B, $\tau = \tau_a \cup \tau_b$, the other subtrees of T appear in either T_A or T_B unchanged, and τ contains the block boundaries of the partitioned output. Furthermore the preordering of the keys in T_A or T_B are a subsequence of the preordering in T (i.e. left-to-right ordering). We assume both partitions are non-empty. The* number of accesses *used by the partition is $|\tau|$.*

▶ **Definition 13.** *A **BT partition sort** on a BT tree T (1) partitions T into T_a and T_b such that for some key k all keys in T_a are less than k and all keys in T_b are greater or equal to k, (2) recurses on each partition, and (3) returns the left and right results appended. The recursion terminates when the tree is of size one. The number of accesses used by the partition sort is the sum of accesses across all partitions.*

Our goal is to show the following theorem, which has the same form as the result for mergesort.

▶ **Theorem 14.** *Let $\mathcal{A}$ be a BT partition sort algorithm that sorts permutation π using $\mathcal{A}(\pi)$ accesses, and let $OPT_{BST}(\pi)$ be the optimal cost of querying π with a BST algorithm. Then $OPT_{BST}(\pi) \in O(\mathcal{A}(\pi))$.*

Our approach is to show a one-to-one correspondence between the tree-based merge and partition sorts.

▶ **Lemma 15.** *For any BT partition sort algorithm $\mathcal{A}$ that sorts permutation π using $\mathcal{A}(\pi)$ accesses, there is a BST mergesort sort algorithm $\mathcal{B}$ that sorts permutation π^{-1} using $\mathcal{B}(\pi^{-1}) = \mathcal{A}(\pi)$ accesses, and vice versa.*

Proof. The idea is to consider running BST mergesort backwards, while reversing the role of time and key order. Consider undoing a merge – i.e. taking the merged tree and partitioning back into its two inputs. Reversing the roles of time and keys, this is equivalent to a BT partition where keys are time order and the partitions is on the first time of the right partition. In particular the size of the top tree and therefore access cost is identical. This continues to be true on the recursive calls. In both cases the base case is of size one. Hence the total access costs of the two algorithms are identical, one applied to the inverse permutation of the other. ◀

Since the size of arborally satisfied sets are invariant under rotation by 90 degrees, reversing the role does not affect the size of the set. Since the proof of Theorem 6 first showed how to map a BST mergesort to an arborally satisfied set and this implied the same bound on a offline BST, this remains true if we rotate the input, arborally satisfy it in the same way, and generate a BST. Theorem 14 follows. Taken together, Theorem 6 and Theorem 14 show the statement in Theorem 1. Although the duality of mergesort and quicksort has been recognized before we are not aware of any formal correspondence such as the one given here.

4 The Log-Interleave Bound

The following two sections contain results that illustrate the utility of the approach shown in Theorem 1: namely, that results in the sorting cost model can directly translate to interesting results in the BST model. In this section we propose an information-theoretic bound on both the cost of accessing a sequence in the BST model and sorting a list in the comparison model. Theorem 17 shows that the log-interleave bound is within a $\lg \lg n$ multiplicative factor of a known lower bound in the BST model. In the next section, we show that there exists a BST mergesort algorithm that sorts any permutation π in $O(\mathrm{LIB}(\pi))$ comparisons, and thus combined with Theorem 6 shows the existence of an offline BST algorithm with the same costs in the BST model.

The log-interleave bound can be thought of as an algorithmic perspective on Wilber's interleave bound. Let P be the static tree with the keys of a permutation π at the bottom. Consider sorting π via mergesort: clearly, each non-leaf vertex of P denotes a merge. The interleave bound charges unit cost for each switch between the right and left subtree during a merge. Another way of looking at this cost is that every continuous run of accesses to the left subtree incurs unit cost. Thus, a mergesort with a merge step that incurred unit cost for each consecutive run – as opposed to the standard mergestep which charges for the size of each run – would sort π using $O(\mathrm{IB}(\pi))$ comparisons.

Lecomte and Weinstein [26] and Chalermsoook et al. [12] independently show that the merge step described above does not exist. However, it is possible to charge the logarithm of the size of each consecutive run, as shown by Brown and Tarjan in [10]. This idea leads us to using such a merge step as an information-theoretic bound, which applies to sorting (sequentially and in parallel), and the BST model. The log-interleave bound is formally defined below; note its similarity to the interleave bound.

▶ **Definition 16.** *Given an access sequence π, fix a static binary tree P with the keys of π at the leaves. For each vertex v_j, query the descendants of v_j in sorted order, then label each with R or L depending on whether it is in the left or right subtree of v_j. Let $S(v_j)$ represent the decomposition of this labeling into the smallest possible number of runs of consecutive accesses to L or R in v_j. Then $LIB(v_j) = \sum_{r_i \in S(v_j)} \lg(|r_i|+1)$ and $LIB(\pi) = \sum_{v \in P} LIB(v_i)$.*

See Figure 3 for an example calculation.

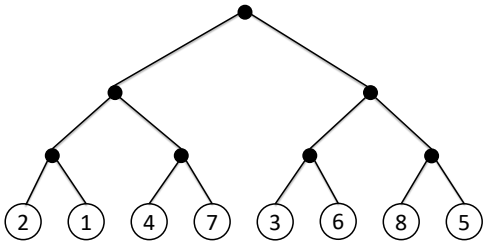


Figure 3 Consider accessing the keys 1-8 in order. For the vertex v_1 at the root of the tree shown here, the labeled access sequence is [L, L, L, R, L, R, R, R]. Since the access sequence switches between the left and right subtree three times, $\mathrm{IB}(v_1) = 3$. Similarly, $\mathrm{LIB}(v_1) = \lg 4 + \lg 2 + \lg 2 + \lg 4$ since the smallest possible decomposition of the labeled access sequence consists of [[L, L, L], [R], [L], [R, R, R]].

A natural question one might ask about a BST algorithm or an adaptive sorting algorithm is how far, in the worst case, is the cost of this algorithm from any known lower bounds? Or, in other words, how close is this algorithm to optimal? In this section, we will settle this question for the log-interleave bound in the BST model, ending in the following theorem:

▶ **Theorem 17.** *For any permutation π, $IB(\pi) \leq LIB(\pi) \in O(\lg\lg n \; IB(\pi))$.*

Our first step is to show that we cannot hope to do better than a $\lg\lg n$ separation; this is stated in the following lemma. This result is similar to the separation result in Theorem 2 of Lecomte and Weinstein [26]; furthermore, the result implies that an online BST algorithm using $\mathrm{LIB}(\pi)$ accesses cannot be dynamically optimal.

▶ **Lemma 18.** *There exists a permutation π such that $LIB(\pi) = \Theta(\lg\lg n \; IB(\pi))$.*

Proof. First we will need to define a particularly useful permutation. The *bit-reversal permutation* π_B on $n = 2^k$ keys is generated by taking a sorted list $[0, 1, \ldots, n]$, writing each key in binary, then reversing the bits of each key. For example, the bit-reversal permutation on 8 keys is $[0, 4, 2, 6, 1, 5, 3, 7]$. Let P be a static tree with keys of π_B at the bottom: then querying its keys in sorted order will switch between the left and right subtree of any $v_j \in P$ on each query. This implies that $\mathrm{IB}(\pi_B) = \Theta(n \lg n)$, thus showing that any BST algorithm will incur this cost when querying π_B.

Consider the permutation π obtained by splitting the sorted list into $n/\lg n$ segments of equal size, and then permuting those $n/\lg n$ segments according to the bit-reversal permutation π_B.

The interleave bound of π will be the same as for a list with $n/\lg n$ elements permuted according to the bit-reversal sequence – that is, $(n/\lg n)\lg(n/\lg n) = O(n)$. In π, every block is of size $\lg n$, so to calculate the log-interleave bound, we multiply by $\lg\lg n$ on all but the bottom $\lg\lg n$ levels. Thus, the log-interleave bound of π is $\Theta(n \lg\lg n)$ while $\mathrm{IB}(\pi) = O(n)$. ◀

Now we have shown there is no possibility of doing better than a $\lg\lg n$ separation, we show that this separation is tight. This starts with the following question: when are the interleave bound and the log-interleave bound farthest apart? It follows from the convexity of the logarithm that for each vertex v_j of a static tree, the interleave bound and the

log-interleave bound are farthest apart when v_j experiences long runs of consecutive accesses to its subtrees (e.g. the list [L, L, L, R, R, R] has fairly different values for its interleave bound and log-interleave bound, but the list [L, R, L, R, L, R] does not).

However, a completely sorted list π_S – translating to the longest run size possible for each vertex of P – has $\text{IB}(\pi_S) = \text{LIB}(\pi_S) = \Theta(n)$. This suggests there must be some intermediate value of the block size that maximizes the difference between the two bounds. As the reader might have inferred from Lemma 18, that size will turn out to be $\lg n$. The next lemma, whose proof follows directly from the convexity of the logarithm, formalizes this intuition.

▶ **Lemma 19.** *For a permutation π, let v be a vertex of the corresponding static tree P such that $IB(v) = S$. Then $LIB(v)$ will differ from $IB(v)$ by the greatest amount when each "run" of L or R in the labeled sequence is the same size.*

Now that we have established that the interleave bound and the log-interleave bound differ the most when all continuous runs are the same size, we move on to ask the following question: how large do the runs of the same size have to be to further maximize this difference? The next lemma shows this fact in the following way: in the inequality below, the expression $S \lg\left(\frac{n}{S} + 1\right)$ bounds the log-interleave bound of any π such that $\text{IB}(\pi) = S$. The left-hand expression $c \lg(\lg n + 1)\left(S + \frac{n}{\lg n}\right)$ will directly suffice to prove Theorem 17. The proof of the lemma draws out the fact that the two expressions are closest to each other when the size of the continuous runs is $\lg n$.

▶ **Lemma 20.** *For a permutation π, let v be a vertex of the corresponding static tree P such that $IB(v) = S$. Furthermore, let the number of leaves below vertex v be n. Then for some constant c,*

$$c \lg(\lg n + 1)\left(S + \frac{n}{\lg n}\right) \geq S \lg\left(\frac{n}{S} + 1\right).$$

Proof. Assume for the sake of contradiction that

$$c \lg(\lg n + 1)S + c \lg(\lg n + 1)\frac{n}{\lg n} < S \lg\left(\frac{n}{S} + 1\right).$$

This would imply that each added term is smaller than $S \lg\left(\frac{n}{S} + 1\right)$. Begin by examining the case where the first term is smaller than the term on the right:

$$\begin{aligned} \lg(\lg n + 1)S &< S \lg\left(\frac{n}{s} + 1\right) \\ \implies \lg(\lg n + 1) &< \lg\left(\frac{n}{S} + 1\right) \\ \implies \lg n &< \frac{n}{S} \\ \implies S &< \frac{n}{\lg n}. \end{aligned}$$

This shows that when $S \geq \frac{n}{\lg n}$, we reach a contradiction and our claim holds. Now, when $S < \frac{n}{\lg n}$, the second term in the sum dominates. When the second term dominates, the expression reads

$$c \lg(\lg n + 1) \cdot \frac{n}{\lg n} \geq \frac{n}{\lg n} \lg(\lg n + 1)$$

which is self-evidently true for all $c \geq 1$. ◀

Now, these two lemmas are put together to prove Theorem 17.

Proof of Theorem 17. Let P be the static tree corresponding to π, and for each vertex v_i of P, let $S_i = \mathrm{IB}(v_i)$. By Lemma 19, we can assume that if the number of leaves below v_i is n_i, then $\mathrm{LIB}(v_i) = S_i \lg\left(\frac{n_i}{S} + 1\right)$. Then $\mathrm{IB}(\pi) = \sum_{i=1}^{n-1} S_i$. Next, we can use the upper bound on $\mathrm{LIB}(v_i)$ from Lemma 20 to upper bound $\mathrm{LIB}(\pi)$:

$$\begin{aligned}\mathrm{LIB}(\pi) &\leq \sum_{i=1}^{n-1} c \lg\lg n \left(S + \frac{n_i}{\lg n}\right) \\ &= c \lg\lg n \left(\mathrm{IB}(\pi) + \frac{1}{\lg n} \sum_{i=1}^{\lg n} 2^i \frac{n}{2^i}\right) \\ &= c \lg\lg n \left(\mathrm{IB}(\pi) + n\right) = O(\mathrm{IB}(\pi) \lg\lg n).\end{aligned}$$

◀

5 Adaptive Parallel Mergesort

In this section we present a parallel BST mergesort which sorts a permutation π using $O(\mathrm{LIB}(\pi))$ accesses, and the same amount of work. We refer to the algorithm as an *adaptive parallel mergesort*.

First we introduce the data structure used in our mergesort. Given a BST T and a key k, a **split** refers to returning two BSTs, one containing all keys from T which are greater than k, and one containing all keys which are less than k. Given two BSTs T_1, T_2 such that any key in T_1 is greater than every key in T_2, **join** returns a single BST T containing the union of the keys in T_1 and T_2. As previously stated, we assume keys are unique.

The tree used in our mergesort algorithm is a modified red-black tree described by Tarjan and Van Wyck in [36], which they call a *heterogeneous finger search tree*. These trees have the useful property that a key d in a heterogenous finger search tree with n elements can be accessed in time $O(\log(\min(d, n-d)+1))$. This property allowed Tarjan and Van Wyck to devise fast split and join algorithms for heterogenous finger search trees; split runs in amortized time $O(\lg(\min(|T_1|, |T_2|)+1))$ – that is, the logarithm of the size of the smaller tree returned. Join similarly is bounded by amortized time $O(\lg(\min(|T_1|, |T_2|)+1))$ – in this case, the size of the smaller of the two trees being joined together. The worst-case complexity of split and join is $O(\log \max |T_1|, |T_2|)$. As presented in [36], the heterogeneous finger search tree is not strictly a BST as it uses more than one pointer; however, work by [13] shows how it can be converted into using a single pointer with an additional constant factor loss.

The natural parallel algorithm to merge two trees is as follows: starting with two trees, split each tree using the other tree's root; then, recurse in parallel to merge the two left halves and the two right halves, respectively, joining the two at the end. This idea was presented by Blelloch et al. [4], and is shown here in Algorithm 2. This algorithm, however, does not meet the log-interleave bound even if we use heterogeneous finger search trees for the split and join. We therefore modify the algorithm as is shown in Algorithm 3 and illustrated in Figure 4, which follows the same idea with some small modifications. In addition to splitting the second tree T_2 into L_2 and R_2 based on the root of the first tree (T_1), it then splits T_1 by the maximum value of L_2 and the minimum value of R_2 to effectively break T_1 into three parts. The middle part need not be split recursively since it falls between two elements of T_2. This avoids redundant splits.

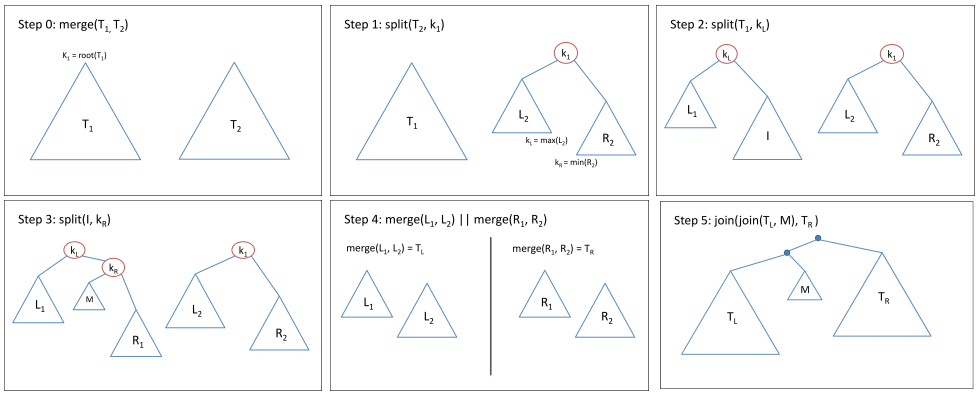


Figure 4 One round of our recursive merge algorithm. The nodes shown in red are the nodes used to split a tree; the small blue nodes denote merges.

Algorithm 2 union(T_1, T_2).
Blelloch et al.'s union algorithm. Here, the function *expose* refers to returning the root and its right and left subtrees.

Input: Two BSTs T_1, T_2 with disjoint keys.
Output: A BST containing the union of the keys of T_1 and T_2

if $T_1 =$ *Leaf* **then return** T_2 ;
else if $T_2 =$ *Leaf* **then return** T_1 ;
else
  $L_1, k_2, R_1 = \text{expose}(T_1)$;
  $L_2, R_2 = \text{split}(T_2, k)$;
  do in parallel
    $T_L = \text{union}(L_1, L_2)$;
    $T_R = \text{union}(R_1, R_2)$;
return *join*(T_L, k_2, T_R);

Algorithm 3 mergeHT(T_1, T_2)
Pseudocode for the merge step of our mergesort.

Input: Two BSTs T_1, T_2
Output: A BST containing the union of the keys of T_1 and T_2

if $T_1 =$ *Leaf* **then return** T_2 ;
else if $T_2 =$ *Leaf* **then return** T_1 ;
else
  $k = \text{root}(T_1)$;
  $L_2, R_2 = \text{split}(T_2, k)$;
  $k_1 = \max(L_2)$; $k_2 = \min(R_2)$;
  $L_1, I = \text{split}(T_1, k_1)$;
  $M, R_1 = \text{split}(I,\ k_2)$;
  do in parallel
    $T_L = \text{merge}(L_1, L_2)$;
    $T_R = \text{merge}(R_1, R_2)$;
return *join*(*join*(T_L, M), T_R);

It is not immediate our modified algorithm's work is bounded by the log-interleave bound, since the set of splits and joins it performs does not neatly correspond to the sums of block sizes at each level in the static tree used to calculate the log-interleave bound. We will show that this different sequence of splits and joins also performs within the log-interleave bound, culminating in the following theorem:

▶ **Theorem 21.** *There exists a parallel mergesort which for any permutation π performs $O(LIB(\pi))$ work with polylogarithmic span.*

Furthermore, since the algorithm proposed is a BST mergesort, it follows from Theorem 6 that there also exists an offline BST algorithm with the same cost in the BST model:

▶ **Corollary 22.** *There exists an offline BST algorithm $\mathcal{A}$ such that $\mathcal{A}(\pi) = O(LIB(\pi))$.*

The proofs of Theorem 21 and Corollary 22 are shown in the full version of the paper.

References

1 N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), April 2001.

2 Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of TimSort. In *European Symposium on Algorithms (ESA)*, volume 112. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

3 Mihai Badoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2), 2007.

4 Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

5 Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

6 Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999.

7 Prosenjit Bose, Jean Cardinal, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Competitive online search trees on trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2020.

8 Prosenjit Bose, Karim Douïeb, John Iacono, and Stefan Langerman. The power and limitations of static binary search trees with lazy finger. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 8889. Springer, 2014.

9 Mark R Brown and Robert E Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2), 1979.

10 Mark R. Brown and Robert Endre Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.*, 9(3):594–614, 1980.

11 Svante Carlsson and Jingsen Chen. An optimal parallel adaptive sorting algorithm. *Inf. Process. Lett.*, 39(4), 1991.

12 Parinya Chalermsook, Julia Chuzhoy, and Thatchaphol Saranurak. Pinning down the strong wilber 1 bound for binary search trees. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 33:1–33:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.APPROX/RANDOM.2020.33.

13 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Multi-finger binary search trees. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 123. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

14 Jingsen Chen and Christos Levcopoulos. Improved parallel sorting of presorted sequences. In *Joint Conference on Vector and Parallel Processing (CONPAR - VAPP)*, volume 634. Springer, 1992.

15 Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1), 2000.

16 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), 2000.

17 Curtis R. Cook and Do Jin Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23(11), 1980.

18 Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patrascu. The geometry of binary search trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009.

19 Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality–almost. *SIAM Journal on Computing*, 37(1), 2007.

20 J. Derryberry, D. D. Sleator, and C. C. Wang. A lowerbound framework for binary search trees with rotations. Technical report, Technical Report CMU-CS-05-187, School of Computer Science, Carnegie Mellon University, 2005.

21 Jonathan Derryberry. *Adaptive Binary Search Trees*. PhD thesis, Carnegie Mellon University, 2009.

22 Vladimir Estivill-Castro and Derick Wood. A new measure of presortedness. *Information and Computation*, 83(1), 1989.

23 John Iacono and Stefan Langerman. Weighted dynamic finger in binary search trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2016.

24 Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. Local insertion sort revisited. In *International Symposium on Optimal Algorithms*, 1989.

25 László Kozma and Thatchaphol Saranurak. Smooth heaps and a dual view of self-adjusting data structures. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 2018.

26 Victor Lecomte and Omri Weinstein. Settling the relationship between Wilber's bounds for dynamic optimality. In *European Symposium on Algorithms (ESA)*, 2020.

27 Christos Levcopoulos and Ola Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3), 1983.

28 Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. In *Scandinavian Workshop on Algorithm Theory*, 1990.

29 Christos Levcopoulos and Ola Petersson. Splitsort–an adaptive sorting algorithm. *Information Processing Letters*, 39(4), 1991.

30 Christos Levcopoulos and Ola Petersson. Exploiting few inversions when sorting: Sequential and parallel algorithms. *Theor. Comput. Sci.*, 163(1&2), 1996.

31 Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4), 1985.

32 Peter McIlroy. Optimistic sorting and information theoretic complexity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

33 Alistair Moffat and Ola Petersson. Historical searching and sorting. In *International Symposium on Algorithms*, 1991.

34 Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59, 1995.

35 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3), 1985.

36 Robert Endre Tarjan and Christopher J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM J. on Computing*, 17(1), 1988.

37 Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. $O(\log \log n)$-competitive dynamic binary search trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM Press, 2006.

38 Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1), 1989.