Joinable Parallel Balanced Binary Trees

GUY BLELLOCH, Carnegie Mellon University, USA
DANIEL FERIZOVIC, Karlsruhe Institute of Technology, Germany
YIHAN SUN, University of California, Riverside, USA

7

In this article, we show how a single function, *join*, can be used to implement parallel **balanced binary search trees** (**BSTs**) simply and efficiently. Based on *join*, our approach applies to multiple balanced tree data structures, and a variety of functions for ordered sets and maps. We describe our technique as an algorithmic framework called *join-based algorithms*. We show that the *join* function fully captures what is needed for rebalancing trees for a variety of tree algorithms, as long as the balancing scheme satisfies certain properties, which we refer to as *joinable* trees. We discuss four balancing schemes that are joinable: AVL trees, red-black trees, weight-balanced trees, and treaps. We present a variety of tree algorithms that apply to joinable trees, including *insert*, *delete*, *union*, *intersection*, *difference*, *split*, *range*, *filter*, and so on, most of them also parallel. These algorithms are generic across balancing schemes. Many algorithms are optimal in the comparison model, and we provide a general proof to show the efficiency in work for joinable trees. The algorithms are highly parallel, all with polylogarithmic span (parallel dependence). Specifically, the set-set operations *union*, *intersection*, and *difference* have work $O(m \log(\frac{n}{m} + 1))$ and polylogarithmic span for input set sizes n and $m \leq n$.

We implemented and tested our algorithms on the four balancing schemes. In general, all four schemes have quite similar performance, but the weight-balanced tree slightly outperforms the others. They have the same speedup characteristics, getting around 73× speedup on 72 cores (144 hyperthreads). Experimental results also show that our implementation outperforms existing parallel implementations, and our sequential version achieves close or much better performance than the sequential merging algorithm in C++ Standard Template Library (STL) on various input sizes.

CCS Concepts: • **Theory of computation → Sorting and searching**; **Shared memory algorithms**; • **Computing methodologies → Shared memory algorithms;**

Additional Key Words and Phrases: Balanced binary trees, searching, parallel, union

ACM Reference format:
Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2, Article 7 (April 2022), 41 pages.
https://doi.org/10.1145/3512769

This is an extended version for paper [14] published in Symposium on Parallel Algorithms and Architectures (SPAA) 2016.
This research was supported by NSF grants CCF-2103483, CCF-1901381, CCF-1910030, and CCF-1919223.
Authors' addresses: G. Blelloch, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA; email: guyb@cs.cmu.edu; D. Ferizovic, Karlsruhe Institute of Technology, Kaiserstraße 12, Karlsruhe, 76131, Germany; email: dani93.f@gmail.com; Y. Sun, 900 University Ave, Riverside, CA 92521, USA; email: yihans@cs.ucr.edu.

2329-4949/2022/04-ART7
https://doi.org/10.1145/3512769

1 INTRODUCTION

Many important and fundamental data structures used in algorithm design and programming are based on various forms of ***binary search trees (BSTs)***. Importantly, the in-order traversal of such trees provides an elegant and efficient mechanism to organize *ordered* data.[1] Typically, we say a binary tree is "balanced" if the length of the path to all nodes (i.e., the depth) is asymptotically logarithmic in the number of nodes. In the sequential setting of "point" updates to the tree (i.e., single insertions or deletions), many balancing schemes for binary trees have been widely studied and are widely used, such as AVL trees [4], red-black trees [10], weight-balanced trees [52], splay trees [57], treaps [56], and many others. Some guarantee the logarithmic depth with high probability, or when amortized across node accesses.

With the recent prevalence of multi-processor (core) parallel computers, it is important to study balanced binary trees in the parallel setting.[2] Although tree algorithms are fundamental and well studied in the sequential setting, applying them in the parallel setting is challenging. One approach is to make traditional updates such as insertions and deletions safe to apply concurrently. To work correctly, this requires various forms of locking and non-blocking synchronization [8, 9, 21, 24, 25, 30, 68]. Such concurrent updates can come with significant overhead, but more importantly only guarantee atomicity at the level of individual updates. Another approach is to update the trees in parallel by supporting "bulk" operations such as inserting a collection of keys (multi-insert), taking the union of two trees, or filtering the tree based on a predicate function. Here parallelism comes from within the operation. We refer to such algorithms as bulk parallel, or just parallel, as opposed to concurrent.

Recently, Blelloch et al. proposed an algorithmic framework [14] for such parallel bulk updates, which unifies multiple balancing schemes. Their algorithms have been applied to various applications [11, 60–62] and have been implemented in a parallel library PAM [59, 62]. Their framework bases all tree algorithms on a single primitive called *join*, which captures all the balancing criteria. As a result, many tree algorithms (most of them parallel) can be designed in a balancing-scheme-independent manner. Similar ideas of using *join* as a primitive for balanced binary trees, sequentially or in parallel, have also been considered in previous work [2, 3, 16, 64]. The original paper of Blelloch et al. about *join*-based algorithms shows that the framework works for four balancing schemes. However, the authors did not answer the more general question about *what makes a balancing scheme fit in the join-based algorithm framework*. This article extends the topic in [14] about efficient *join*-based parallel algorithms on binary trees, and specifically formalizes the preferred property for a balancing scheme to be *joinable*, which allows all *join*-based algorithms to be efficient on this balancing scheme.

As mentioned above, all the tree algorithms in this article are built on top of a single primitive *join*. For discussion purposes, we assume that a *binary tree* T can be either empty (a *nil-node*), or can be a node with a left tree T_L, data entry u (e.g., a key, or key-value), and a right tree T_R, denoted as $node(T_L, u, T_R)$. The function $join\ (T_L, k, T_R)$ for a given balancing scheme takes two balanced binary trees T_L and T_R, and a key (or key-value) k as input arguments, and returns a new valid balanced binary tree that has the same entries and the same in-order traversal as $node(T_L, k, T_R)$. The output has to be valid in satisfying the balancing criteria. We call the middle key k the *pivot* of the *join*. An illustration of the *join* function is presented in Figure 1. In this article, we use *join* as the

[1] Beyond BSTs, balanced binary trees can also be used to represent a sequence (list) of elements that are ordered by their position, but not necessarily by their value [40].

[2] In this article, we focus on the shared-memory multi-core setting. We note that comparing the superiority of different platforms is beyond the scope of this work. In Section 7, we will discuss some related work in the concurrent and distributed setting.

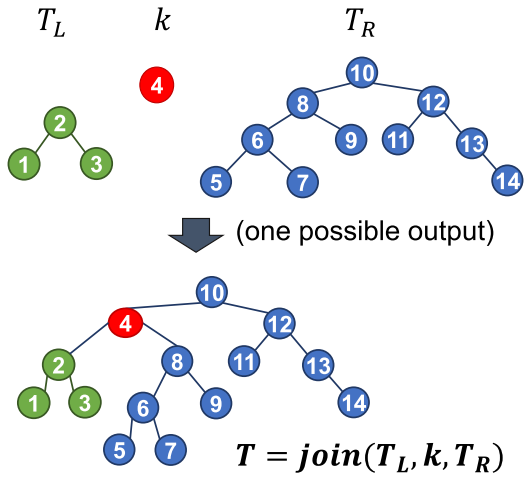


Fig. 1. An example of the *join* algorithm on AVL trees. The function *join* (T_L, k, T_R) takes two AVL trees T_L and T_R, and key k as input arguments, and returns a new valid AVL tree that has the same entries and the same in-order traversal as *node*(T_L, k, T_R). The above figure shows *one possible output* of *join* (T_L, k, T_R).

only primitive for connecting and rebalancing. Since rebalancing involves settling the balancing invariants, the *join* algorithm itself is specific to each balancing scheme. However, we note that in general, the *join* operation is oblivious to the values of the keys and depends only on the shape of the two trees.

One important observation of the *join*-based algorithms is that *join* captures all balancing criteria of each balancing scheme. As such, various tree algorithms (except *join* itself) can be implemented generically across balancing schemes. These algorithms need not be aware of the operated balancing scheme, but can just rely on the corresponding *join* algorithm for rebalancing. These *join*-based algorithms range from simple insertions and deletions, to more involved bulk operations. Moreover, the generality is not at the cost of efficiency in asymptotical cost or practical performance. In fact, the generic code leads to similar good performance for different balancing schemes (see Section 6). Also, all the *join*-based algorithms are still optimal in sequential work, and most of them can be parallelized with poly-logarithmic span (parallel dependency chain). These include some non-trivial and interesting theoretical results. For example, the *join*-based *union* algorithm, which combines two balanced binary search trees into one (keeping the ordering), costs $O(m \log(\frac{n}{m} + 1))$ work and $O(\log n \log m)$ span on two trees with sizes m and $n \geq m$. This work bound is optimal under the comparison model [37].

To show the cost bounds of the *join*-based algorithms, one also must consider multiple balancing schemes. Fortunately, *join* also unifies the theoretical analysis. The key idea is to define a *rank* function for each balancing scheme, that maps every (sub-)tree to a real number, called its *rank*. The rank can be thought, roughly, as the abstract "height" of a tree. For example, for AVL trees the rank is simply the height of the tree, and for weight-balance trees, it is the log of the size. We then define a set of rules about rank and the *join* algorithm, that ensure a balancing scheme to be *joinable*. These rules apply to at least four existing balancing schemes. Based on these rules, we are able to show generic algorithms and analysis for all joinable trees. Our bounds for *join*-based algorithms then hold for any balancing scheme for which these properties hold.

We also implemented our algorithms and compare the performance with existing implementations, sequentially and in parallel. For merging two sets S_1 and S_2, we compare our *union*

algorithm to existing data structures that support batch updates of a batch S_2 on a tree of S_1. Our implementation has close performance when $|S_1|$ is large or close to $|S_2|$, and outperforms existing implementations by about 8× when $|S_2|$ is much larger than $|S_1|$. We also compare our sequential version with STL tree and vector merging algorithms. Our implementation running on one core is about 8× faster for *union* of two equal sized maps (10^8 each) than STL-tree, and over four orders of magnitude faster than STL-vector when one is of size 10^4 and the other 10^8. This is because our implementation is asymptotically more efficient than the STL implementation.

In the rest of the article, we will first show some preliminaries in Section 2. We will then present the useful properties that make a balancing scheme *joinable* in Section 3. We then present the *join* algorithms for four balancing schemes and show that they are all joinable in Section 4. In Section 5, we show several *join*-based algorithms, and present their cost bound. Finally, we show some experimental results in Section 6. Some related work and discussions are presented in Section 7.

2 PRELIMINARY

Parallel Cost Model. Our algorithms are based on nested fork-join parallelism and no other synchronization or communication among parallel tasks.[3] All analysis is using the *binary-forking* model [15], which is based on nested fork-join, but only allows for two branches in a fork. All our algorithms are deterministic.[4] To analyze asymptotic costs of a parallel algorithm, we use *work* W and *span* (or *depth* D), where work is the total number of operations and span is the length of the critical path. In the algorithm descriptions, the notation of $s_1 \mid\mid s_2$ indicates that statement s_1 and s_2 can run in parallel. Any computation with W work and D span will run in time $T < \frac{W}{P} + D$ assuming a **PRAM (random access shared memory)** [38] with P processors and a greedy scheduler [18, 20, 34], or in time $T < \frac{W}{P} + D$ w.h.p. (defined below) when using a work-stealing scheduler [19]. We assume **concurrent reads and exclusive writes (CREW)**.

Notation. We use $\langle \cdot, \cdot \rangle$ to denote a pair (similarly for tuples and sequences). We say $O(f(n))$ **with high probability (w.h.p.)** in n to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$. When clear from context, we drop the "in n."

2.1 Balanced Binary Trees

A *binary tree* is either a *nil-node*, or a node consisting of a *left* binary tree T_l, a data entry k, and a *right* binary tree T_r, which is denoted as $node(T_l, k, T_r)$. We use the *nil-node* to refer to an external (empty) node with no data stored in it. The data entry can be simply a value (e.g., a key), or also hold data associated with the value. The *size* of a binary tree, or $|T|$, is 0 for a *nil-node* and $|T_l| + |T_r| + 1$ for a $node(T_l, k, T_r)$. The *weight* of a binary tree, or $w(T)$, is one more than its size (i.e., the number of *nil-node*s in the tree). The *height* of a binary tree, or $h(T)$, is 0 for a *nil-node*, and $\max(h(T_l), h(T_r)) + 1$ for a $node(T_l, k, T_r)$. *Parent*, *child*, *ancestor*, and *descendant* are defined as usual (ancestor and descendant are inclusive of the node itself). We use $lc(T)$ and $rc(T)$ to extract the left and right child (subtree) of T, respectively. A node is called a *leaf* when both of its children are *nil nodes*. The *left spine* of a binary tree is the path of nodes from the root to a *nil-node* always following the left child, and the *right spine* the path to a leaf following the right child. The *in-order values* of a binary tree is the sequence of values returned by an in-order traversal of the tree. When the context is clear, we use a node u to refer to the subtree T_u rooted at u, and vice versa.

[3] This does not preclude using our algorithms in a concurrent setting.

[4] Note that the bounds and the data structure themselves are not necessarily deterministic. For example, the treaps depends on random priorities. However, as long as the priorities are known (independently of the algorithms), the algorithm does not use randomization.

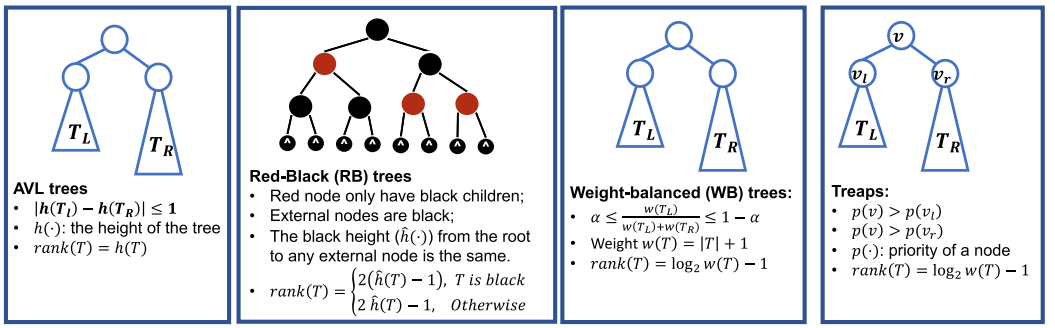


Fig. 2. The balancing schemes used in this article. $rank(\cdot)$ is defined in Section 4 for each balancing scheme.

A *balancing scheme* for binary trees is an invariant (or set of invariants) that is true for every node of a tree, and is for the purpose of keeping the tree nearly balanced. In this article, we consider four balancing schemes that ensure the height of every tree of size n is bounded by $O(\log n)$. When keys have a total order and the in-order values of the tree are consistent with the order, then we call it a *BST*. We note that the balancing schemes defined below, although typically applied to BSTs, do not require that the binary tree be a search tree.

We will then briefly introduce the balancing schemes that we use in this article. An illustration of these balancing schemes is shown in Figure 2.

AVL Trees [4]. AVL trees have the invariant that for every $node(T_l, e, T_r)$, the height of T_l and T_r differ by at most 1. This property implies that any AVL tree of size n has height at most $\log_\phi(n+1)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Red-Black (RB) Trees [10]. RB trees associate a color with every node and maintain two invariants: (the red rule) no red node has a red child, and (the black rule) the number of black nodes on every path from the root down to a leaf is equal. All *nil nodes* are always black. Unlike some other presentations, we do not require that the root of a tree is black. Although this does not affect the correctness of our algorithms, our proof of the work bounds requires allowing a red root. We define the *black height* of a node T, denoted $\hat{h}(T)$ to be the number of black nodes on a downward path from the node to a *nil-node* (inclusive of the node and the *nil-node*). Any RB tree of size n has height at most $2\log_2(n+1)$.

Weight-Balanced (WB) Trees [52]. WB trees with parameter α (also called BB[α] trees) maintain for every $T = node(T_l, e, T_r)$ the invariant $\alpha \leq \frac{w(T_l)}{w(T)} \leq 1 - \alpha$. We say two WB trees T_1 and T_2 have *like* weights if $node(T_1, e, T_2)$ is WB. Any α WB tree of size n has height at most $\log_{\frac{1}{1-\alpha}} n$. For $\frac{2}{11} < \alpha \leq 1 - \frac{1}{\sqrt{2}}$, insertion and deletion can be implemented on WB trees using just single and double rotations [17, 52]. We require the same condition for our implementation of *join*, and in particular use $\alpha = 0.29$ in experiments. We also denote $\beta = \frac{1-\alpha}{\alpha}$, which means that either subtree cannot have a size that is more than β times the size of the other subtree.

Treaps [56]. Treaps associate a uniformly random priority with every key and maintain the invariant that the priority of the key at each node is no greater than the priority of its two children. Any treap of size n has height $O(\log n)$ w.h.p.. The priority can usually be decided by a hash value of the key, and thus for operations across search trees we assume equal keys have equal priorities.

The notation we use for binary trees is summarized in Table 1. Some concepts in Table 1 will be introduced later in this article. In particular, we will define the $rank(\cdot)$ function for each balancing

Table 1. Summary of Notation

Notation	Description
$\|T\|$	The size of tree T
$h(T)$	The height of tree T
$\hat{h}(T)$	The black height of an RB tree T
$rank(T)$	The rank of tree T
$w(T)$	The weight of tree T (i.e., $\|T\| + 1$)
$p(T)$	The parent of node T
$k(T)$	The key of node T
$lc(T)$	The left child of node T
$rc(T)$	The right child of node T
$\texttt{expose}(T)$	$\langle lc(T), e(T), rc(T)\rangle$

The definition of some of them are postponed to Section 5.3.

scheme, which is generally based on their balancing criteria. We do so because that can help to simplify the proof of bounding the cost of the *join*-based algorithms on trees. We note that the *rank* functions are defined for the purpose of analysis. As long as the *join* function is implemented correctly as defined, all the *join*-based algorithms will behave as expected.

3 JOINABLE TREES

Here, we define the properties that make a tree *joinable*. These properties are used in later sections to prove bounds on each of our algorithms—i.e., if someone gives us a balancing scheme with these properties, our algorithms will be efficient without needing to know anything else about how balance is maintained. The concept of joinable trees relies on two subcomponents: a value associated with each tree node (subtree) called the *rank*, and a proper *join* algorithm. The definition of rank and the *join* algorithm depend on the balancing scheme. The rank of a tree node only relies on the shape of the subtree rooted at it, not the set of entries. For a tree T, we denote its rank as $rank(T)$.

We note that the joinable properties are mainly for the purpose of analysis. In fact, as long as the *join* function is implemented *correctly* (no need to be *efficiently*), all the *join*-based algorithms in later sections will also be *correct*. However, to use the general proof to show the cost bounds, these properties will help to simplify the analysis.

Definition 1 (Strongly Joinable Trees). A balancing scheme $\mathcal{S}$ is *strongly joinable*, if we can assign a *rank* for each subtree from $\mathcal{S}$, and there exists a *join* (T_1, k, T_2) algorithm on two trees from $\mathcal{S}$, such that the following rules hold:

(1) **EMPTY RULE**. The rank of a *nil-node* is 0.
(2) **MONOTONICITY RULE**. For $C = join(A, k, B)$, $\max(rank(A), rank(B)) \leq rank(C)$.
(3) **SUBMODULARITY RULE**. Suppose $C = node(A, e, B)$ and $C' = join(A', e, B')$. If $0 \leq rank(A') - rank(A) \leq x$ and $0 \leq rank(B') - rank(B) \leq x$, then $0 \leq rank(C') - rank(C) \leq x$ (increasing side). In the other direction, if $0 \leq rank(A) - rank(A') \leq x$ and $0 \leq rank(B) - rank(B') \leq x$, then $0 \leq rank(C) - rank(C') \leq x$ (decreasing side).
(4) **COST RULE**. $join(A, k, B)$ uses time $O(|rank(A) - rank(B)|)$.
(5) **BALANCING RULE**. For a node A,

$$\max(rank(lc(A)), rank(rc(A))) + c_l \leq rank(A) \leq \min(rank(lc(A)), rank(rc(A))) + c_u,$$

where $c_l \leq 1$ and $c_u \geq 1$ are constants.

The last rule about balancing says that the ranks of a child and its parent cannot differ by much. This is not true for some randomization-based balancing schemes such as treaps. To generalize our results to such balancing schemes, we define a weakly joinable tree as follows.

Definition 2 (Weakly Joinable Trees). A balancing scheme $\mathcal{S}$ is *weakly joinable*, if it satisfies the **empty rule**, **monotonicity rule**, and **submodularity rule** in Definition 2, and the **relaxed balancing rule** and **weak cost rule** as follows:

- **relaxed balancing rule**. There exist constants c_l, c_u, $0 < p_l \leq 1$ and $0 < p_u \leq 1$, such that for a node A and any of its child B:
  (1) $rank(B) \leq rank(A) - c_l$ happens with probability at least p_l.
  (2) $rank(B) \geq rank(A) - c_u$ happens with probability at least p_u.
- **weak cost rule**. $join(A, k, B)$ uses time $O(rank(A) + rank(B))$ w.h.p.

In this article, we use *joinable* to refer to weakly joinable trees. Later in this section, we will show that AVL trees, RB trees, and WB trees are strongly joinable, and treaps are weakly joinable.

We also say a tree T is strongly (weakly) joinable if T is from a strongly (weakly) joinable balancing scheme.

Here we first present some properties of joinable trees.

Property 1. *For a strongly joinable tree T, $c_l h(T) \leq rank(T) \leq c_u h(T)$.*

This can be inferred directly from the **balancing rule**.

Property 2 (Logarithmic Rank for Strongly Joinable Balanced Trees). *For a strongly joinable tree T, $rank(T) = O(\log w(T))$, where $w(T) = |T| + 1$ is T's weight.*

Proof. Let $f(r)$ be a function denoting the minimum weight that T can have when T has rank r, i.e., $f(r) = \min_{rank(T)=r} w(T)$. We now look at the two children of T, noted as L and R. **Without loss of generality** (**WLOG**), assume $w(L) \geq w(R)$. In the **balancing rule**, we know that $rank(T) - c_u \leq rank(R) \leq rank(T) - c_l$. The total weight of T is at most $2w(R)$. In other words,

$$f(r) \geq 2 \min_{r-c_u \leq r' \leq r-c_l} f(r').$$

This means that $f(r) \geq c \cdot 2^{r/c_u}$ for some constant c. Thus, $rank(T)$ is at most $O(\log w(T))$. □

From the above lemma we have the following.

Property 3 (Logarithmic Height for Strongly Joinable Balanced Trees). *For a strongly joinable tree T with weight n, $h(T) = O(rank(T)) = O(\log n)$.*

From the **balancing rule** we have the following.

Property 4 (Similar Ranks for Balanced Strongly Joinable Trees). *For a strongly joinable tree $T = node(lc(T), k, rc(T))$, then $rank(lc(T))$ and $rank(rc(T))$ differ by at most a constant.*

For weakly joinable trees, we also have a similar property for rank and height.

Property 5 (Logarithmic Height for Weakly Joinable Balanced Trees). *For a weakly joinable tree T with weight n, $h(T) = O(rank(T)) = O(\log n)$ w.h.p.*

Proof. First of all, note that for a parent node A and a child node B, $h(A) = h(B) + 1$, but $rank(A) \geq rank(B) + c_l$ happens with a constant probability. Thus, $h(T) = O(rank(T))$.

Consider a node T with rank $r = rank(T)$. We follow the path and always go to the smaller subtree (with smaller weight). We call it a *round* every time we go down one level. After m rounds we hit the leaf. Obviously, $m \leq \log n$ since every time we go to the smaller side of the subtree.

We note that with every such round, the rank increased by no more than c_u with a constant probability. Adding all the m rounds gets r, which is the rank of T. Based on Chernoff bound, we know that $\Pr[r \geq (1+\delta)c_u p_u m] \leq e^{-\frac{\delta c_u p_u m}{3}}$. Let $(1+\delta)c_u p_u m$ be $c \log n$ for sufficient large c; we have

$$\begin{aligned}\Pr[r \geq c \log n] &\leq e^{-\frac{c \log n - c_u m}{3}} \\ &= n^{-c/3} \cdot e^{\frac{c_u p_u m}{3}} \\ &\leq n^{-c+c_u p_u/3}.\end{aligned}$$

□

We then extend the **monotonicity rule** and prove the following theorem.

Property 6 (Bounded Rank Increasing by *join*). *For a joinable tree $C = join(A, k, B)$, there exists a constant $c' \leq c_u$, such that*

$$\max(rank(A), rank(B)) \leq rank(C) \leq \max(rank(A), rank(B)) + c'.$$

Proof. The left half of the inequation holds because of the **monotonicity rule**.

For the right half, consider a single tree node $v = node(nil\text{-}node, e, nil\text{-}node)$. $rank(v)$ is at most c_u (the **balancing rule**). Consider $v' = join(A, k, B)$, which increases the rank of both sides by at most $\max(rank(A), rank(B))$. Then

$$rank(v') \leq rank(v) + \max(rank(A), rank(B)) \leq c_u + \max(rank(A), rank(B)).$$

□

We then prove some useful definitions and theorems that will be used in later proofs of the *join*-based algorithms.

Definition 3 (Layer i in Joinable Trees). In a joinable tree, we say a node v in layer i if $i \leq rank(v) < i + 1$.

Definition 4 (Rank Root). In a joinable tree, we say a node v is a *rank root*, or a rank(i)-root if v in layer i and v's parent is not in layer i. We use $s_i(T)$ to denote the number of rank(i)-root nodes in tree T.

When the context is clear, we simply use s_i.

Property 7. *For a joinable tree, suppose a node T has rank r; then the number of its descendants with rank more than $r - 1$ is a constant.*

Proof. From the **balancing rule** we know that after $1/c_l$ generations from T, the rank of the tree node must be smaller than $r-1$. That is to say, there are at most $2^{1/c_l}$ such nodes in T's subtree with rank no smaller than $r - 1$. □

Definition 5 (Rank Cluster). For any rank root v, its *rank cluster* contains all tree nodes with rank in the range $(rank(v) - 1, rank(v)]$. We use $d(v)$ of a rank root v to denote the size of its rank cluster.

For any v, these rank clusters form a contiguous tree-structured component.

Definition 6. In a BST, a set of nodes V is called *ancestor-free* if and only if for any two nodes v_1, v_2 in V, v_1 is not the ancestor of v_2.

Obviously, for each layer i, all the rank(i)-root nodes form an ancestor-free set.

The following lemma is important for proving the work bound of the *join*-based set-set algorithms in Section 5.3.

Lemma 1. *There exists a constant t such that for a strongly joinable tree with size N, $s_i \leq \frac{N}{2^{\lfloor i/t \rfloor}}$. More precisely, $t = 1 + \lceil c_u \rceil$.*

Proof. We now organize all rank roots from layer kt to layer $(k+1)t-1$ into a *group* k, for an integer k. Assume there are s'_k such rank roots in root layer k. We first prove that $s'_k \leq s'_{k-1}/2$, which indicates that $s'_k \leq \frac{N}{2^k}$.

For a node u in group k, we will show that its subtree contains at least two nodes in group $k-1$, one in each subtree. u's rank is at least $kt-1$ (because of rounding), but at most $(k+1)t-1$. For the left subtree (the right one is symmetric), we follow one path until we find a node in group k but its children v in a group smaller than k, so $rank(v) \leq kt-1$. We will prove that v must be in group $k-1$. Because of the **balancing rule**, $rank(u) - c_u \leq rank(v) \leq kt-1$. Considering the range of u's rank, we have $(k-1)t \leq rank(v) \leq kt-1$. This means that v is in group $k-1$. Therefore, every node in group k corresponds to at least two nodes in group $k+1$. This proves $s'_k \leq s'_{k-1}/2$.

Obviously, each set of rank(i)-root is a subset of its corresponding group. This proves the above lemma. □

Figure 14(b) shows an example of the layers of an AVL tree, in which we set the rank of a node to be its height. Because the rank of all AVL nodes are integers, all nodes are rank roots. Lemma 1 says that from the bottom up, every three layers, the number of nodes in an AVL shrinks by a half.[5]

4 THE *JOIN* ALGORITHMS AND RANKS FOR EACH BALANCING SCHEME

Here we describe the *join* algorithms for the four balancing schemes we defined in Section 2, as well as the definition of rank for each of them. We will then prove they are joinable. For *join*, the pivot can be either just the data entry (such that the algorithm will create a new tree node for it), or a pre-allocated tree node in memory carrying the corresponding data entry (such that the node may be reused, allowing for in-place updates).

As mentioned in Section 1, *join* fully captures what is required to rebalance a tree and can be used as the only function that knows about and maintains the balance invariants. For AVL, RB, and WB trees, we show that *join* takes work that is proportional to the difference in the ranks of the two trees. For treaps, the work depends on the priority of the pivot. All the *join* algorithms are sequential so the span is equal to the work. We show in this article that the *join* algorithms for all balancing schemes we consider lead to optimal work for many functions on maps and sets.

At a high level, all the *join* algorithms try to find a "balancing point" in the larger tree, which is a node v on its left or right spine, and is balanced with the smaller tree, join at that point, and rebalance along the way back (see Figure 3). WLOG, we assume in $join(T_L, k, T_R)$, T_L has larger or equal rank (e.g., for an AVL it means T_L has larger or equal height). In this case, we go along the right spine of T_L until we reach a node that is balanced with T_R. We then connect v with T_R using k, and let this new subtree replace v in T_L. This guarantees that the subtree below v is balanced, but may cause imbalance in v's ancestors. The algorithm then will try to rebalance the nodes on the way back to the root, and the rebalancing operations would depend on each specific balancing scheme. This high-level idea applies to multiple balancing schemes in this article (except for the treap *join* algorithm, which is slightly different because of the variant about priorities). In the following, we will elaborate each of them, and prove the cost bounds. The main result of this section is the following theorem.

Theorem 1. *AVL, RB, and WB trees are strongly joinable, and treaps are weakly joinable.*

[5] Actually, considering that there is no rounding issue for AVL trees, this means that from the bottom up, every two layers, the number of nodes in an AVL tree shrinks by a half.

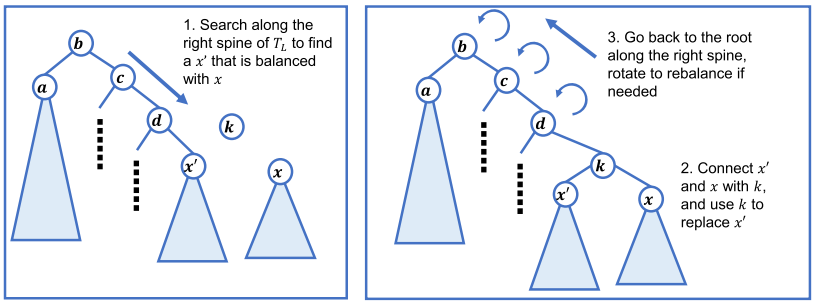


Fig. 3. The high-level idea of the *join* algorithm. An illustration of the high-level idea of the *join* algorithm (when T_L is larger than T_R, and the other case is symmetric).

4.1 AVL Trees

For AVL trees, we define the rank to be the height, i.e., $rank(T) = h(T)$. Pseudocode for AVL *join* is given in Figure 4 and illustrated in Figure 5. Every node stores its own height so that $h(\cdot)$ takes constant time. If the two trees T_l and T_r differ by height at most one, *join* can simply create a new $node(T_l, k, T_r)$. However, if they differ by more than one, then rebalancing is required. Suppose that $h(T_l) > h(T_r) + 1$ (the other case is symmetric). The idea is to follow the right spine of T_l until a node c for which $h(c) \leq h(T_r) + 1$ is found (line 3). At this point, a new $node(c, k, T_r)$ is created to replace c (line 4). Since either $h(c) = h(T_r)$ or $h(c) = h(T_r) + 1$, the new node satisfies the AVL invariant, and its height is one greater than c. The increase in height can increase the height of its ancestors, possibly invalidating the AVL invariant of those nodes. This can be fixed either with a double rotation if the invariant is invalid at the parent (line 6) or a single left rotation if the invariant is invalid higher in the tree (line 10), in both cases restoring the height for any further ancestor nodes. The algorithm will therefore require at most two rotations, as we summarized in the following lemma.

Lemma 2. *The join algorithm in Figure 4 on AVL trees requires at most two rotations.*

This lemma directly follows the algorithm description above.

Lemma 3. *For two AVL trees T_l and T_r, the AVL join algorithm works correctly, runs with $O(|h(T_l) - h(T_r)|)$ work, and returns a tree satisfying the AVL invariant with height at most $1+\max(h(T_l), h(T_r))$.*

Proof. Consider the case where T_L has a larger height (the other case is symmetric). After connecting the small tree with tree node c (as described above and in Figure 4), all tree nodes below k are guaranteed to be balanced. The only tree nodes that can be unbalanced are those on the right spine of T_L and above k. The algorithm then checks them bottom-up. If the imbalance occurs upon connection, which is as shown in Figure 4, then the double-rotation shown in the figure fixes the imbalance. Note that, after rotation, the entire subtree height restores to $h + 2$, which is the same as the subtree size in T_l before *join*. This means that no further rebalance is needed.

Another case is that, the imbalance does not directly occur at node p (k's parent in the output). Some other nodes on the right spine were affected to unbalanced because its right subtree height increased by 1. Assume the lowest such node is B, with left child A and right child C. Assume $h(A) = h$. This case happens only when $h(C)$ was $h + 1$, but increased to $h + 2$ after *join*. In other words, $h(rc(C)) = h + 1$ after join, and $h(lc(C))$ must be h (both before and after *join*, since the shape of C''s left subtree is not affected by *join*). In that case, a single rotation on B will rebalance the subtree, getting a new subtree rooted at C with height $h + 2$. Note that this is also the original height of the subtree in T_L rooted at B. As a result, no further rebalance is needed.

```
1  joinRightAVL(T_l, k, T_r) {
2    (l, k', c) = expose(T_l);
3    if h(c) ≤ h(T_r) + 1 then {
4      T' = node(c, k, T_r);
5      if h(T') ≤ h(l) + 1 then return node(l, k', T');
6      else return rotateLeft(node(l, k', rotateRight(T')));
7    } else {
8      T' = joinRightAVL(c, k, T_r);
9      T'' = node(l, k', T');
10     if h(T') ≤ h(l) + 1 then
11       return T'';
12     else return rotateLeft(T''); }}
13 join(T_l, k, T_r) {
14   if h(T_l) > h(T_r) + 1 then return joinRightAVL(T_l, k, T_r);
15   else if h(T_r) > h(T_l) + 1 then return joinLeftAVL(T_l, k, T_r);
16   else return node(T_l, k, T_r); }
```

Fig. 4. The *join* algorithm on AVL trees. `joinLeftAVL` is symmetric to `joinRightAVL`.

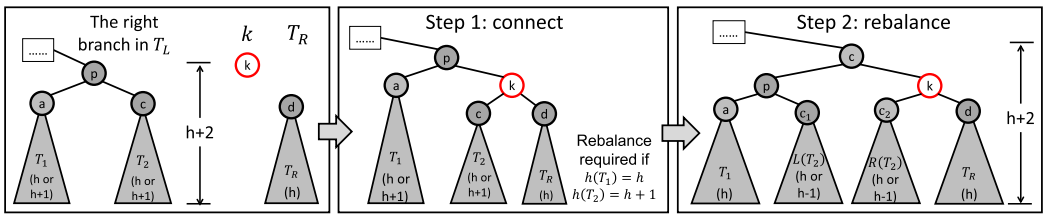


Fig. 5. An example for *join* on AVL trees. An example for *join* on AVL trees ($h(T_l) > h(T_r) + 1$). We first follow the right spine of T_l until a subtree of height at most $h(T_r) + 1$ is found (i.e., T_2 rooted at c). Then, a new $node(c, k, T_r)$ is created, replacing c (Step 1). If $h(T_1) = h$ and $h(T_2) = h + 1$, the node p will no longer satisfy the AVL invariant. A double rotation (Step 2) restores both balance and its original height.

Since the algorithm only visits nodes on the path from the root to c, and only requires at most two rotations (Lemma 2), it does work proportional to the path length. The path length is no more than the difference in height of the two trees since the height of each consecutive node along the right spine of T_l differs by at least one. Along with the case when $h(T_r) > h(T_l) + 1$, which is symmetric, this gives the stated work bounds. The resulting tree satisfies the AVL invariants since rotations are used to restore the invariant. The height of any node can increase by at most one, so the height of the whole tree can increase by at most one. □

Theorem 2. *AVL trees are strongly joinable.*

Proof. The AVL invariant and the definition of AVL trees' rank ensure the **empty rule** and **balancing rule** ($c_l = 1$, $c_u = 2$). Lemma 3 ensures the **cost rule** and **monotonicity rule**. We prove the **submodularity rule** as follows.

We start with the increasing side.

First note that the ranks of AVL trees are always integers. For $C = node(A, k, B)$ and $C' = join(A', k, B')$, WLOG suppose $rank(A) \geq rank(B)$.

When $0 \leq rank(A') - rank(A) \leq x$ and $0 \leq rank(B') - rank(B) \leq x$, the larger rank of A' and B' is within $rank(A) + x$. Then, the rank of C' is in the range $[rank(A), rank(A) + x + 1]$ (Lemma 4). On

```
1  joinRightRB(T_l, k, T_r) {
2    if (rank(T_l) = ⌊rank(T_r)/2⌋ × 2) then return node(T_l, ⟨k, red⟩, T_r);
3    else {
4      (L', ⟨k', c'⟩, R')=expose(T_l);
5      T' = node(L', ⟨k', c'⟩, joinRightRB(R', k, T_r));
6      if (c'=black) and (color(rc(T')) = color(rc(rc(T')))=red) then {
7        set rc(rc(T')) to be black;
8        return rotateLeft(T');
9      } else return T'; }}
10 joinRB(T_l, k, T_r, flag) { // flag denotes if k was a double-black node
11   if ⌊rank(T_l)/2⌋ > ⌊rank(T_r)/2⌋ then {
12     T' =joinRightRB(T_l, k, T_r);
13     if (color(T')=black) and (color(rc(T')) = color(rc(rc(T')))=red) then {
14         set rc(rc(T')) to be black;
15         return rotateLeft(T');
16     }
17     if (color(T')=red) and (color(rc(T'))=red) then
18       return node(lc(T'), ⟨k(T'), black⟩, rc(T'));
19     else return T';
20   } else if ⌊rank(T_r)/2⌋ > ⌊rank(T_l)/2⌋ then {
21     //symmetric to the previous case
22   } else {
23     if (flag) then
24       return node(T_l, ⟨k, black⟩, T_r);
25     else if (color(T_l)=black) and (color(T_r)=black)
26       return node(T_l, ⟨k, red⟩, T_r);
27     else return node(T_l, ⟨k, black⟩, T_r); }
28 }
```

Fig. 6. The *join* algorithm on RB trees. `joinLeftRB` is symmetric to `joinRightRB`.

the other hand, $C = node(A, k, B)$, $rank(C)$ is no smaller than $rank(A)+1$ (based on AVL invariants). Thus, $0 \leq rank(C) - rank(C') \leq x$ except for when $rank(C') = rank(A)$ and $rank(C) = rank(A)+1$. We will show that this is impossible.

First of all, $rank(C') = rank(A)$ means that $rank(A') = rank(A)$. Also, $rank(B')$ must be at most $rank(A) - 2$, otherwise the *join* will directly connect A' and B', and $rank(C')$ will be at least $rank(A) + 1$. Considering $rank(B') \geq rank(B)$, this means that A and B cannot be balanced, which leads to a contradiction.

This proves the increasing side of **submodularity rule**. We then prove the decreasing side. Both $rank(A)$ and $rank(B)$ are at most $rank(C) - 1$. Based on Lemma 3, *join*ing them back gets C' with rank at most $rank(C)$. □

4.2 RB Trees

In RB trees, $rank(T) = 2(\hat{h}(T) - 1)$ if T is black and $rank(T) = 2\hat{h}(T) - 1$ if T is red. Tarjan describes how to implement the *join* function for RB trees [64]. Here, we describe a variant that does not assume the roots are black (this is to bound the increase in rank in some *join*-based algorithms). The pseudocode is given in Figure 6. We store at every node its black height $\hat{h}(\cdot)$. One special issue for the RB trees is to decide the color of the pivot. Therefore, we introduce a special parameter

`flag` in the *join* function of RB trees, which indicates if we want to assign a specific color to the pivot in certain circumstances. This will be used in other *join*-based algorithms in Section 5.2. In particular, we will set the color for the middle node (the new node to hold the pivot) to be red by default. However, when `flag` is true, and when the two input trees are already balanced, we will set the color for the middle node to black. This is to avoid decreasing in rank when we re-*join* two black subtrees with their black parent.[6] In the *join*-based algorithms, we set this `flag` to be true only when the pivot we use is from a *double-black* node, i.e., when the node and all its children are black.

In the algorithm, the first case is when $\hat{h}(T_r) = \hat{h}(T_l)$. As mentioned above, if `flag` is true, we will still use it as a black node and directly concatenate the two input trees. This increases the rank of the input trees by at most 2. Otherwise, if both $root(T_r)$ and $root(T_l)$ are black, we create red $node(T_l, k, T_r)$. When either $root(T_r)$ or $root(T_l)$ is red, we create black $node(T_l, k, T_r)$.

The second case is when $\hat{h}(T_r) < \hat{h}(T_l) = \hat{h}$ (the third case is symmetric). Similarly to AVL trees, *join* follows the right spine of T_l until it finds a black node c for which $\hat{h}(c) = \hat{h}(T_r)$. It then creates a new red $node(c, k, T_r)$ to replace c. Since both c and T_r have the same height, the only invariant that can be violated is the red rule on the root of T_r, the new node, and its parent, which can all be red. In the worst case, we may have three red nodes in a row. This is fixed by a single rotation: if a black node v has $rc(v)$ and $rc(rc(v))$ both red, we turn $rc(rc(v))$ black and perform a single left rotation on v, turning the new node black, and then perform a single left rotation on v. The update is illustrated in Figure 7. The double- or triple-red issue is resolved after the rotation. The rotation, however, makes the new subtree root red, which can again violate the red rule between the root of the rotated tree and its parent. This imbalance can propagate to the root. If the original root of T_l is red, the algorithm may end up with a red root with a red child, in which case the root will be turned black, turning T_l rank from $2\hat{h} - 1$ to $2\hat{h}$. If the original root of T_l is black, the algorithm may end up with a red root with two black children, turning the rank of T_l from $2\hat{h} - 2$ to $2\hat{h} - 1$. In both cases, the rank of the result tree is at most $1 + rank(T_l)$.

We note that the rank of the output can increase the larger rank of the input trees by 2 only when the pivot was a double-black node, and the two input trees are balanced and both have black roots. This additional condition is to guarantee the **submodularity rule** for RB trees.

Lemma 4. *For two RB trees T_l and T_r, the RB join algorithm works correctly, runs with $O(|rank(T_l) - rank(T_r)|)$ work, and returns a tree satisfying the RB invariants and with rank at most* $2 + \max(rank(T_l), rank(T_r))$.

Proof. The base case where $h(T_l) = h(T_r)$ is straightforward. For symmetry, here we only prove the case when $h(T_l) > h(T_r)$. We prove the proposition by induction.

We first show the correctness. As shown in Figure 7, after appending T_r to T_l, if p is black, the rebalance has been done, and the height of each node stays unchanged. Thus, the RB tree is still valid. Otherwise, p is red, and p's parent g must be black. By applying a left rotation on p and g, we get a balanced RB tree rooted at p, except the root p is red. If p is the root of the whole tree, we change p's color to black, and the height of the whole tree increases by 1. The RB tree is still valid. Otherwise, if the current parent of p (originally g's parent) is black, the rebalance is done here. Otherwise, a similar rebalance is required on p and its current parent. Thus, finally we will either

[6] In particular, in some of our algorithms, we may need to get $t' = join(lc(t), t, rc(t))$, which ideally should just get t' the same as the original (sub-)tree t. However, consider when all tree nodes are black. If we call such *join* operation on all nodes in t, setting pivot to be red will cause the rank of t' to be only a half of the original tree t. This will affect the analysis of some *join*-based algorithms.

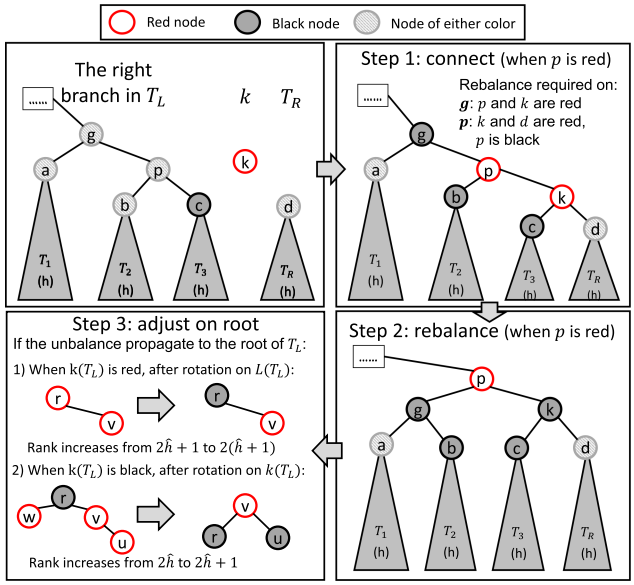


Fig. 7. An example of *join* on red-black trees. An example of *join* on red-black trees ($\hat{h} = \hat{h}(T_l) > \hat{h}(T_r)$). We follow the right spine of T_l until we find a black node with the same black height as T_r (i.e., c). Then a new red $node(c, k, T_r)$ is created, replacing c (Step 1). The only invariant that can be violated is when either c's previous parent p or T_r's root d is red. If so, a left rotation is performed at some black node. Step 2 shows the rebalance when p is red. The black height of the rotated subtree (now rooted at p) is the same as before ($h + 1$), but the parent of p might be red, requiring another rotation. If the red-rule violation propagates to the root, the root is either colored red, or rotated left (Step 3).

find the current node valid (current red node has a black parent), or reach the root, and change the color of root to be black. Thus, when we stop, we will always get a valid RB tree.

Since the algorithm only visits nodes on the path from the root to c, and only requires at most a single rotation per node on the path, the overall work for the algorithm is proportional to the depth of c in T_r. This in turn is no more than twice the difference in black height of the two trees since the black height decrements at least every two nodes along the path. This gives the stated work bounds.

For the rank, note that throughout the algorithm, before reaching the root, the black rule is never invalidated (or is fixed immediately), and the only invalidation occurs on the red rule. If the two input trees are originally balanced, the rank increases by at most 2. The only case that the rank increases by 2 is when k is from a double node, and both $root(T_r)$ and $root(T_l)$ are black.

If the two input trees are not balanced, the black height of the root does not change before the algorithm reaches the root (Step 3 in Figure 7). There are then three cases:

(1) The rotation does not propagate to the root, and thus the rank of the tree remains as $\max(\hat{h}(T_l), \hat{h}(T_r))$.
(2) (Step 3 Case 1) The original root color is red, and thus a double-red issue occurs at the root and its right child. In this case, the root is colored black. The black height of the tree increases by 1, but since the original root is red, the rank increases by only 1.
(3) (Step 3 Case 2) The original root color is black, but the double-red issue occurs at the root's child and grandchild. In this case, another rotation is applied as shown in Figure 7. The black height remains, but the root changed from black to red, increasing the rank by 1. □

Theorem 3. *RB trees are joinable.*

Proof. The RB invariant and the definition of RB trees' rank ensures the **empty rule** and **balancing rule** ($c_l = 1, c_u = 2$). Lemma 4 ensures the **cost rule** and **monotonicity rule**. We prove the **submodularity rule** as follows.

We start with the increasing side.

First note that the ranks of RB trees are always integers. For $C = node(A, k, B)$ and $C' = join(A', e, B')$. WLOG suppose $rank(A) \geq rank(B)$.

When $0 \leq rank(A') - rank(A) \leq x$ and $0 \leq rank(B') - rank(B) \leq x$, the larger rank of A' and B' is within $rank(A) + x$. Then the rank of C' is in the range $[rank(A), rank(A) + x + 2]$ (Lemma 4). On the other hand, $C = node(A, k, B)$; C is either $rank(A) + 1$ or $rank(A) + 2$ (because all ranks are integers). Thus, $0 \leq rank(C') - rank(C) \leq x$ except for the following cases.

(1) $rank(C') = rank(A) = r$.
(2) $rank(C') = r + 1$, but $rank(C) = r + 2$.
(3) $rank(C') = r + x + 2$, but $rank(C) = r + 1$.

We first show that case (1) is impossible.

First of all, $rank(A') \geq rank(A) = rank(C')$. On the other hand, $rank(C') \geq rank(A')$ because $C' = join(A', e, B')$. Therefore, $rank(A') = rank(C') = rank(A)$.

Also, $rank(B)$ is at least $r - 1$ because A and B are balanced. Considering $rank(B') \geq rank(B)$, $rank(B')$ is at least $r - 1$, but at most $rank(C') = r$.

If A' and B' are balanced, the rank of C' is at least $rank(A) + 1$, which leads to a contradiction.

If A' and B' are not balanced, their ranks differ by only 1. This is only possible when one of them has black height h with a black root, and the other one has black height $h - 1$ with a red root. However, the *join* algorithm results in double-red on the root's child and its grandchild. After fixing it the rank also increases by 1, which leads to a contradiction.

We then show that case 2 is impossible.

First of all, $rank(B)$ is at least $rank(A) - 1$ because A and B are balanced. Considering $rank(B') \geq rank(B)$, $rank(B')$ is at least $rank(A') - 1$.

If $rank(C) = r + 2$, C must be a double-black node. This means that $rank(A) = rank(B) = r$, and they are both black. $rank(A') \geq rank(A) = rank(C') - 1$. On the other hand, $rank(C') \geq rank(A')$ because $C' = join(A', k, B')$. Therefore, $rank(A') = rank(C')$ or $rank(A') = rank(C') - 1 = rank(A)$.

(1) If $rank(A') = rank(C')$, from the same statement of case (1), we can show this is impossible. This is because the *join* algorithm will always lead to the result where C' has rank larger than A'.
(2) If $rank(A') = rank(C') - 1 = rank(A)$, A' must be black since $rank(A)$ is even. In this case, $rank(B')$ is r, or $r + 1$.
    If $rank(A') = rank(B') = r$, B' is also black. In this case, the algorithm will result in $C' = r + 2$. This leads to a contradiction.
    If $rank(B') = r + 1$, B' is red but is still balanced with A'. In this case, the algorithm also results in C' with rank $r + 2$. This also leads to a contradiction.

Finally, we prove that case (3) is impossible.

$rank(C') = r + x + 2$ means that A', B', and C' are all black. Also, A' and B' must both have rank $r + x$ and black roots. This means that C' (and also C) is a double-black node. Thus, $rank(C)$ must be $r + 2$, which leads to a contradiction.

This proves the increasing side of the **submodularity rule**. Next, we look at the decreasing side. There are two cases.

```
1  joinRightWB(T_l, k, T_r) {
2    (l, k', c)=expose(T_l);
3    if (balance(|T_l|, |T_r|) then return node(T_l, k, T_r)); else {
4      T' = joinRightWB(c, k, T_r);
5      (l_1, k_1, r_1) = expose(T');
6      if like(|l|, |T'|) then return node(l, k', T');
7      else if (like(|l|, |l_1|)) and (like(|l| + |l_1|, r_1)) then
8        return rotateLeft(node(l, k', T'));
9      else return rotateLeft(node(l, k', rotateRight(T'))); }}

10 joinWB(T_l, k, T_r) {
11   if heavy(T_l, T_r) then return joinRightWB(T_l, k, T_r);
12   else if heavy(T_r, T_l) then return joinLeftWB(T_l, k, T_r);
13   else return node(T_l, k, T_r); }
```

Fig. 8. The *join* algorithm on weight-balanced trees – `joinLeftWB` is symmetric to `joinRightWB`.

(1) Both $rank(A') < rank(A)$ and $rank(B') < rank(B)$ hold. They have to decrease by at least one because the rank of an RB tree is always an integer. First of all, we know that $rank(C') \leq \max(rank(A'), rank(B')) + 2$.

If $rank(C') = \max(rank(A'), rank(B')) + 2$, C' must be a double-black node. $rank(C)$ is $rank(A) + 2$ (also $rank(B) + 2$). Also, A, B, C, A', B', and C' are all black. Then, $rank(A')$ and $rank(B')$ can be at most $rank(A) - 2$ and $rank(B) - 2$, respectively. *join*ing A' and B' increase the maximum rank by at most 2. Therefore, $rank(C')$ is no more than $rank(C)$ holds.

If $rank(C') \leq \max(rank(A'), rank(B')) + 1$, *join*ing them back results in an output tree of rank at most $\max(rank(A'), rank(B')) + 1$. This proves $rank(C') \leq rank(C)$.

(2) Either $rank(A') = rank(A)$ or $rank(B') = rank(B)$. WLOG assume $rank(A') = rank(A)$ and $rank(B') \leq rank(B)$. There are three cases.
   (a) A (so is A') is black with rank $2h$, and C is red with rank $2h + 1$. $rank(B') \leq rank(B) = 2h$. Therefore, $rank(C') \leq rank(A') + 1 = rank(C)$.
   (b) A (so is A') is black with rank $2h$, and C is black with rank $2h + 2$. $rank(B') \leq rank(B) \leq 2h + 1$. When $rank(B') \leq 2h$, $rank(C') \leq rank(A') + 2 = rank(C)$. When $rank(B') = 2h + 1$, B is red and C' is not a double-black node. Therefore, we also get $rank(C') \leq rank(B') + 1 = rank(C)$.
   (c) A (so is A') is red with rank $2h + 1$, and C is black with rank $2h + 2$. $rank(B') \leq rank(B) \leq 2h + 1$. Therefore, $rank(C') \leq rank(A') + 1 = rank(C)$.

In summary, the **submodularity rule** holds for RB trees. RB trees are strongly joinable. □

4.3 WB Trees

For WB trees, $rank(T) = \log_2(w(T)) - 1$. We store the weight of each subtree at every node. The algorithm for joining two WB trees is similar to that of AVL trees and RB trees. The pseudocode is shown in Figure 8. The *like* function in the code returns true if the two input tree sizes are balanced based on the factor of α, and false otherwise. If T_l and T_r have like weights, the algorithm returns a new $node(T_l, k, T_r)$. Suppose $|T_r| \leq |T_l|$; the algorithm follows the right branch of T_l until it reaches a node c with like weight to T_r. It then creates a new $node(c, r, T_r)$ replacing c. The new node will have weight greater than c and therefore could imbalance the weight of c's ancestors. This can be fixed with a single or double rotation (as shown in Figure 9) at each node assuming α is within the bounds given in Section 2.

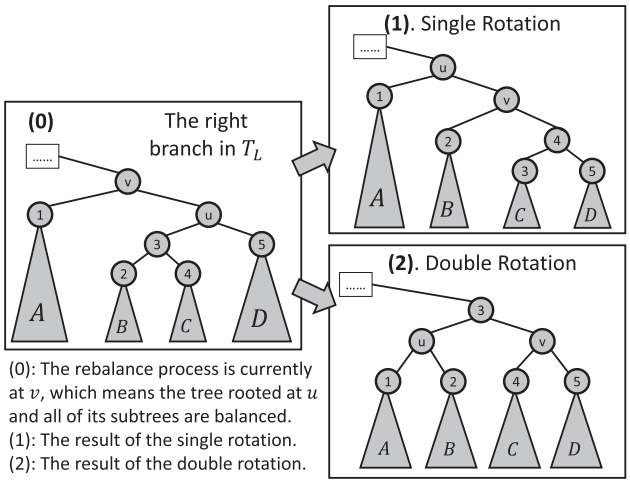


Fig. 9. An illustration of single and double rotations possibly needed to rebalance weight-balanced trees – In the figure the subtree rooted at u has become heavier due to joining in T_l and its parent v now violates the balance invariant.

Lemma 5. *For two α WB trees T_l and T_r and $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$, the WB join algorithm works correctly, runs with $O(|\log(w(T_l)/w(T_r))|)$ work, and returns a tree satisfying the α WB invariant and with rank at most $1 + \max(rank(T_l), rank(T_r))$.*

The proof can be found in [13].

Notice that this upper bound is the same as the restriction on α to yield a valid WB tree when inserting a single node. Then, we can induce that when the rebalance process reaches the root, the new WB tree is valid. The proof is intuitively similar as the proof stated in [17, 52], which proved that when $\frac{2}{11} \leq \alpha \leq 1 - \frac{1}{\sqrt{2}}$, the rotation will rebalance the tree after one single insertion. In fact, in our *join* algorithm, the "inserted" subtree must be along the left or right spine, which actually makes the analysis easier.

Theorem 4. *WB trees are strongly joinable when $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$.*

Proof. The WB invariant and the definition of WB's rank ensures the **empty rule** and **balancing rule** ($c_l = \log_{(1-\alpha)} 2$, $c_u = \log_\alpha 2$). Lemma 5 ensures the **cost rule** and the **monotonicity rule**.

For the **submodularity rule**, note that $rank(C) = \log_2(w(A) + w(B))$, and $rank(C') = \log_2(w(A') + w(B'))$. When either A or B changes their weight by more than $\log_2 x$, obviously the total weight of C will not increase or decrease by a factor of $\log_2 x$. □

4.4 Treaps

For treaps $rank(T) = \log_2(w(T)) - 1$. The treap *join* algorithm (as in Figure 10) first picks the key with the highest priority among k, $k(T_l)$, and $k(T_r)$ as the root. If k is the root, then we can return $node(T_l, k, T_r)$. Otherwise, WLOG, assume $k(T_l)$ has a higher priority. In this case, $k(T_l)$ will be the root of the result, $lc(T_l)$ will be the left tree, and $rc(T_l)$, k, and T_r will form the right tree. Thus, *join* recursively calls itself on $rc(T_l)$, k, and T_r and uses the result as $k(T_l)$'s right child. When $k(T_r)$ has a higher priority, the case is symmetric. The cost of *join* is therefore the depth of the key k in the resulting tree (each recursive call pushes it down one level). In treaps, the shape of the result tree, and hence the depth of k, depend only on the keys and priorities and not the history. Specifically,

```
1 joinTreap(T_l, k, T_r) {
2   if prior(k, k_1) and prior(k, k_2) then return node(T_l, k, T_r) else {
3     (l_1, k_1, r_1)=expose(T_l);
4     (l_2, k_2, r_2)=expose(T_r);
5     if prior(k_1, k_2) then return node(l_1, k_1, joinTreap(r_1, k, T_r));
6     else return node(joinTreap(T_l, k, l_2), k_2, r_2); }}
```

Fig. 10. The *join* algorithm on treaps. prior(k_1, k_2) decides if the node k_1 has a higher priority than k_2.

if a key has the t-th highest priority among the keys, then its expected depth in a treap is $O(\log t)$ (also *whp*). If it is the highest priority, for example, then it remains at the root.

Lemma 6. *For two treaps T_l and T_r, if the priority of the pivot k is the t-th highest among all keys in $T_l \cup \{k\} \cup T_r$, the treap join algorithm works correctly, runs with $O(\log t + 1)$ work in expectation and* whp*, and returns a tree satisfying the treap invariant with rank at most $1 + \max(rank(T_l), rank(T_r))$.*

Theorem 5. *Treaps are weakly joinable.*

Proof. The **empty rule** and **monotonicity rule** hold from the definition of rank. The **submodularity rule** holds for the same reason as the WB tree.

For the **relaxed balancing rule**, note that the weight of the tree shrinks by a factor of 1/3 to 2/3 with probability 1/3. This means that going down from a parent to a child, the rank of a node decreases by a constant between $\log_2 3$ and $\log_2 3/2$ with probability 1/3. This proves the **relaxed balancing rule**.

For **weak cost rule**, note that the cost of *join* is at most the height of the larger input tree, which is $O(rank(T))$ *whp*. □

We also present Lemma 7, which is useful in the proofs in a later section. Recall that we use $d(v)$ to denote the size of a rank root v's *rank cluster*. In treaps, each rank cluster is a chain. This is because if two children of one node v are both in layer i, the weight of v is more than 2^{i+1}, meaning that v should be layer $i + 1$. This means that for a treap node v, $d(v)$ is the number of generations to take from v to reach a node u with rank at most $rank(v) - 1$. The lemma below means that the expected size of a rank cluster $\mathbb{E}[d(v)]$ is also a constant.

Lemma 7. *If v is a rank root in a treap, $d(v)$ is less than a constant in expectation.*

Proof. Consider a rank(k)-root v that has weight $N \in [2^k, 2^{k+1})$. The probability that $d(v) \geq 2$ is equal to the probability that one of its grandchildren has weight at least 2^k. This probability P is

$$P = \frac{1}{2^k} \sum_{i=2^k+1}^{N} \frac{i - 2^k}{i} \tag{1}$$

$$\leq \frac{1}{2^k} \sum_{i=2^k+1}^{2^{k+1}} \frac{i - 2^k}{i} \tag{2}$$

$$\approx 1 - \ln 2. \tag{3}$$

We denote $1 - \ln 2$ as p_c. Similarly, the probability that $d(v) \geq 4$ should be less than p_c^2, and the probability shrinks geometrically as $d(v)$ increase. Thus, the expected value of $d(v)$ is a constant. □

Finally, we prove that Lemma 1 also holds for treaps.

Split

```
1  split(T, k) {
2    if T = ∅ then
3      return (∅, false, ∅);
4    (L, k′, R) = expose(T);
5    if k = k′ then return (L, true, R);
6    if k < k′ then {
7      (T_L, b, T_R) = split(L, k);
8      return (T_L, b, join(T_R, k′, R)); }
9    (T_L, b, T_R) = split(R, k);
10   return (join(L, k′, T_L), b, T_R); } }
```

join2

```
1  split_last(T) {
2    (L, k, R) = expose(T);
3    if R = ∅ then return(L, k);
4    (T′, k′) = split_last(R);
5    return (join(L, k, T′), k′); }
6  join2(T_l, T_r) {
7    if T_l = ∅ then return T_r;
8    (T′, k) = split_last(T_l);
9    return join(T′, k, T_r);
10 }
```

Fig. 11. *split* and *join2* algorithms – They are both independent of balancing schemes.

Lemma 8. *For a treap with size N, $s_i \leq \frac{N}{2^{\lfloor i/2 \rfloor}}$.*

This holds trivially from the definition of the rank of a treap, which is $\log_2 N$ for a treap of size N.

As a summary of this section, we present the following theorem.

Theorem 6. *AVL, RB, and WB trees are strongly joinable, and treaps are weakly joinable.*

5 ALGORITHMS USING *JOIN*

The *join* function, as a subroutine, has been used and studied by many researchers and programmers to implement more general set operations. In this section, we describe algorithms for various functions that use just *join* for rebalancing. The algorithms are generic across balancing schemes. The pseudocodes for the algorithms in this section are shown in Figure 13. Beyond *join*, the only access to the trees we make use of is through *expose*, which returns the left child, root entry, and the right child. This can be done by simply reading the root. The set-set operations, including *union*, *intersection*, and *difference*, are *efficient* in work, which means the work of the algorithm is asymptotically the same as the best known sequential algorithm. In particular, the cost of the *join*-based algorithms are optimal in the number of comparisons performed (see more details below). The pseudocode for all the algorithms introduced in this section is presented in Figures 11, 13, and 15. We note that for RB trees, for every invocation of *join*, we need to give an extra flag to indicate if the pivot of *join* is a double-black node (defined in Section 4.2). For simplicity, this is not shown in the code given. We note that this is used only to ensure the desired theoretical bound, and removing this condition does not affect the correctness.

As mentioned, all the *join*-based algorithms use *join* as a black box. As long as the *join* algorithm is implemented correctly as defined, the *join*-based algorithms can behave as expected. Also, the algorithms themselves do not require the *rank* function. However, to prove the efficiency in work and span, we need more careful design and use of *rank* functions.

We present a summary of the cost of all *join*-based algorithms discussed in the article in Table 2.

5.1 Two Helper Functions: *split* and *join2*

We start with presenting two helper functions *split* and *join2*. For a BST T and key k, $split(T, k)$ returns a triple (T_l, b, T_r), where T_l (T_r) is a tree containing all keys in T that are smaller (larger) than k, and b is a flag indicating whether $k \in T$. For two binary trees T_l and T_r, not necessarily BSTs, $join2(T_l, T_r)$ returns a binary tree for which the in-order values are the concatenation of the in-order values of the binary trees T_l and T_r (the same as *join* but without the middle key). For BSTs, all keys in T_l have to be less than keys in T_r.

Table 2. The Core *join*-based Algorithms and Their Asymptotic Costs – The Cost is Given Under the Assumption That All Parameter Functions Take Constant Time to Return

Function	***Work***	***Span***
insert, *delete*, *update*, *find*, *first*, *last*, *range*, *split*, *join2*, *previous*, *next*, *rank*, *select*, *up_to*, *down_to*	$O(\log n)$	$O(\log n)$
union, *intersection*, *difference*	$O\left(m \log\left(\frac{n}{m}+1\right)\right)$	$O(\log n \log m)$
map, *reduce*, *map_reduce*, *to_array*	$O(n)$	$O(\log n)$
build, *filter*	$O(n)$	$O(\log^2 n)$

For functions with two input trees (*union*, *intersection* and *difference*), n is the size of the larger input, and m of the smaller.

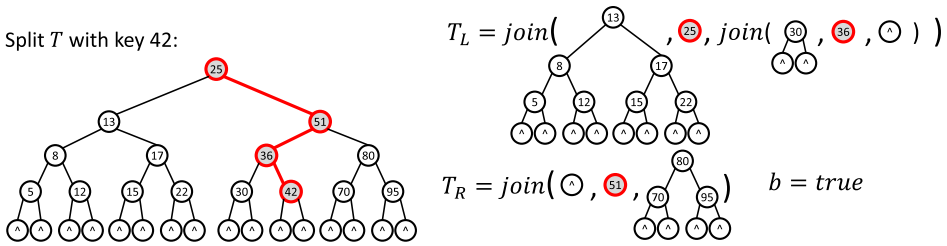


Fig. 12. An example of *split* in a BST with key 42. We first search for 42 in the tree and split the tree by the searching path, then use *join* to combine trees on the left and on the right, respectively, bottom-top.

Although both sequential, these two functions, along with the *join* function, are essential to parallelize other algorithms. Intuitively, when processing a tree in parallel, we recurse on two subcomponents of the tree in parallel by *split*ing the tree by some key. After the recursions return, we combine the results of the left and right parts, with or without the middle key, using *join* or *join2*. Because of the balance of the tree, this framework usually gives high parallelism with low span (e.g., poly-logarithmic).

Split. As mentioned above, $split(T, k)$ splits a BST T by a key k into T_l and T_r, along with a bit b indicating if $k \in T$. Intuitively, the *split* algorithm first searches for k in T, splitting the tree along the path into three parts: keys to the left of the path, k itself (if it exists), and keys to the right. Then by applying *join*, the algorithm merges all the subtrees on the left side (using keys on the path as intermediate nodes) from bottom to top to form T_l, and merges the right parts to form T_r. Writing the code in a recursive manner, this algorithm first determines if k falls in the left (right) subtree, or is exactly the root. If it is the root, then the algorithm directly returns the left and the right subtrees as the two return trees and *true* as the bit b. Otherwise, WLOG, suppose k falls in the left subtree. The algorithm further *split* the left subtree into T_L and T_R with the return bit b'. Then, the return bit $b = b'$, the T_l in the final result will be T_L, and T_r means to *join* T_R with the original right subtree by the original root. Figure 12 gives an example.

The cost of the algorithm is proportional to the rank of the tree. Intuitively (and informally), this is because the cost of *split* is the sum of the cost of a sequence of *join* algorithms, which is $O(|r(T_L) - r(T_R)|)$. For the chain of trees involved, the middle terms cancel out, and the total cost is then $O(r(T))$. We summarize and prove the cost of the *split* algorithm in the following theorem.

Theorem 7. *The work of* $split(T, k)$ *is* $O(h(T))$ *for all strongly joinable trees and treaps. The two resulting trees* T_l *and* T_r *will have rank at most* $rank(T) + c_0$ *for some constant* c_0.

Union

```
union(T1,T2) {
  if T1 = ∅ then return T2;
  if T2 = ∅ then return T1;
  (L2,k2,R2) = expose(T2);
  (L1,b,R1) = split(T1,k2);
  Tl = union(L1,L2) ||
       Tr = union(R1,R2);
  return join(Tl,k2,Tr); }
```

Intersection

```
intersect(T1,T2) {
  if T1 = ∅ then return ∅;
  if T2 = ∅ then return ∅;
  (L2,k2,R2) = expose(T2);
  (L1,b,R1) = split(T1,k2);
  Tl = intersect(L1,L2) ||
       Tr = intersect(R1,R2);
  if b then return join(Tl,k2,Tr);
  else return join2(Tl,Tr); }
```

Difference

```
difference(T1,T2) {
  if T1 = ∅ then ∅;
  if T2 = ∅ then T1;
  (L2,k2,R2) = expose(T2);
  (L1,b,R1) = split(T1,k2);
  Tl = difference(L1,L2) || Tr = difference(R1,R2);
  return join2(Tl,Tr); }
```

Fig. 13. *join*-based algorithms for set-set operations. They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

Proof. We only consider the work of joining all subtrees on the left side. The other side is symmetric. Suppose we have l subtrees on the left side, denoted from bottom to top as $T_1, T_2, \ldots T_l$. As stated above, we consecutively join T_1 and T_2 returning T_2', then join T_2' with T_3 returning T_3' and so forth, until all trees are merged. The overall work of *split* is the sum of the cost of $l-1$ *join* functions. One observation is that, the pivot of the *join* of T_{i-1}' and T_i, denoted as e_i, used to be T_i's parent. Meanwhile, T_{i-1}' is a subset of T_i's original sibling, denoted as X_i. We use $T(e_i) = node(X_i, e_i, T_i)$ to denote the original subtree rooted at e_i in the input.

We first prove by induction that computing $T_i' = join(T_{i-1}', e_i, T_i)$ gets T_i' with rank no more than $rank(X_{i+1})$.

From the induction hypothesis, we know that $rank(T_{i-1}') \leq r(X_i)$. Considering $T_i' = join(T_{i-1}', e_i, T_i)$ and $T(e_i) = node(X_i, e_i, T_i)$, from the **submodularity rule**, we can get $rank(T_i') \leq rank(T(e_i))$. Considering $T(e_i)$ is a subtree in X_{i+1}, we have

$$rank(T_i') \leq rank(T(e_i)) \leq rank(X_{i+1}).$$

We next prove the cost. The cost of the i-th *join* is $W_i \leq c|rank(T_i) - rank(T_{i-1}')|$. Note that $rank(T_{i-1}') \leq rank(X_i) \leq rank(T_i) + c_u - c_l$. Also, note that T_i' is achieved by joining T_i and another tree. Therefore, $rank(T_i) - rank(T_{i-1}') \leq rank(T_i') - rank(T_{i-1}')$.

This means that either $W_i = c(rank(T_i') - rank(T_{i-1}'))$, or W_i is a constant no more than $c_2 = c \cdot (c_u - c_l)$. Therefore, $W_i \leq c(rank(T_i') - rank(T_{i-1}') + 2c_2)$.

$$\begin{aligned}\sum_{i=1}^{h(T)} W_i &\leq \sum_{i=1}^{h(T)} c(rank(T_i') - rank(T_{i-1}') + 2c_2) \\ &\leq 2c \cdot c_2 \cdot h(T) + rank(T_i') \\ &\leq O(h(T)) + rank(T(e_{h(T)})) \leq O(h(T)).\end{aligned}$$

For treaps, each *join* uses the key with the highest priority since the key is always on an upper level. Hence, by Lemma 6, the complexity of each *join* is $O(1)$ and the work of split is at most $O(h(T))$. Obviously for treaps we have $rank(T_l)$ and $rank(T_r)$ at most $rank(T)$. □

Join2. As stated above, the *join2* function is defined similar to *join* without the middle entry. The *join2* algorithm first choose one of the input trees, and extract its last (if it is T_l) or first (if it is T_r) element k. The two cases take the same asymptotical cost. The extracting process is similar to the *split* algorithm. The algorithm then uses k as the pivot to *join* the two trees. In the code shown in Figure 11, the *split_last* algorithm first finds the last element k (by following the right spine) in T_l and on the way back to root, *join*s the subtrees along the path. We denote the result of dropping k in T_L as T'. Then, $join\ (T', k, T_r)$ is the result of *join2*. Unlike *join*, the work of *join2* is proportional to the rank of both trees since both *split* and *join* take at most logarithmic work.

THEOREM 8. *The work of $T = join2(T_l, T_r)$ is $O(r(T_l) + r(T_r))$ for all joinable trees. Furthermore, $rank(T) \leq \max(rank(T_l), rank(T_r)) + c'$, where c' is defined in Theorem 6.*

PROOF. The cost bound holds because *split_last* and *join* both take work asymptotically no more than the larger tree rank. We next prove the range of $rank(T)$. First of all, splitting the last from T_l only decreases its rank (Theorem 7). Therefore, based on Theorem 6, $T = join\ (T', k, T_r)$ has rank no more than $\max(rank(T_r), rank(T_l)) + c'$. This proves the theorem. □

5.2 Set Functions Using *join*

In this section, we will present *join*-based algorithms on BSTs for set-set functions, including *union*, *intersection*, and *difference*. Many other set operations, such as symmetric difference, can be implemented by a combination of *union*, *intersection*, and *difference* with no extra asymptotical work. We will start with presenting some background of these algorithms, and then explain in detail about the *join*-based algorithms. Finally, we show the proof of their cost bound.

Background. The parallel set functions are particularly useful when using parallel machines since they can support parallel bulk updates. As mentioned, although supporting efficient algorithms for basic tree operations (e.g., insertion and deletion) are rather straightforward, it is more difficult to implement efficient bulk operations. This is more challenging considering parallelism and dealing with multiple balancing schemes. For example, combining two ordered sets of size n and $m \leq n$ in the format of two arrays would take work $O(m + n)$ using the standard merging algorithm in the merge sort algorithm. In this case, even inserting a single element into a set of size n will have linear cost. Another simple implementation is to store both sets as balanced trees, and insert the elements in the smaller tree into the larger one, costing $O(m \log n)$ work. It overcomes the issue of redundant scanning and copying, because many subtrees in the larger tree remain untouched. However, this results in $O(n \log n)$ work for combining two ordered sets of the same size n, while it is easy to make it $O(n)$ by arrays. This is because the algorithm fails to make use of the ordering in the smaller tree.

The lower bound for comparison-based algorithms for *union*, *intersection*, and *difference* for inputs of size n and $m \leq n$, and returning an ordered structure,[7] is $\log_2 \binom{m+n}{n} = \Theta(m \log(\frac{n}{m} + 1))$, which is the number of possible ways n keys can be interleaved with m keys [37]. Brown and Tarjan first matched these bounds, asymptotically, using a sequential algorithm based on RB trees [22]. Although designed for merging, the algorithm can be adapted for *union*, *intersection*, and *difference* with the same bounds. The bound is interesting since it shows that implementing insertion with

[7]By "ordered structure," we mean any data structure that can output elements in sorted order without any further comparisons (e.g., a sorted array, or a binary search tree).

union, or deletion with difference, is asymptotically efficient ($O(\log n)$ time), as is taking the union of two equal sized sets ($O(n)$ time). However, the Brown and Tarjan algorithm is complicated, only works on RB trees, and is completely sequential.

Adams later described very elegant algorithms for union, intersection, and difference, as well as other functions based on *join* [2, 3]. Adams' algorithms were proposed in an international competition for the Standard ML community, which is about implementations on "set of integers." Prizes were awarded in two categories: fastest algorithm, and most elegant yet still efficient program. Adams won the elegance award, while his algorithm is almost as fast as the fastest program for very large sets, and was faster for smaller sets. Because of the elegance of the algorithm, at least three libraries use Adams' algorithms for their implementation of ordered sets and tables (Haskell [48] and MIT/GNU Scheme, and SML). The idea of Adams' algorithm enlightens our parallel *join*-based set algorithms and the implementation in the PAM library, for which the sequential version on WB trees is exactly the same as Adams' algorithm.

Although only considering WB trees, Adams' algorithms actually show that in principle all balance criteria for search trees can be captured by the single function *join*. As long as a valid *join* algorithm on a certain balancing scheme is provided, the correctness of the *join*-based set operations can be guaranteed on the corresponding balancing scheme.

Surprisingly, however, there have been almost no results on bounding the work (time) of Adams' algorithms, in general or on specific tree types. Adams informally argues that his algorithms take $O(n + m)$ work for WB tree, but that is a very loose bound. Blelloch and Reid-Miller later show that similar algorithms for treaps [16] are optimal for work (i.e., $\Theta(m \log(\frac{n}{m} + 1))$), and are also parallel. Their algorithms, however, are specific for treaps. The problem with bounding the work of Adams' algorithms, is that just bounding the time of *split*, *join*, and *join2* with logarithmic costs is not sufficient.[8] One needs additional properties of the trees. As a result, there is no tight bound even for Adams' original algorithms on WB trees, not to mention other balancing schemes.

Our work gives the first work-optimal bounds for the *join*-based algorithms for all four balancing schemes in Section 2. We show that with the *join* algorithms in Section 4, we achieve asymptotically optimal bounds on work for the set operations. These bounds hold when either input tree is larger (this was surprising to us). Furthermore, the algorithms have $O(\log n \log m)$ span, and hence are highly parallel.

Algorithms. *union*(T_1, T_2) takes two BSTs and returns a BST that contains the union of all keys. The algorithm uses a classic divide-and-conquer strategy, which is parallel. At each level of recursion, T_1 is split by $k(T_2)$, breaking T_1 into three parts: one with all keys smaller than $k(T_2)$ (denoted as L_1), one in the middle either of only one key equal to $k(T_2)$ (when $k(T_2) \in T_1$) or empty (when $k(T_2) \notin T_1$), and the third one with all keys larger than $k(T_2)$ (denoted as R_1). Then two recursive calls to *union* are made in parallel. One unions $lc(T_2)$ with L_1, returning T_l, and the other one unions $rc(T_2)$ with R_1, returning T_r. Finally, the algorithm returns *join* $(T_l, k(T_2), T_r)$, which is valid since $k(T_2)$ is greater than all keys in T_l are less than all keys in T_r.

The functions *intersection* (T_1, T_2) and *difference* (T_1, T_2) take the intersection and difference of the keys in their sets, respectively. The algorithms are similar to *union* in that they use one tree to split the other. However, the method for joining and the base cases are different. For *intersection*, *join2* is used instead of *join* if the root of the first *is not* found in the second. Accordingly, the base case for the *intersection* algorithm is to return an empty set when either set is empty. For *difference*,

[8]Bounding the cost of *join*, *split*, and *join2* by the logarithm of the *smaller tree* is probably sufficient, but implementing a data structure with such bounds is very much more complicated.

Table 3. Descriptions of Notations Used in Section 5.3

Notation	Description
T_p	The pivot tree
T_d	The decomposed tree
n	$\max(\|T_p\|, \|T_d\|)$
m	$\min(\|T_p\|, \|T_d\|)$
$T_p(v), v \in T_p$	The subtree rooted at v in T_p
$T_d(v), v \in T_p$	The tree from T_d that v splits[9]
s_i	The number of nodes in layer i (Definition 4)
v_{kj}	The j-th node on layer k in T_p
$d(v)$	The size of the rank cluster of a rank root v (Definition 5)

join2 is used anyway because $k(T_2)$ should be excluded in the result tree. The base cases are also different in the obvious way.

The cost of the algorithms described above can be summarized in the following theorem.

Theorem 9. *For all strongly joinable trees (and treaps), the work and span of the algorithm (as shown in Figure 13) of union, intersection, or difference on two balanced BSTs of sizes m and n ($n \geq m$) is $O(m \log(\frac{n}{m} + 1))$ (in expectation for treaps) and $O(\log n \log m)$, respectively (w.h.p. for treaps).*

The work bound for these algorithms is optimal in the comparison-based model. In particular, considering all possible interleavings, the minimum number of comparisons required to distinguish them is $\log \binom{m+n}{n} = \Theta(m \log(\frac{n}{m} + 1))$ [37]. A generic proof of Theorem 9 working for four balancing schemes will be shown in the next section. The span of these algorithms can be reduced to $O(\log m)$ for WB trees even on the binary-forking model [15] by doing a more complicated divide-and-conquer strategy.

5.3 The Proof of Theorem 9

We now prove Theorem 9, for all the joinable trees and all three set algorithms (*union*, *intersection*, *difference*) from Figure 13.

For this purpose, we make two observations. The first is that all the work for the algorithms can be accounted for within a constant factor by considering just the work done by the *split*s and the *join*s (or *join2*s), which we refer to as *split work* and *join work*, respectively. This is because the work done between each split and join is constant. The second observation is that the split work is identical among the three set algorithms. This is because the control flow of the three algorithms is the same on the way down the recursion when doing *split*s—the algorithms only differ in what they do at the base case and on the way up the recursion when they join.

Given these two observations, we prove the bounds on work by first showing that the join work is asymptotically at most as large as the split work (by showing that this is true at every node of the recursion for all three algorithms), and then showing that the split work for *union* (and hence, the others) satisfies our claimed bounds.

We start with some notation, which is summarized in Table 3. In the three algorithms, the first tree (T_1) is split by the keys in the second tree (T_2). We therefore call the first tree the *decomposed tree* and the second the *pivot tree*, denoted as T_d and T_p, respectively. The tree that is returned is denoted as T. Since our proof works for either tree being larger, we use $m = \min(|T_p|, |T_d|)$ and $n = \max(|T_p|, |T_d|)$. We denote the subtree rooted at $v \in T_p$ as $T_p(v)$, and the tree of keys from T_d

[9]The nodes in $T_d(v)$ form a subset of T_d, but not necessarily a subtree. See details later.

that v splits as $T_d(v)$ (i.e., *split* $(v, T_d(v))$ is called at some point in the algorithm). For $v \in T_p$, we refer to $|T_d(v)|$ as its *splitting size*.

Figure 14(a) illustrates the pivot tree with the splitting size annotated on each node. Since *split* has logarithmic work, we have

$$\text{split work } = O\left(\sum_{v \in T_p} (\log |T_d(v)| + 1)\right),^{10}$$

which we analyze in Theorem 11. We first, however, show that the join work is bounded by the split work. We use the following Lemma.

Lemma 9. *For* $T = union\ (T_p, T_d)$ *on strongly joinable trees, then* $\max(rank(T_p), rank(T_d)) \leq rank(T) \leq rank(T_p) + rank(T_d)$.

Proof. We prove it by induction on the tree size. For small trees, this conclusion obviously holds. Note that T_d will be split up into two trees T_l and T_r, with rank at most $r = rank(T_d)$ (Theorem 7). $lc(T_p)$ and $rc(T_p)$ will take *union* with either T_l or T_r, i.e., $L = union(lc(T_p), T_l)$ and $R = union(lc(T_p), T_l)$. Because of the induction hypothesis, $0 \leq rank(L) \leq rank(lc(T_p)) + rank(T_l) \leq rank(lc(T_p)) + r$, and similarly $0 \leq rank(R) \leq rank(lc(T_p)) + r$. From the **submodularity rule**, joining them increases the rank of T_p by at least 0 and at most $rank(T_d)$. □

Theorem 10. *For each function call to union, intersection, or difference on strongly joinable trees and treaps* $T_p(v)$ *and* $T_d(v)$*, the work to do the join (or join2) is asymptotically no more than the work to do the split.*

Proof. For *intersection* or *difference*, the cost of *join* (or *join2*) is $O(\log(|T|))$, where T is the result tree. Notice that *difference* returns the keys in $T_d \backslash T_p$. Thus, for both *intersection* and *difference* we have $T \subseteq T_d$. The join work is $O(\log(|T|))$, which is no more than $O(\log(|T_d|))$ (the split work).

For *union*, if $rank(T_p) \leq rank(T_d)$, the *join* will cost $O(rank(T_d))$, which is no more than the split work.

Consider $rank(T_p) > rank(T_d)$ for strongly joinable trees. The rank of $lc(T_p)$ and $rc(T_p)$, which are used in the recursive calls, is at least $rank(T_p) - c_u$. Using Lemma 9, the rank of the two trees returned by the two recursive calls will be at least $(rank(T_p) - c_u)$ and at most $(rank(T_p) + rank(T_d))$, and differ by at most $O(rank(T_d)) = O(\log |T_d|)$. Thus, the join cost is $O(\log |T_d|)$, which is asymptotically no more than the split work.

Consider $rank(T_p) > rank(T_d)$ for treaps. If $rank(T_p) > rank(T_d)$, then $|T_p| \geq |T_d|$. The root of T_p has the highest priority among all $|T_p|$ keys, so on expectation it takes at most the $\frac{|T_p|+|T_d|}{|T_p|} \leq 2$-th place among all the $|T_d| + |T_p|$ keys. From Lemma 6, we know that the cost on expectation is $\mathbb{E}[\log t] + 1 \leq \log \mathbb{E}[t] + 1 \leq \log 2 + 1$, which is a constant. □

This implies the total join work is asymptotically bounded by the split work.

We now analyze the split work. We do this by layering the pivot tree starting at the leaves and going to the root and such that nodes in a layer are not ancestors of each other. We use the definition in Section 3. We define layers based on the ranks and denote the number of rank(i)-root nodes as s_i. Lemma 1 shows that s_i shrinks geometrically for joinable trees, which helps us prove our bound on the split work. Figure 14(b) shows an example of the layers of an AVL tree on the two input trees of the *join*-based set functions.

Lemma 10. *For any ancestor-free set* $V \subseteq T_p$, $\sum_{v \in V} |T_d(v)| \leq |T_d|$.

[10] When $|T_d(v)| = 0$, we set $log|T_d(v)|$ as 0.

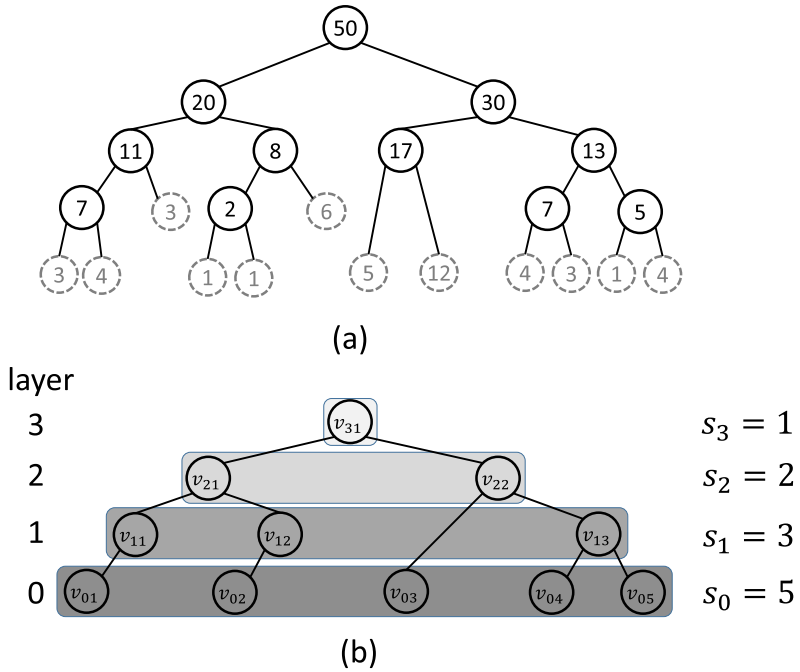


Fig. 14. An illustration of splitting tree and layers. The tree in (a) is T_p, and the dashed circles are the exterior nodes. The numbers on the nodes are the sizes of the tree from T_d to be split by this node, i.e., the "splitting size" $|T_d(v)|$. (b) is an illustration of layers on an AVL tree.

The proof of this lemma is straightforward.

Not all nodes are rank roots. However, Property 7 shows that each rank cluster attached to a rank root contains only a constant number of nodes, and they are all the rank root's descendants.

By applying Lemma 1 and Property 7, we prove the split work. In the following proof, we denote v_{kj} as the j-th node in layer k.

Theorem 11. *The split work in union, intersection, and difference on two joinable trees of size m and n is $O(m \log(\frac{n}{m} + 1))$.*

Proof. The total work of *split* is the sum of the log of all the splitting sizes on the pivot tree $O(\sum_{v \in T_p} \log(|T_d(v)| + 1))$. Denote l as the number of layers in the tree. Also, notice that in the pivot tree, in each layer there are at most $|T_d|$ nodes with $|T_d(v_{kj})| > 0$. Since those nodes with splitting sizes of 0 will not cost any work, we can assume $s_i \leq |T_d|$. We calculate the dominant term $\sum_{v \in T_p} \log(|T_d(v)| + 1)$ in the complexity by summing the work across layers. We first only consider all rank roots.

$$\sum_{k=0}^{l} \sum_{j=1}^{s_k} \log\left(|T_d(v_{kj})| + 1\right) \leq \sum_{k=0}^{l} s_k \log\left(\frac{\sum_j |T_d(v_{kj})| + 1}{s_k}\right)$$
$$= \sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k} + 1\right).$$

We split it into two cases. If $|T_d| \geq |T_p|$, $\frac{|T_d|}{s_k}$ always dominates 1. We have

$$\sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k}+1\right) = \sum_{k=0}^{l} s_k \log\left(\frac{n}{s_k}+1\right) \tag{4}$$

$$\leq \sum_{k=0}^{l} \frac{m}{2^{\lfloor k/c \rfloor}} \log\left(\frac{n}{m/2^{\lfloor k/c \rfloor}}+1\right) \tag{5}$$

$$\leq c\sum_{k=0}^{l/c} \frac{m}{2^k} \log \frac{n}{m/2^k}$$

$$\leq c\sum_{k=0}^{l/c} \frac{m}{2^k} \log \frac{n}{m} + 2\sum_{k=0}^{l/c} k\frac{m}{2^k}$$

$$= O\left(m \log \frac{n}{m}\right) + O(m)$$

$$= O\left(m \log\left(\frac{n}{m}+1\right)\right). \tag{6}$$

If $|T_d| < |T_p|$, $\frac{|T_d|}{s_k}$ can be less than 1 when k is smaller, thus the sum should be divided into two parts. Also note that we only sum over the nodes with splitting size larger than 0. Even though there could be more than $|T_d|$ nodes in one layer in T_p, only $|T_d|$ of them should count. Thus, we assume $s_k \leq |T_d|$, and we have

$$\sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k}+1\right) = \sum_{k=0}^{l} s_k \log\left(\frac{m}{s_k}+1\right) \tag{7}$$

$$\leq \sum_{k=0}^{2\log_c \frac{n}{m}} |T_d| \log(1+1)$$

$$+ \sum_{k=c\log_2 \frac{n}{m}}^{l} \frac{n}{2^{\lfloor k/c \rfloor}} \log\left(\frac{m}{n/2^{\lfloor k/c \rfloor}}+1\right) \tag{8}$$

$$\leq O\left(m \log \frac{n}{m}\right) + c\sum_{k'=0}^{\frac{l}{c}-\log_2 \frac{m}{n}} \frac{m}{2^{k'}} \log 2^{k'}$$

$$= O\left(m \log \frac{n}{m}\right) + O(m)$$

$$= O\left(m \log\left(\frac{n}{m}+1\right)\right). \tag{9}$$

From (4) to (5) and (7) to (8), we apply Lemmas 1 and 8 and the fact that $f(x) = x\log(\frac{n}{x}+1)$ is monotonically increasing when $x \leq n$.

The above cost does not consider the nodes that are not rank roots. Recall that we use $d(v)$ to denote the rank cluster of a rank root v. Applying Property 7, the total cost (in expectation for

treaps) is less than

$$\sum_{k=0}^{l}\sum_{j=1}^{x_k} d(v_{kj})\log((T_d(v_{kj})+1)$$
$$= d(v_{kj}) \times 2\sum_{k=0}^{l}\sum_{j=1}^{x_k}\log((T_d(v_{kj})+1)$$
$$= O\left(m\log\left(\frac{n}{m}+1\right)\right).$$

Therefore, the split work is $O(m\log(\frac{n}{m}+1))$ in all cases discussed in the theorem. □

Theorem 12. *The total work of union, intersection, or difference of all four balancing schemes on two trees of size m and n ($m \geq n$) is $O(m\log(\frac{n}{m}+1))$.*

This directly follows Theorems 10 and 11.

Theorem 13. *The span of union and intersection or difference on all four balancing schemes is $O(\log n \log m)$. Here, n and m are the sizes of the two trees.*

Proof. For the span of these algorithms, we denote $D(h_1, h_2)$ as the span on *union*, *intersection*, or *difference* on two trees of height h_1 and h_2. According to Theorem 10, the work (span) of *split* and *join* are both $O(\log|T_d|) = O(h(T_d))$. We have

$$D(h(T_p), h(T_d)) \leq D(h(T_p)-1, h(T_d)) + 2h(T_d)$$

Thus, $D(h(T_p), h(T_d)) \leq 2h(T_p)h(T_d) = O(\log n \log m)$. □

Combining Theorems 12 and 13, we come to Theorem 9.

5.4 Other Tree Algorithms Using *join*

Insert and Delete. Instead of the classic implementations of *insert* and *delete*, which are specific to the balancing scheme, we define versions based purely on *join*, and hence independent of the balancing scheme.

We present the pseudocode in Figure 15 to insert an entry e into a tree T. The base case is when t is empty, and the algorithm creates a new node for e. Otherwise, this algorithm compares k with the key at the root and recursively inserts e into the left or right subtree. After that, the two subtrees are *join*ed again using the root node. Because of the correctness of the *join* algorithm, even if there is imbalance, *join* will resolve the issue.

The *delete* algorithm is similar to *insert*, except when the key to be deleted is found at the root, where *delete* uses *join2* to connect the two subtrees instead. Both the *insert* and the *delete* algorithms run in $O(\log n)$ work (and span since sequential).

One might expect that abstracting insertion or deletion using *join* instead of specializing for a particular balance criteria has significant overhead. Our experiments show this is not the case—and even though we maintain the reference counter for persistence, we are only 17% slower sequentially than the highly optimized C++ STL library (see Section 6).

Theorem 14. *For a joinable tree T with weight $n = |T|$, the join-based insertion algorithm does $O(\log n)$ work. The rank of the output tree is at most $c_u + rank(T)$.*

Proof. We prove this by induction. This is obviously true for the base case. Consider $T = node(L, t, R)$, assume e goes to the left subtree L, and the new left subtree is L'. Then the output

Multi-insertion

```
1  ins_sorted(T, A, m) {
2    if (T = ∅) return build(A, m);
3    if (m = 0) return T;
4    ⟨L, e, R⟩ = expose(T);
5    b = binary_search(A, m, k(e));
6    d = (b < m) and (k(A[b]) > k(e));
7    L = ins_sorted(r->lc, A, b) ||  R = ins_sorted(r->rc, A+b-d, m-b-d);
8    return join(L, e, R); }
9  multi_insert(t, A, m) {
10   (A2, m2) = sort_rm_dup(A, m);
11   return ins_sorted(t, A2, m2);}
```

Insertion

```
1  insert(T, e) {
2    if T = ∅ then
3      return singleton(e);
4    ⟨L, e′, R⟩ = expose(T);
5    if k(e) = k(e′) then return T;
6    if k(e) < k(e′) then
7      return join(insert(L, e), e′, R);
8    return join(L, e′, insert(R, e)); }
```

Deletion

```
1  delete(T, k) {
2    if T = ∅ then return ∅;
3    ⟨L, e′, R⟩ = expose(T);
4    if k = k(e′) then
5      return join2(L, R);
6    if k < k(e′) then
7      return join(delete(L, k), e′, R);
8    return join(L, e′, delete(R, k)); }
```

Map and Reduce

```
1  map_reduce(T, g′, f′, I′) {
2    if T = ∅ then return I′;
3    ⟨L, k, v, R⟩ = expose(T);
4    L′ = MapReduce(L, g′, f′, I′) ||
5      R′ = MapReduce(R, g′, f′, I′);
6    return f′(L′, f′(g′(k, v), R′)); }
```

Range

```
1  range(T, l, r) {
2    (T1, T2) = split(T, l);
3    (T3, T4) = split(T2, r);
4    return T3; }
```

Foreach Index

```
1  foreach_index(T, φ, s) {
2    if (t = ∅) return;
3    (L, e, R) = expose(T);
4    left = size(L);
5    L = foreach_index(L, φ, s); ||
6      R = foreach_index(R, φ, s+1+left);
7    φ(e, left);}
```

Build

```
1  build_sorted(S, i, j) {
2    if i = j then return ∅;
3    if i + 1 = j then
4      return singleton(S[i]);
5    m = (i + j)/2;
6    L = build'(S, i, m) ||
7      R = build'(S, m + 1, j);
8    return join(L, S[m], R); }
9  build(S, m) {
10   (S2, m2) = sort_rm_dup(S, m);
11   build_sorted(S2,0,m2);}
```

Filter

```
1  filter(T, f) {
2    if T = ∅ then return ∅;
3    (L, e, R) = expose(T);
4    L′ = filter(L, f) ||
5        R′ = filter(R, f);
6    if f(e) then return join(L′, e, R′);
7    else join2(L′, R′); }
```

Fig. 15. Pseudocode of some *join*-based functions. They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can run in parallel based on fork-join parallelism.

tree is $T' = join(L', t, R)$. Because of the induction hypothesis $rank(L') \leq rank(L) + c_u$. Because of the **submodularity rule**, $rank(T') \leq rank(T) + c_u$. □

Theorem 15. *For a joinable tree T with weight $n = |T|$, the join-based deletion algorithm does $O(\log n)$ work. The rank of the output tree is at most $rank(T)$.*

Proof. We first prove this by induction. This is obviously true for the base case. Consider $T = node(L, t, R)$, assume k falls in the right subtree R, and the new left subtree is R'. Then the output tree is $T' = join(L, t, R')$. From the induction hypothesis, $rank(R') \leq rank(R)$. Based on the decreasing side of the **submodularity rule**, $rank(T') \leq rank(T)$. This proves that the output tree of *delete* is at most the rank of the input.

In *delete*, there is at most one invocation of *join2*, which takes time no more than $\log n$. For $T' = join(L, t, R')$, we next prove the cost. Consider two cases.

(1) The key $k \neq k(R)$. WLOG assume k falls in the right subtree of R (the other case is symmetric). $R' = join\ (lc(R), R, R_r)$ where $R_r = delete\ (rc(R), k)$. Then

$$rank(R') \geq rank(lc(R)) \geq rank(R) - c_u \geq rank(L) - 2c_u + c_l.$$

Meanwhile, $rank(R') \leq rank(R)$ as we have proved above. Therefore,

$$rank(R') \leq rank(R) \leq rank(L) + c_u - c_l.$$

In summary, $rank(L)$ and $rank(R')$ differ by a constant. The cost of a single *join* is a constant.

(2) The key $k = k(R)$. $R' = join2\ (lc(R), rc(R))$. The cost of this *join* is at most $O(\log n)$, but this only happens once.

In summary, the total cost of all *join* is $h(T) + O(\log n) = O(\log n)$, and the cost of the *join2* is at most $O(\log n)$. This proves the above theorem. □

Build. A balanced binary tree can be created from a sorted array of key-value pairs using a balanced divide-and-conquer over the input array and combining with *join*. To construct a balanced binary tree from an arbitrary array, we first sort the array by the keys, then remove the duplicates. All entries with the same key are consecutive after sorting, so the algorithm first applies a parallel sorting and then follows by a parallel packing. The algorithm then extracts the median in the de-duplicated array, and recursively constructs the left/right subtree from the left/right part of the array, respectively. Finally, the algorithm uses *join* to connect the median and the two subtrees. The work is then $O(W_{\text{sort}(n)} + W_{\text{remove}(n)} + n)$ and the span is $O(S_{\text{sort}(n)} + S_{\text{remove}(n)} + \log n)$. For work-efficient sort and remove-duplicates algorithms with $O(\log n)$ span this gives the bounds in Table 2.

Bulk Updates. We use *multi_insert* and *multi_delete* to commit a batch of write operations. The function *multi_insert*(T, A, m) takes as input a P-Tree root t, and the head pointer of an array A with its length m.

We present the pseudocode of *multi_insert* in Figure 15. This algorithm first sorts A by keys, and then removes duplicates in a similar way as in *build*. We then use a divide-and-conquer algorithm *multi_insert_s* to insert the sorted array into the tree. The base case is when either the array A or T is empty. Otherwise, the algorithm uses a binary search to locate t's key in the array, getting the corresponding index b in A. d is a bit denoting if k appears in A. Then, the algorithm recursively multi-inserts A's left part (up to $A[b]$) into the left subtree, and A's right part into the right subtree. The two recursive calls can run in parallel. The algorithm finally concatenates the two results by the root of T. A similar divide-and-conquer algorithm can be used for *multi_delete*, using *join2* instead of *join* when necessary.

Decoupling sorting from inserting has several benefits. First, parallel sorting is well studied and there exist highly optimized sorting algorithms that can be used. This simplifies the problem. Second, after sorting, all entries in A to be merged with a certain subtree in T become consecutive. This enables the divide-and-conquer approach which provides good parallelism, and also gives better locality.

The total work and span of inserting or deletion of a sorted array of length m into a tree of size $n \geq m$ is $O(m \log(\frac{n}{m} + 1))$ and $O(\log m \log n)$, respectively [14]. The analysis is similar to the *union* algorithm.

Range. *range*(T, k_L, k_R) extracts a subset of tuples from T in a key range $[k_L, k_R]$, and outputs them in a new P-Tree. The cost of the *range* function is $O(\log n)$. The pure *range* algorithm copies nodes on two paths, one to each end of the range, and uses them as pivots to *join* the subtrees back. When the extracted range is large, this pure *range* algorithm is much more efficient (logarithmic time) than visiting the whole range and copying it.

Filter. The *filter*(T, ϕ) function returns a tree with all tuples in T satisfying a predicate ϕ. This algorithm filters the two subtrees recursively, in parallel, and then determines if the root satisfies ϕ. If so, the algorithm uses the root as the pivot to *join* the two recursive results. Otherwise, it calls *join2*. The work of *filter* is $O(n)$ and the depth is $O(\log^2 n)$ where n is the tree size.

Map and Reduce. The function *map_reduce*$(T, f_m, \langle f_r, I\rangle)$ on a tree t (with data type E for the tuples) takes three arguments and returns a value of type V'. $f_m : E \mapsto V'$ is the a map function that converts each stored tuple to a value of type V'. $\langle f_r, I\rangle$ is a monoid where $f_r : V' \times V' \mapsto V'$ is an associative reduce function on V', and $I \in V'$ is the identity of f_r. The algorithm will recursively call the function on its two subtrees in parallel, and reduce the results by f_r afterwards.

6 EXPERIMENTS

To evaluate the performance of our algorithms, we performed several experiments across the four balancing schemes using different set functions, while varying the core count and tree sizes. We also compare the performance of our implementation to other existing libraries and algorithms.

In this article, we mainly show the performance for the set operations. For other *join*-based algorithms, some performance evaluations are available in [11, 60–62].

Implementation Details. We implemented all the algorithms in the **PAM** (**Parallel Augmented Map**) library [59, 62]. PAM is a C++ library supporting normal ordered maps (key-value store) and augmented maps (as defined in [62]) based on balanced binary trees. All four discussed balancing schemes are supported in PAM. The algorithms in PAM are based on the *join*-based algorithms, which matches the versions described in this article. Our code is available online [59]. We refer to the trees implemented in PAM as **P-Trees** [61].

As a general-purpose library, PAM also supports persistence (and thus multi-versioning) based on path-copying, and reference count garbage collection. A persistent (or purely functional) update will not modify the input tree, but generate a new version with the update as the output. To perform an update operation using path-copying, the algorithm will copy all tree nodes on the affected path, and the new root pointer represents the output tree. We refer the readers to [61, 62] for more details. This means that a tree node can be shared among multiple versions, and thus we use a reference counter for each node to aid memory management. PAM also allows for in-place updates. In our experiments, since the two implementations that we are comparing with are not persistent, we test the in-place version of the algorithms in PAM.

Each tree node in P-tree contains the key, the value, two child pointers ($2 \times 8 = 16$ bytes), the balance information (4 bytes if needed; see details below), the size of the subtree (4 bytes), and a

reference count (4 bytes, used for garbage collection; see details above). The balancing information is only needed for AVL trees (the height) and RB trees (the black height and the color). For both cases, we use a 4-byte integer to store them (for RB tree we encoded the color as a bit in the black height). For treaps, the priority of each node is generated by a user-specified hash function on the key. For WB trees, the weight can be directly computed by one plus the size. This means that for a key-value pair with 8-byte key and 8-byte value (which is the setting we used in our experiments), each node is 40 bytes (for WB trees and treaps) or 44 bytes (for AVL and RB trees). As discussed below, experiments show that WB trees generally show the best performance among all the balancing schemes we tested. We believe the reason is exactly because WB tree does not store extra balancing information, and generally has lower height than treaps. Therefore, the default interface of PAM is using WB trees.

We note that some metadata stored in the tree nodes, such as the size and the reference counter, are not necessary for implementing set operations. However, they are useful in supporting more complete functionalities in the ordered map interface. For example, the size of each subtree is useful in answering queries such as selecting the k-th element, or answering the rank of an entry in the ordered map. The reference counter is useful in memory management for persistent updates. This information is not maintained in the other implementations that we compare to. In Section 7, we will present some discussions about more space-efficient search tree structures.

Experiment Setups and Baseline Algorithms. For all the experiments, we use a 72-core Dell R930 with 4× Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4 GHz, and 45 MB L3 cache), and 1 Tbyte memory. Each core is two-way hyperthreaded giving 144 hyperthreads. Our code was compiled using g++ 5.4.1. In the semantics, we only use the binary fork-join, and parallel for loop. We compile with `-O2` because this gives us more stable results. We use `numactl -i all` in all experiments with more than one thread. It evenly spreads the memory pages across the processors in a round-robin fashion.

In all our experiments,, we use keys and values of the 64-bit integer data type. Throughout this section, n_1 and n_2 represent the sizes of the two input sets. We generate multiple sets varying in size up to 10^8. For most of the experiments (except for Figure 17), the keys are 64-bit integers drawn from a uniform distribution. As mentioned, some experiments using different input distributions, and real-world data distributions (e.g., Yahoo! Cloud Serving Benchmark (YCSB) [28] using Zipfian distribution), have been shown in some other papers [14, 61]. We note that uniform distribution also means that the two input trees are fully interleaving with each other, which is the worst case for our algorithms. Because our algorithms are comparison-based, the actual key distribution does not affect the running time, but how the two sets interleaving with each other matters. To better understand this, in Figure 17 we will also show when the two trees have different overlapping ratio in key ranges. Our experiments show that when the two sets partially overlap with each other, our algorithms can exhibit better performance.

We test our algorithm by comparing it to other available implementations. This includes the sequential version of the set functions defined in the C++ **Standard Template Library** (**STL**) [50], which we use for `std::vector`, and STL's `std::set` (implemented by RB tree). To see how well our algorithm performs in a parallel setting, we compare it to parallel WBB-trees [31] and the MCSTL library [33], both supporting array-tree *union*. For array-tree *union*, we use the tree to denote the first set (size n_1) and the array as the second set (size n_2). For MCSTL and WBB-trees, the entry stored in each tree node is just a key. For STL-map, STL-set, and our implementation based on PAM, the entries are key-value pairs.

We note that more experiments on the *join*-based algorithms are also available in some other papers, e.g., on different input distributions [14], comparing with concurrent data structures

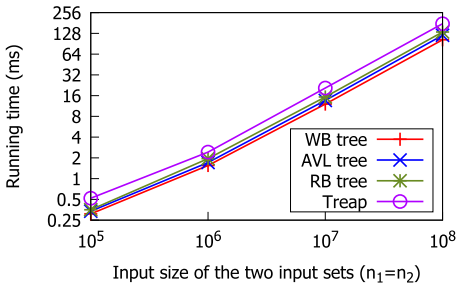
[11, 61, 62], and comparing with existing geometric libraries [60]. The PAM library has also been tested on real-world database benchmark and micro-benchmarks, including YCSB [28] and TPC benchmarks [1] (see more details in [61]).

Comparing Different Balancing Schemes and Set Operations. To compare the four balancing schemes, we choose *union* as the representative operation. Other operations give similar results. We compare the schemes across varying input sizes. The two input sets are of the same size varying from 10^5 to 10^8. Figure 16(a) shows the runtime of *union* for varying tree sizes and all four balancing schemes on 72 cores (144 threads). The overall trend and running times are similar across the balancing schemes. Generally speaking, WB trees have the best performance among the four tested balancing schemes. This is because the other three balancing schemes also need to store an extra field as the balancing information (e.g., the height for AVL trees). For WB trees, the balancing criteria is the size of each subtree, which is maintained anyway for answering other queries (e.g., quickly selecting the i-th element in the set). It is perhaps not surprising that all balancing schemes achieve similar performance because the dominant cost is in cache misses along the paths in the tree, and all schemes keep the trees reasonably balanced. Treaps have the worst performance among the four, likely due to the larger height of the tree. The difference in running times between treaps (slowest) and WB trees (fastest) can be up to 70%. However, for AVL and RB trees, the difference from WB trees is usually within 30%. For this reason, the PAM library uses the WB trees as the default underlying balancing scheme.

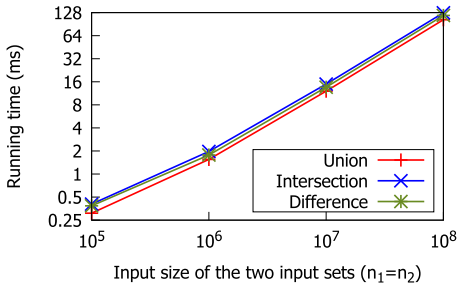
We will use the WB tree as the representative tree in the following experiments. We first show tests to compare different set operations. The two input sets are of the same size varying from 10^5 to 10^8. To make *intersection* and *difference* have reasonable outputs (instead of outputting an empty set for most of the time), we select at least a half of the elements in the second set from the first set. Figure 16(b) compares time for the *union*, *intersection*, and *difference* functions. The three functions have very similar performance. For *intersection* and *difference*, the time also includes possible cost in collecting nodes that are not in the output. For this reason, *intersection* and *difference* are slightly more expensive than *union*. We note that by using a persistent version of these algorithms, one can avoid the extra cost of garbage collection. This is out of the scope of this article, so we omit the details.

Comparing to Existing Parallel Implementations. To see how well our algorithm performs in the parallel setting, we compare it to parallel WBB-trees [31] and the MCSTL library [33]. WBB-trees, as well as the MCSTL, offer an interface for bulk insertions and deletions, which takes a tree and a sorted array, and inserts (or deletes) the elements in the array into (from) the tree. Our version is symmetric in taking both input sets as trees. If the size of the array is smaller than the tree, array-tree *union*s have an inherent advantage over tree-tree unions since accessing an array is much more cache efficient than accessing an tree. As shown in Figure 16(c), we set the first set size (n_1) as 10^8 and vary the other set size (n_2) from 10^5 to 10^8. In this case, the smaller set will be represented as the input array in WBB-trees and MCSTL. In this case, WBB-trees have better performance than our implementation when n_2 is small. This is probably because WBB-tree itself has a more cache-aware layout (eight keys per cache line as opposed to one), leading to a better cache utilization compared to both the MCSTL and our implementation. When n_2 gets larger, our implementation achieves similar performance to WBB-tree, and is much faster than MCSTL.

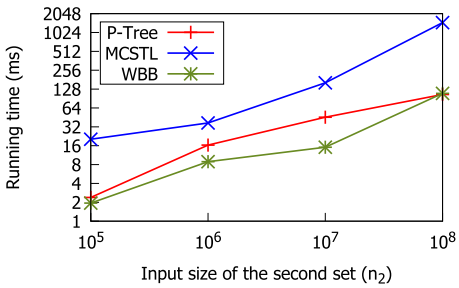
In Figure 16(d), we fix the second set size n_2 as 10^8, which is the array in the input of WBB-tree and MCSTL. We vary the first set size (n_1) from 10^5 to 10^8. In this case, the running time of both WBB-tree and MCSTL becomes (almost) flat. In other words, lowering the second set size does not reduce the running time for either WBB-tree or MCSTL, and the running time is proportional



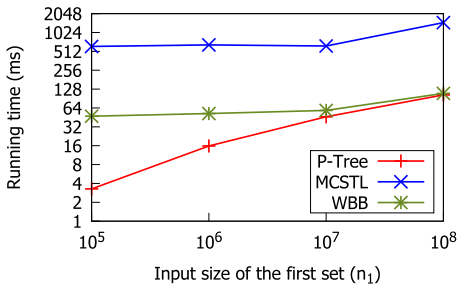
(a) Union, 144 threads
Various balancing schemes
$n_1 = n_2$ shown on x-axis



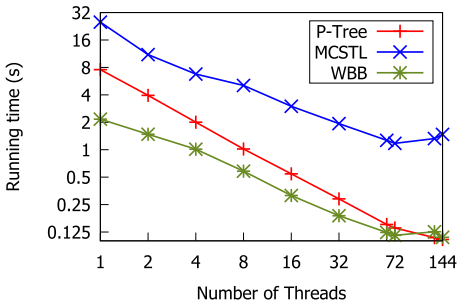
(b) WB Trees, 144 threads
Various set operations
$n_1 = n_2$ shown on x-axis



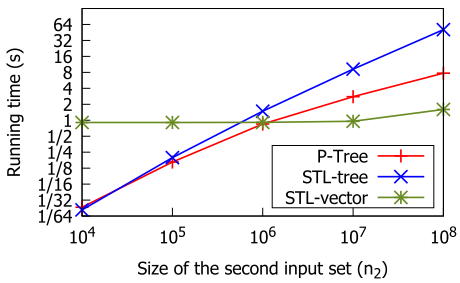
(c) WB Trees, union, 144 threads
Comparing to MCSTL and WBB-trees
$n_1 = 10^8$, varying n_2 as shown on x-axis



(d) WB Trees, union, 144 threads
Comparing to MCSTL and WBB-trees
$n_2 = 10^8$, varying n_1 as shown on x-axis



(e) WB Trees, union, $n_1 = n_2 = 10^8$
Comparing to MCSTL and WBB-trees
Varying number of threads as shown on x-axis



(f) WB Trees, union, 1 thread (sequential)
Comparing to STL-map and STL-set
$n_1 = 10^8$, varying n_2 as shown on x-axis

Fig. 16. Experimental results on *join*-based algorithms.

to n_2. In our symmetric version, the order of the two input sizes does not make a difference. The curves of P-Trees in Figure 16(c) and (d) show a similar trend.

Scalability Tests. We also compare the parallelism of these implementations. In Figure 16(e), we show their performance across 72 cores (144 threads). The inputs are both of size 10^8, and generated from a uniform distribution of integers. The trend shows that all three tested implementations

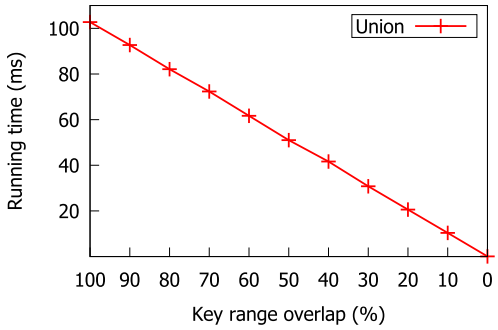
have good speedup up to 72 threads. Our version gets some extra speedup when hyperthreading is involved. With small number of threads, WBB-trees are slightly faster than our code, but when it comes to all 144 threads, P-Trees are slightly faster than WBB-trees. This indicates that our algorithm achieves better parallelism. On 144 threads, P-Trees get a 73-fold self-speedup, while MCSTL and WBB-trees have self-speedup numbers as 17 and 20, respectively. Because the sequential version of WBB-trees is about 3.5× faster than our implementation, the parallel performances using all 144 threads of P-Trees and WBB-trees are close to each other.

To conclude, in terms of parallel performance, our code and WBB-trees are always much better than MCSTL because of MCSTL's slow sequential performance (indicating that it may have more expensive work). WBB-trees achieve a slightly better performance than ours when fewer threads are used, but when using all available threads, WBB-trees achieve similar performance to ours, indicating that our implementation explores better parallelism. Also, when the second input set size is small, neither MCSTL nor WBB-trees can take the advantage, while the running time of our implementation is always proportional to the *small* set size.

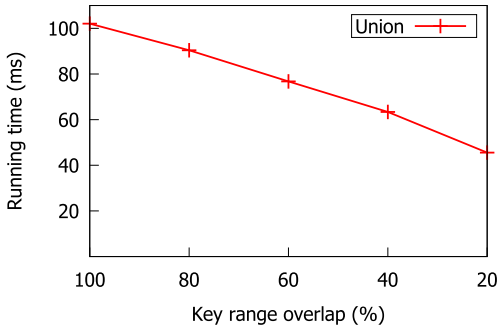
Comparing to Sequential Implementations. The STL supports `set_union`, `set_intersection`, and `set_difference` on any container class, including sets based on RB trees, and sorted vectors (arrays). Since the STL does not offer any parallel version of these functions, we could only use it for sequential experiments. For two inputs of size n_1 and n_2, `set_union` takes $O(n_1 + n_2)$ time on `std::vectors` by using the standard merging algorithm which uses two pointers moving from left to right on the two inputs, comparing the current values, and writing the lesser to the end of the output. This is also the merge algorithm in a standard merge sort. For `std::set` that uses RB trees, we can insert elements from the smaller set into the larger, leading to a time of $O(\min(n_1, n_2) \log(n_1 + n_2))$. These two sequential algorithms are noted as STL-tree and STL-vector in our tests.

Figure 16(f) gives a comparison of times for *union*. For equal lengths, our implementation is about a factor of 7× faster than STL-tree, and about 5× slower than STL-vector. This is not surprising since we are asymptotically faster than STL-tree. Meanwhile, for STL-vector, the theoretical cost ($O(n_1 + n_2)$) is asymptotically the same as ours ($O(n_2 \log(n_1/n_2 + 1))$) when $n_1 = n_2$. However, STL-vector's array-based implementation just reads and writes the values, one by one, from flat arrays, and therefore has much less overhead and much fewer cache misses than ours. We note that using trees as the representation of ordered sets has the benefit of allowing for fast insertions and deletions, which is not supported by array-based implementations. For taking the union of smaller and larger inputs, our *union* is orders of magnitude faster than either STL version. This is because their theoretical work bound ($O(n_1 + n_2)$ and $O(n_2 \log(n_1 + n_2))$ is worse than our $O(n_2 \log(n_1/n_2 + 1))$), which is optimal in the comparison model.

Testing Different Key Range Overlapping. In all previous experiments, we use two sets with fully interleaved key ranges. As mentioned, this is actually the worst case for our algorithms. We also tested two sets of different key range overlapping, and show the running time in Figure 17(a) and (b). In Figure 17(a), both sets have a key range of 10^9, but they have partially overlapped key range. The first set will always have keys in range $[0, 10^9]$, and the second set will have keys in key range $[i \times 10^8, 10^9 + i \times 10^8]$ for test i. We vary i from 0 to 10, which means the key range overlapping of the two sets varies from 0% (fully non-overlapping) to 100% (fully interleaving). As shown in Figure 17, as the key range overlapping ratio decreases, the total running time also decreases linearly. This is not surprising because when the overlapping range is small, it means that in the divide-and-conquer algorithm, the two subproblems usually have unbalanced size, and some of the subproblems quickly reach the base case. When the two key ranges do not overlap with



(a) Key range partially overlapping



(b) The key range of T_2 is contained in T_1

Fig. 17. The running time of *union* algorithm on two input trees with different key range overlapping. We use WB trees and 144 threads. The numbers on the x-axis show how much the key ranges of the input sets overlap.

each other, the *union* algorithm degenerates to a *join2* algorithm, which has work $O(\log^2 n)$. This matches our experimental result, where when the overlapping is 0%, the running time is smaller than 1 ms.

We also tested when the key range of a set is a subset of the other key range. In particular, in Figure 17(b), we always set the key range of the first set to be $[0, 10^9]$, and the other key range is set to be $[0, 2i \times 10^8]$ for $i = 1$ to 5. This means the key range of T_2 is always contained in T_1's key range, and the overlapping changes from 20% to 100%. Similar to Figure 17(a), we see that when the overlapping ratio decreases, the performance also improves in an almost linear manner. This is also because in many subproblems, one of the tree sizes hits 0 quickly. As a result, the total work performed by the algorithm is much smaller than normal (i.e., when they fully interleave with each other).

7 RELATED WORK

Join-Based Algorithms. Tarjan first studied the *join* and *split* functions in [64]. Tarjan showed how to efficiently implement the functions on RB and splay trees but did not give any applications for the two functions. Adams [2, 3] applied *join* and *split* on WB trees to some bulk functions and showed an elegant way to implement *union*, *intersection*, and *difference*. The method was then implemented in some languages and libraries such as MIT/GNU Scheme in Haskell [48]. Adams' paper was reported buggy [36, 58] in parameter choosing and also in that the *join* (which is called *concat3* in his paper) function does not rebalance the tree, but the framework of set functions implementation is still used and studied today. Adams did not consider parallelism in the algorithm, but the divide-and-conquer scheme is inherently parallel.

join and *split* appear in the LEDA library [49] for sorted sequences, and the CGAL library for ordered maps [67]. None of this work considered parallel algorithms based on the functions, or how to build an interface out of just *join*. Frias and Singler [33] use *join* and *split* on RB trees for an implementation of the MCSTL, a multi-core version of the C++ STL. Their algorithms are lower level based on partitioning across processors, and are for bulk insertion and deletion. The functions *join*, *split* and *join2* are also studied and used in various previous work for trees to support multiple applications [12, 27, 56].

Set Operations. Merging two ordered sets has been well studied in the sequential setting. Hwang and Lin [37] describe an algorithm to merge two arrays, which costs optimal work. Their algorithm

works for arrays, and since writing back both array costs $O(m + n)$ work, the algorithm only returns the cross pointers between two arrays. Brown and Tarjan [22] considered input data to be arranged in a BST, which allows the merged result to be explicitly given by a new BST in time $O(m \log(\frac{n}{m} + 1))$. Their algorithm works on AVL and 2-3 trees. None of these algorithms considered parallelism. Katajainen et al. [42] studied the space efficiency on merging two sets in parallel. Their focus is not on reducing time complexity. Furthermore, the above-mentioned works are not based on join and are much more complicated than our algorithm.

There is also previous work studying parallel set operations on two ordered sets, but each previous algorithm only works on one type of balance tree. Paul et al. studied bulk insertion and deletion on 2-3 trees in the PRAM model [55]. Park and Park showed similar results for RB trees [54]. These algorithms are not based on *join* and are not work-efficient, requiring $O(m \log n)$ work. Katajainen [41] claimed an algorithm with $O(m \log(\frac{n}{m} + 1))$ work and $O(\log n)$ span using 2-3 trees, but it appears to contain some bugs in the analysis [16]. Blelloch and Reid-Miller described a similar algorithm as Adams' (as well as ours) on treaps with optimal work (in expectation) and $O(\log n)$ span (with high probability) on a EREW PRAM with scan operations. This implies $O(\log n \log m)$ span on a plain EREW PRAM, and $O(\log n \log^* m)$ span on a plain CRCW PRAM. The pipelining that is used is quite complicated. Akhremtsev and Sanders [7] describe an algorithm for array-tree *union* based on (a, b)-trees with optimal work and $O(\log n)$ span on a CRCW PRAM. Our focus in this article is in showing that very simple algorithms are work efficient, have polylogarithmic span, and generic for multiple balancing schemes, and less with optimizing the span. We note that their algorithms use multi-way search tree instead of binary, which potentially would have better I/O efficiency than the binary trees. We present some discussions about this at the end of this section. There has also been work about batched updates on certain balancing schemes [5, 6].

Set Algorithms in the Concurrent and Distributed Setting. Many researchers have considered concurrent implementations of balanced search trees (e.g., [21, 43, 44, 51]). None of these are work efficient for *union* since it is necessary to insert one tree into the other requiring at least $\Omega(m \log n)$ work. Researchers have also studied distributed memory implementations of maps and sets, including a distributed version of STL as part of the HPC++ effort [39], and the STAPL library [63]. The emphasis of this work is on how the maps and sets are partitioned across the memories.

Discussion about Other Balancing Schemes. This article shows that the *join*-based algorithm framework applies to four balancing schemes. The authors believe that similar ideas apply to other balancing schemes. For example, a recent paper [35] proposed the WAVL tree, which uses a similar notion of *rank* as our work, but only allows integer ranks. Their methodology applies to balanced binary trees where the height of the two siblings can differ by at most 2. They also showed the equivalence of WAVL tree to an RB tree. As a result, our *join*-based framework also applies to WAVL trees. This is another recent paper on zip trees [65], which is a randomized balanced tree structure. The *join* algorithms for treaps and all *join*-based algorithms work correctly on zip trees. However, zip trees does not directly fit in the definition of weakly joinable trees, and thus the bounds and analysis does not directly apply to zip trees.

Discussion about Concurrent Tree Structures. There is previous work focusing on supporting concurrent operations on BSTs [23, 26, 45–47, 53, 66]. Some of this previous work aims at making such operations lock-free or wait-free [9, 21, 24, 32, 68]. The supported operations usually include insertion, deletion, update, lookup, range queries, and sometimes more complicated queries. Instead of focusing on supporting concurrent operations, P-Trees provide bulk functions (e.g., *union* and *multi_insert*) to commit a batch of operations in parallel. P-Trees and the PAM library are also safe for concurrency, but concurrency is supported by making P-Trees persistent by path-copying

[61, 62]. This requires all concurrent threads to work on a snapshot of the tree. To allow for serializable concurrent operations on P-Trees, one can batch all update operations and commit them using the parallel bulk functions. Since this is not the main focus of this article, we omit the details, and more information can be found in [11, 61].

Discussion about I/O- and Space-Efficient Search Tree Structures. One concern of using binary trees is the space overhead and I/O efficiency. In particular, each data entry is stored in a tree node, which needs to maintain the pointers, size, reference count, and so on. For this reason, there are also researchers attempting to design algorithms for B-tree or B-tree-like data structures.

Akhremtsev and Sanders [7] described an algorithm for array-tree set algorithms based on (a, b)-trees. Their algorithms are also based on the (multi-)split-join framework, and are work efficient and highly parallelized. Their data structure uses an array of pointers to the nodes on the left or right spine of the input tree with each rank. This provides fast access to spine nodes of a given rank, and enables theoretical efficiency. In their paper [7], their implementation was compared to a previous version of the PAM library, and achieved better performance. Their experiments combine a bulk (an array) to a tree structure, while our *union* algorithm combines two trees. As mentioned above, our implementation maintains extra metadata such as reference count and size, and uses binary tree. The advantage of their algorithm is likely due to the better space efficiency (less metadata in each tree node) and I/O efficiency by using (a, b)-trees (multiple keys maintained in one tree node). We choose to store extra information and use binary structure because we want to support more functionalities. Because of the extra space used, our implementation supports persistence (multi-versioning) with garbage collection, and more operations such as selecting the k-th element (see more details in Section 6). Also, using the binary structure enables cheaper path-copying (for persistence), which would be expensive for a regular B-tree or (a, b)-tree (see details below). This is important for some applications such as databases with multi-versioning [61]. Although these functionalities are not specifically necessary for implementing set operations, we do so in order to support the general-purpose ordered map interface.

P-trees have also been compared with concurrent B-tree-like structures, and have been shown to achieve competitive or better performance by batching the updates and using a divide-and-conquer algorithm similar to *union* [61].

It is also possible to extend our *join*-based algorithms to B-tree-like data structures. One concern of using B-trees is that the large internal nodes in B-trees are *expensive to copy*, and therefore not well suited for supporting persistence. In 2019, Dhulipala et al. [29] extend the *join*-based algorithms to *C-trees* based on treaps. Instead of supporting a full ordered map interface, C-trees only focused on graph streaming. While C-trees are binary trees, they store multiple keys (in their case, graph edges) in the same tree node to allow for better cache locality and compression. Each node is represented by a randomly sampled head key k, and a subsequent *chunk*, which contains all elements between k and the next head. Each chunk can further be compressed. A C-tree is then represented as a *prefix*, which is the chunk before the first head, and a tree, where each node contains a head and its subsequent chunk. When a persistent update occurs, only the *head* of the relevant internal nodes need to be copied. C-trees are shown to be practical for graph streaming. The C-tree provides useful insights for extending our work to B-tree-like data structures. To extend our general ordered map library to B-tree or its relatives, and to enable both good theoretical bounds and practical performance is an interesting future direction.

8 CONCLUSION

In this article, we study parallel balanced binary trees. We proposed the *join*-based algorithms, along with its analysis framework. We show conditions to make trees *joinable*, such that the

join-based (parallel) algorithms work correctly and efficiently. In particular, we show for the first time that a very simple "classroom-ready" set of algorithms is indeed work optimal when used with four different balancing schemes—AVL, RB, WB trees and treaps—and also highly parallel. The only tree-specific algorithm that is necessary is the *join* algorithm. For analysis, our approach defines the notion of rank (differently for different balancing schemes) and shows invariants on the rank. The definition of rank, along with the *join* algorithm for a specific balancing scheme, ensures that the *join*-based algorithms are work optimal with polylogarithmic span.

We also test the performance of our algorithm. Our experiments show that our sequential algorithm is about 7× faster for *union* on two maps of size 10^8 compared to the STL RB tree implementation. In parallel, our code outperforms or is competitive to two baseline algorithms (MCSTL and WBB-tree) on different input sizes. Our code also achieves 73× speedup on 72 cores with hyperthreading.

ACKNOWLEDGMENTS

We thank the reviewers for their comments and suggestions.

REFERENCES

[1] [n. d.]. TPC Benchmarks. ([n. d.]). Retrieved April 2019 from http://tpc.org/.
[2] Stephen Adams. 1992. *Implementing Sets Effciently in a Functional Language*. Technical Report CSTR 92-10. University of Southampton.
[3] Stephen Adams. 1993. Efficient sets—a balancing act. *Journal of Functional Programming* 3, 04 (1993).
[4] Georgy Adelson-Velsky and E. M. Landis. 1962. An algorithm for the organization of information. *USSR Academy of Sciences* 145 (1962), 263–266. In Russian, English translation by Myron J. Ricci, Soviet Doklady 3 (1962), 1259–1263.
[5] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. 84–95.
[6] Kunal Agrawal, Seth Gilbert, and Wei Quan Lim. 2018. Parallel working-set search structures, in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA'18)*.
[7] Yaroslav Akhremtsev and Peter Sanders. 2016. Fast parallel operations on search trees. In *23rd IEEE International Conference on High Performance Computing*. 291–300.
[8] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM Symposium on Principles and Practice of Parallel Programming*. 14–27. https://doi.org/10.1145/3178487.3178489
[9] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A key-value map for scalable real-time analytics. In *Proceedings of the 22nd ACM Symposium on Principles and Practice of Parallel Programming*. 357–369. https://doi.org/10.1145/3018743.3018761
[10] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (1972), 290–306.
[11] Naama Ben-David, Guy Blelloch, Yihan Sun, and Yuanhao Wei. 2019. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'19)*.
[12] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. 1985. Biased search trees. *SIAM Journal on Computing* 14, 3 (1985), 545–568.
[13] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Parallel ordered sets using join. arXiv preprint:1602.02120.
[14] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 253–264.
[15] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2019. Optimal parallel algorithms in the binary-forking model. *CoRR* abs/1903.04650 (2019). arXiv:1903.04650, http://arxiv.org/abs/1903.04650.
[16] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast set operations using treaps. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*. 16–26.
[17] Norbert Blum and Kurt Mehlhorn. 1980. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science* 11, 3 (1980), 303–320.
[18] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing* 27, 1 (1998), 202–229.

[19] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46, 5 (1999), 720–748.
[20] Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM* 21, 2 (April 1974), 201–206.
[21] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. 257–268.
[22] Mark R. Brown and Robert E. Tarjan. 1979. A fast merging algorithm. *Journal of the ACM (JACM)* 26, 2 (1979), 211–226.
[23] Trevor Brown. 2016. Lock-free Chromatic Trees in C++. Retrieved April 2019 from https://bitbucket.org/trbot86/implementations/src/.
[24] Trevor Brown and Hillel Avni. 2012. Range queries in non-blocking k-ary search trees. In *Proceedings of the 16th International Conference on Principles of Distributed Systems*, Lecture Notes in Computer Science, Vol. 7702. 31–45.
[25] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. 329–342.
[26] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*.
[27] Marek Chrobak, Tomasz Szymacha, and Adam Krawczyk. 1990. A data structure useful for finding Hamiltonian cycles. *Theoretical Computer Science* 71, 3 (1990), 419–424.
[28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.
[29] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.
[30] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing*. See also Technical Report CSE-2010-04, EECS Department, York University, 2010.
[31] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. 2014. Parallel bi-objective shortest paths using weight-balanced B-trees with bulk updates. In *Experimental Algorithms*. Springer, 111–122.
[32] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 275–286. https://doi.org/10.1145/3323165.3323197
[33] Leonor Frias and Johannes Singler. 2007. Parallelization of bulk operations for STL dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007*. 49–58.
[34] R. L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* 17, 2 (1969), 416–429.
[35] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2009. Rank-balanced trees. In *Workshop on Algorithms and Data Structures*. Springer, 351–362.
[36] Yoichi Hirai and Kazuhiko Yamamoto. 2011. Balancing weight-balanced trees. *Journal of Functional Programming* 21, 03 (2011), 287–307.
[37] Frank K. Hwang and Shen Lin. 1972. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing* 1, 1 (1972), 31–39.
[38] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Vol. 17. Addison-Wesley, Reading, MA.
[39] Elizabeth Johnson and Dennis Gannon. 1997. HPC++: Experiments with the parallel standard template library. In *International Conference on Supercomputing (ICS'97)*. 124–131.
[40] Haim Kaplan and Robert Endre Tarjan. 1996. Purely functional representations of catenable sorted lists. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC'96)*. 202–211.
[41] J. Katajainen. 1994. Efficient parallel algorithms for manipulating sorted sets. In *Proceedings of Computer Science Conference*. University of Canterbury.
[42] Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. 1992. *Space-Efficient Parallel Merging*. Springer.
[43] H. T. Kung and Philip L. Lehman. 1980. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems* 5, 3 (1980), 354–382.
[44] Kim S. Larsen. 2000. AVL trees with relaxed balance. *Journal of Computer Systems and Science* 61, 3 (2000), 508–522.
[45] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*. 3:1–3:8.
[46] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'13)*. 302–313.

[47] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems*. 183–196.
[48] Simon Marlow et al. 2010. Haskell 2010 language report. Available online http://www.haskell.org/(May2011).
[49] Kurt Mehlhorn and Stefan Näher. 1999. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York, NY.
[50] David R. Musser, Gillmer J. Derge, and Atul Saini. 2009. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional.
[51] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. 317–328.
[52] Jürg Nievergelt and Edward M. Reingold. 1973. Binary search trees of bounded balance. *SIAM Journal on Computing* 2, 1 (1973), 33–43.
[53] Otto Nurmi and Eljas Soisalon-Soininen. 1996. Chromatic binary search trees. *Acta Informatica* 33, 6 (1996), 547–557.
[54] Heejin Park and Kunsoo Park. 2001. Parallel algorithms for red-black trees. *Theoretical Computer Science* 262, 1–2 (2001), 415–435.
[55] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. 1983. Parallel dictionaries in 2-3 trees. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'83)*. 597–609.
[56] Raimund Seidel and Celcilia R. Aragon. 1996. Randomized search trees. *Algorithmica* 16 (1996), 464–497.
[57] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM* 28, 2 (1985). https://doi.org/10.1145/2786.2793
[58] Milan Straka. 2012. Adams' trees revisited. In *Trends in Functional Programming*. Springer, 130–145.
[59] Yihan Sun, Guy Blelloch, and Daniel Ferizovic. 2018. The PAM Library. https://github.com/cmuparlay/PAM.
[60] Yihan Sun and Guy E. Blelloch. 2019. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX'19)*. 159–173.
[61] Yihan Sun, Guy E. Blelloch, Andrew Pavlo, and Wan Shen Lim. 2020. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB* (2020).
[62] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*.
[63] Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. 2007. Associative parallel containers in STAPL. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*. 156–171.
[64] Robert Endre Tarjan. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA.
[65] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. 2019. Zip trees. In *Workshop on Algorithms and Data Structures*. Springer, 566–577.
[66] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 473–488.
[67] Ron Wein. 2005. *Efficient Implementation of Red-black Trees with Split and Catenate Operations*. Technical Report. Tel-Aviv University.
[68] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-free contention adapting search trees. In *Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures*. 121–132. https://doi.org/10.1145/3210377.3210413

Received September 2020; revised October 2021; accepted December 2021