Just Join for Parallel Ordered Sets

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Daniel Ferizovic
Karlsruhe Institute of Technology
dani93.f@gmail.com

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

ABSTRACT

Ordered sets (and maps when data is associated with each key) are one of the most important and useful data types. The set-set functions union, intersection and difference are particularly useful in certain applications. Brown and Tarjan first described an algorithm for these functions, based on 2-3 trees, that meet the optimal $\Theta\big(m \log\big(\frac{n}{m}+1\big)\big)$ time bounds in the comparison model (n and $m \leq n$ are the input sizes). Later Adams showed very elegant algorithms for the functions, and others, based on weight-balanced trees. They only require a single function that is specific to the balancing scheme—a function that joins two balanced trees—and hence can be applied to other balancing schemes. Furthermore the algorithms are naturally parallel. However, in the twenty-four years since, no one has shown that the algorithms, sequential or parallel are asymptotically work optimal.

In this paper we show that Adams' algorithms are both work efficient and highly parallel (polylog span) across four different balancing schemes—AVL trees, red-black trees, weight balanced trees and treaps. To do this we use careful, but simple, algorithms for JOIN that maintain certain invariants, and our proof is (mostly) generic across the schemes.

To understand how the algorithms perform in practice we have also implemented them (all code except JOIN is generic across the balancing schemes). Interestingly the implementations on all four balancing schemes and three set functions perform similarly in time and speedup (more than 45x on 64 cores). We also compare the performance of our implementation to other existing libraries and algorithms.

1. INTRODUCTION

Ordered sets and ordered maps (sets with data associated with each key) are two of the most important data types used in modern programming. Most programming languages either have them built in as basic types (e.g. python) or supply them as standard libraries (C++, C# Java, Scala, Haskell, ML). These implementations are based on some form of balanced tree (or tree-like) data structure and, at minimum, support lookup, insertion, and deletion in logarithmic time. Most also support set-set functions such as union, intersection, and difference. These functions are particularly useful when using parallel machines since they can support parallel bulk updates. In this paper we are interested in simple and efficient parallel algorithms for such set-set functions.

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[1], is $\log_2 \binom{m+n}{n} = \Theta\big(m \log\big(\frac{n}{m}+1\big)\big)$. Brown and Tarjan first matched these bounds, asymptotically, using a sequential algorithm based on red-black trees [12]. Although designed for merging, the algorithm can be adapted for union, intersection and difference with the same bounds. However, the Brown and Tarjan algorithm is complicated, and completely sequential.

Adams later described very elegant algorithms for union, intersection, and difference, as well as other functions based on a single function, JOIN [1, 2] (see Figure 1). JOIN(L, k, R) takes a key k and two ordered sets L and R such that $L < k < R$ and returns the union of the keys [30, 28]. It can be used to implement JOIN2(L, R), which does not take the key in the middle, and SPLIT(T, k), which splits a tree at a key k returning the two pieces and a flag indicating if k is in T (See Section 4). With these three functions, union, intersection, and difference (as well as insertion, deletion and other functions) are almost trivial. Because of this, at least three libraries use Adams' algorithms for their implementation of ordered sets and tables (Haskell [21] and MIT/GNU Scheme, and SML).

Adam's original algorithms implemented JOIN using weight-balanced trees[2]. JOIN can also be implemented using other balance criteria. Sleator and Tarjan describe an algorithm for JOIN based on splay trees which runs in amortized logarithmic time [28]. Tarjan describes a version for red-black tree that runs in worst case logarithmic time [30].

Surprisingly, however, there have been almost no results on bounding the work (time) of Adams' algorithms, in general nor on specific tree types. Adams informally argues that his algorithms take $O(n+m)$ work for weight-balanced tree, but that is a very loose bound. Blelloch and Reid-Miller later show that similar algorithms for treaps [7], are optimal for work (i.e. $\Theta\big(m \log\big(\frac{n}{m}+1\big)\big)$), 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.[3] One needs additional properties of the trees.

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935768

[1] 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.

[2] Adams' version had some bugs in maintaining the balance, but these were later fixed [16, 29].

[3] 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.

```
split(T, k) =
  if T = Leaf then (Leaf, false, Leaf)
  else (L, m, R) = expose(T);
    if k = m then (L, true, R)
    else if k < m then
      (L_L, b, L_R) = split(L, k);
      (L_L, b, join(L_R, m, R))
    else (R_L, b, R_R) = split(R, k);
      (join(L, m, R_L), b, R_R)

splitLast(T) =
  (L, k, R) = expose(T);
  if R = Leaf then (L, k)
  else (T', k') = splitLast(R);
    (join(L, k, T'), k')

join2(T_L, T_R) =
  if T_L = Leaf then T_R
  else (T'_L, k) = splitLast(T_L);
    join(T'_L, k, T_R)

insert(T, k) =
  (T_L, m, T_R) = split(T, k);
  join(T_L, k, T_R)

delete(T, k) =
  (T_L, m, T_R) = split(T, k);
  join2(T_L, T_R)
```

```
union(T_1, T_2) =
  if T_1 = Leaf then T_2
  else if T_2 = Leaf then T_1
  else (L_2, k_2, R_2) = expose(T_2);
    (L_1, b, R_1) = split(T_1, k_2);
    T_L = union(L_1, L_2) || T_R = union(R_1, R_2);
    join(T_L, k_2, T_R)

intersect(T_1, T_2) =
  if T_1 = Leaf then Leaf
  else if T_2 = Leaf then Leaf
  else (L_2, k_2, R_2) = expose(T_2);
    (L_1, b, R_1) = split(T_1, k_2);
    T_L = intersect(L_1, L_2) || T_R = intersect(R_1, R_2);
    if b = true then join(T_L, k_2, T_R)
    else join2(T_L, T_R)

difference(T_1, T_2) =
  if T_1 = Leaf then Leaf
  else if T_2 = Leaf then T_1
  else (L_2, k_2, R_2) = expose(T_2);
    (L_1, b, R_1) = split(T_1, k_2);
    T_L = difference(L_1, L_2) || T_R = difference(R_1, R_2);
    join2(T_L, T_R)
```

Figure 1: Implementing UNION, INTERSECT, DIFFERENCE, INSERT, DELETE, SPLIT, and JOIN2 with just JOIN. EXPOSE returns the left tree, key, and right tree of a node. The || notation indicates the recursive calls can run in parallel. These are slight variants of the algorithms described by Adams [1], although he did not consider parallelism.

The contribution of this paper is to give the first work-optimal bounds for Adams' algorithms. We do this not only for the weight-balanced trees, but for three other balancing schemes: AVL trees, red-black trees and treaps. We analyze exactly the algorithms in Figure 1. We show that with appropriate (and simple) implementations of JOIN for each balancing scheme, we achieve asymptotically optimal bounds on work. These bounds hold when either input tree is larger (this was surprising to us). Furthermore the algorithms have $O(\log n \log m)$ span (parallel time), and hence are highly parallel. To prove the bounds on work we show that our implementations of JOIN satisfy certain conditions based on a rank we define for each tree type. In particular the cost of JOIN must be proportional to the difference in ranks of two trees, and the rank of the result of a join must be at most one more than the maximum rank of the two arguments.

In addition to the theoretical analysis of the algorithms, we implemented parallel versions of all of the algorithms on all four tree types and describe experiments. Our implementation is generic in the sense that we use common code for the algorithms in Figure 1, and only wrote specialized code for each tree type for the JOIN function. Our implementations of JOIN are as described in this paper. We compare performance across a variety of parameters. We compare across the tree types, and interestingly all four balance criteria have very similar performance. We measure the speedup on up to 64 cores and achieve close to a 46-fold speedup. We compare to other implementations, including the set implementation in the C++ Standard Template library (STL) for sequential performance, parallel weight-balanced B-trees (WBB-trees) [14] and the multi-core standard template library (MCSTL) [15] for parallel performance, and to previously reported results on concurrent balanced trees [11]. The conclusion from the experiments is that although not always as fast as (WBB-trees) [14] on uniform distributions, the generic code is quite competitive, and on keys with a skewed overlap (two Gaussians with different means), our implementation is much better than all the other baselines. Our times are very much faster than the concurrent balanced trees, but this is perhaps not fair since concurrent trees are not asymptotically efficient requiring at least $\Omega(m \log n)$ work instead of $\Theta\big(m \log\big(\frac{n}{m} + 1\big)\big)$.

Related Work.

Parallel set operations on two ordered sets have been well-studied. Paul, Vishkin, and Wagener studied bulk insertion and deletion on 2-3 trees in the PRAM model [26]. Park and Park showed similar results for red-black trees [25]. These algorithms are not based on JOIN and are not work efficient, requiring $O(m \log n)$ work. Katajainen [18] claimed an algorithm with $O\big(m \log(\frac{n}{m} + 1)\big)$ work and $O(\log n)$ span using 2-3 trees, but it appears to contain some bugs in the analysis [7]. 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 [4] recently 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 paper is in showing that very simple algorithms are work efficient and have polylogarithmic span, and less with optimizing the span.

Many researchers have considered concurrent implementations of balanced search trees (e.g., [19, 20, 11, 23]). 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.

2. PRELIMINARIES

A *binary tree* is either a `Leaf`, or a node consisting of a *left* binary tree T_L, a value (or key) v, and a *right* binary tree T_R, and denoted $\texttt{Node}(T_L, v, T_R)$. The *size* of a binary tree, or $|T|$, is 0 for a `Leaf` and $|T_L| + |T_R| + 1$ for a $\texttt{Node}(T_L, v, T_R)$. The *weight* of a binary tree, or $w(T)$, is one more than its size (i.e., the number of leaves in the tree). The *height* of a binary tree, or $h(T)$, is 0 for a `Leaf`, and $\max(h(T_L), h(T_R)) + 1$ for a $\texttt{Node}(T_L, v, T_R)$. *Parent*, *child*, *ancestor* and *descendant* are defined as usual (ancestor and descendant are inclusive of the node itself). The *left spine* of a binary tree is the path of nodes from the root to a leaf always following the left tree, and the *right spine* the path to a leaf following the right tree. The *in-order values* of a binary tree is the sequence of values returned by an in-order traversal of the tree.

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 paper we consider four balancing schemes that ensure the height of every tree of size n is bounded by $O(\log n)$. For each balancing scheme we define the *rank* of a tree, or $r(T)$.

AVL trees [3] have the invariant that for every $\texttt{Node}(T_L, v, T_R)$, the height of T_L and T_R differ by at most one. 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. For AVL trees $r(T) = h(T) - 1$.

Red-black (RB) trees [5] 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. Unlike some other presentations, we do not require that the root of a tree is black. 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 leaf (inclusive of the node). Any RB tree of size n has height at most $2\log_2(n+1)$. In RB trees $r(T) = 2(\hat{h}(T) - 1)$ if T is black and $r(T) = 2\hat{h}(T) - 1$ if T is red.

Weight-balanced (WB) trees with parameter α (also called BB$[\alpha]$ trees) [24] maintain for every $T = \texttt{Node}(T_L, v, T_R)$ the invariant $\alpha \leq \frac{w(T_L)}{w(T)} \leq 1 - \alpha$. We say two weight-balanced trees T_1 and T_2 have *like* weights if $\texttt{Node}(T_1, v, T_2)$ is weight balanced. Any α weight-balanced 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 weight balanced trees using just single and double rotations [24, 8]. We require the same condition for our implementation of JOIN, and in particular use $\alpha = 0.29$ in experiments. For WB trees $r(T) = \lceil \log_2(w(T)) \rceil - 1$.

Treaps [27] associate a uniformly random priority with every node and maintain the invariant that the priority at each node is no greater than the priority of its two children. Any treap of size n has height $O(\log n)$ with high probability (w.h.p)[4]. For treaps $r(T) = \lceil \log_2(w(T)) \rceil - 1$.

For all the four balancing schemes $r(T) = \Theta(\log(|T|+1))$. The notation we use for binary trees is summarized in Table 1.

A *Binary Search Tree* (BST) is a binary tree in which each value is a key taken from a total order, and for which the in-order values are sorted. A *balanced BST* is a BST maintained with a balancing scheme, and is an efficient way to represent ordered sets.

[4]Here w.h.p. means that height $O(c \log n)$ with probability at least $1 - 1/n^c$ (c is a constant).

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
$r(T)$	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 value (or key) of node T
$L(T)$	The left child of node T
$R(T)$	The right child of node T
$\texttt{expose}(T)$	$(L(T), k(T), R(T))$

Table 1: Summary of notation.

Our algorithms are based on nested parallelism with nested fork-join constructs and no other synchronization or communication among parallel tasks.[5] All algorithms are deterministic. We use work W and span (or depth) S to analyze asymptotic costs, where the work is the total number of operations and span is the critical path. We use the simple composition rules $W(e_1 \mid\mid e_2) = W(e_1) + W(e_2) + 1$ and $S(e_1 \mid\mid e_2) = \max(S(e_1), S(e_2)) + 1$. For sequential computation both work and span compose with addition. Any computation with W work and S span will run in time $T < \frac{W}{P} + S$ assuming a PRAM (random access shared memory) with P processors and a greedy scheduler [10, 9].

3. THE JOIN FUNCTION

Here we describe algorithms for JOIN for the four balancing schemes we defined in Section 2. The function $\text{JOIN}(T_L, k, T_R)$ takes two binary trees T_L and T_R, and a value k, and returns a new binary tree for which the in-order values are a concatenation of the in-order values of T_L, then k, and then the in-order values of T_R.

As mentioned in the introduction and shown in Section 4, 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 rank of the two trees. For treaps the work depends on the priority of k. All versions of JOIN are sequential so the span is equal to the work. Due to space limitations, we describe the algorithms, state the theorems for all balancing schemes, but only show a proof outline for AVL trees.

```
1  joinRight(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 Node(l, k', T')
6      else rotateLeft(Node(l, k', rotateRight(T')))
7    else
8      T' = joinRight(c, k, T_R);
9      T'' = Node(l, k', T');
10     if h(T') ≤ h(l) + 1 then T''
11     else rotateLeft(T'')

12 join(T_L, k, T_R) =
13   if h(T_L) > h(T_R) + 1 then joinRight(T_L, k, T_R)
14   else if h(T_R) > h(T_L) + 1 then joinLeft(T_L, k, T_R)
15   else Node(T_L, k, T_R)
```

Figure 2: AVL JOIN algorithm.

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

```
1  joinRightRB(T_L, k, T_R) =
2    if (r(T_L) = ⌊r(T_R)/2⌋ × 2) then
3      Node(T_L, ⟨k, red⟩, T_R);
4    else
5      (L', ⟨k', c'⟩, R')=expose(T_L);
6      T' = Node(L', ⟨k', c'⟩, joinRightRB(R', k, T_R));
7      if (c'=black) and (c(R(T')) = c(R(R(T')))=red) then
8        c(R(R(T')))=black;
9        T''=rotateLeft(T')
10     else T''

11 joinRB(T_L, k, T_R) =
12   if ⌊r(T_L)/2⌋ > ⌊r(T_R)/2⌋ then
13     T' =joinRightRB(T_L, k, T_R);
14     if (c(T')=red) and (c(R(T'))=red) then
15       Node(L(T'), ⟨k(T'), black⟩, R(T'))
16     else T'
17   else if ⌊r(T_R)/2⌋ > ⌊r(T_L)/2⌋ then
18     T' =joinLeftRB(T_L, k, T_R);
19     if (c(T')=red) and (c(L(T'))=red) then
20       Node(L(T'), ⟨k(T'), black⟩, R(T'))
21     else T'
22   else if (c(T_L)=black) and (c(T_R)=black) then
23     Node(T_L, ⟨k, red⟩, T_R)
24   else Node(T_L, ⟨k, black⟩, T_R)
```

Figure 3: RB JOIN algorithm.

```
1  joinRightWB(T_L, k, T_R) =
2    (l, k', c)=expose(T_L);
3    if (balance(|T_L|, |T_R|) then Node(T_L, k, T_R));
4    else
5      T' = joinRightWB(c, k, T_R);
6      (l_1, k_1, r_1) = expose(T');
7      if like(|l|, |T'|) then Node(l, k', T')
8      else if (like(|l|, |l_1|)) and (like(|l| + |l_1|, r_1)) then
9        rotateLeft(Node(l, k', T'))
10     else rotateLeft(Node(l, k', rotateRight(T')))

11 joinWB(T_L, k, T_R) =
12   if heavy(T_L, T_R) then joinRightWB(T_L, k, T_R)
13   else if heavy(T_R, T_L) then joinLeftWB(T_L, k, T_R)
14   else Node(T_L, k, T_R)
```

Figure 4: WB JOIN algorithm.

```
1  joinTreap(T_L, k, T_R) =
2    if prior(k, k_1) and prior(k, k_2) then Node(T_L, k, T_R)
3    else (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
6        Node(l_1, k_1, joinTreap(r_1, k, T_R))
7      else Node(joinTreap(T_L, k, l_2), k_2, r_2)
```

Figure 5: Treap JOIN algorithm.

AVL trees. Pseudocode for AVL JOIN is given in Figure 2 and illustrated in Figure 6. 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 $\texttt{Node}(T_L, k, T_R)$. 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 $\texttt{Node}(c, k, T_R)$ 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 invalid at the parent (line 6) or a single left rotation if invalid higher in the tree (line 11), in both cases restoring the height for any further ancestor nodes. The algorithm will therefore require at most two rotations.

LEMMA 1. *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 outline. Since the algorithm only visits nodes on the path from the root to c, and only requires at most two rotations, 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 (details left out). The height of any node can increase by at most one, so the height of the whole tree can increase by at most one. □

Red-black Trees. Tarjan describes how to implement the JOIN function for red-black trees [30]. Here we describe a variant that does not assume the roots are black (this is to bound the increase in rank by UNION). The pseudocode is given in Figure 3. We store at every node its black height $\hat{h}(\cdot)$. The first case is when $\hat{h}(T_R) = \hat{h}(T_L)$. Then if both $k(T_R)$ and $k(T_L)$ are black, we create red $\texttt{Node}(T_L, k, T_R)$, otherwise we create black $\texttt{Node}(T_L, k, T_R)$. 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 $\texttt{Node}(c, k, T_R)$ 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 left rotation: if a black node v has $R(v)$ and $R(R(v))$ both red, we turn $R(R(v))$ black and perform a single left rotation on v. The update is illustrated in Figure 7. The rotation, however can again violate the red rule between the root of the rotated tree and its parent, requiring another rotation. The double-red issue might proceed up to the root of T_L. 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 $1 + r(T_L)$.

LEMMA 2. *For two RB trees T_L and T_R, the RB* JOIN *algorithm works correctly, runs with $O(|r(T_L) - r(T_R)|)$ work, and returns a tree satisfying the red-black invariants and with rank at most $1 + \max(r(T_L), r(T_R))$.*

The proof is similar as Lemma 1.

Weight Balanced Trees. We store the weight of each subtree at every node. The algorithm for joining two weight-balanced trees is similar to that of AVL trees and RB trees. The pseudocode is shown in Figure 4. The `like` function in the code returns true if

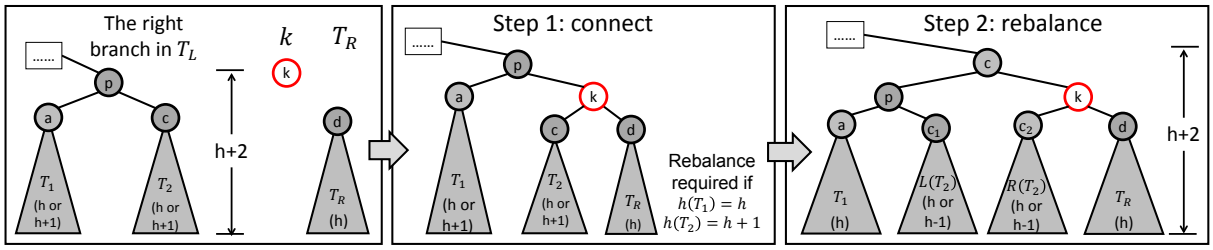

Figure 6: An example for JOIN on AVL trees ($h(T_L) > h(T_R)+1$). We first follow the right spine of T_L until a subtree of height at most $h(T_r)+1$ is found (i.e., T_2 rooted at c). Then a new $\texttt{Node}(c,k,T_R)$ is created, replacing c (Step 1). If $h(T_1) = h$ and $h(T_2) = h+1$, the node p will no longer satisfy the AVL invariant. A double rotation (Step 2) restores both balance and its original height.

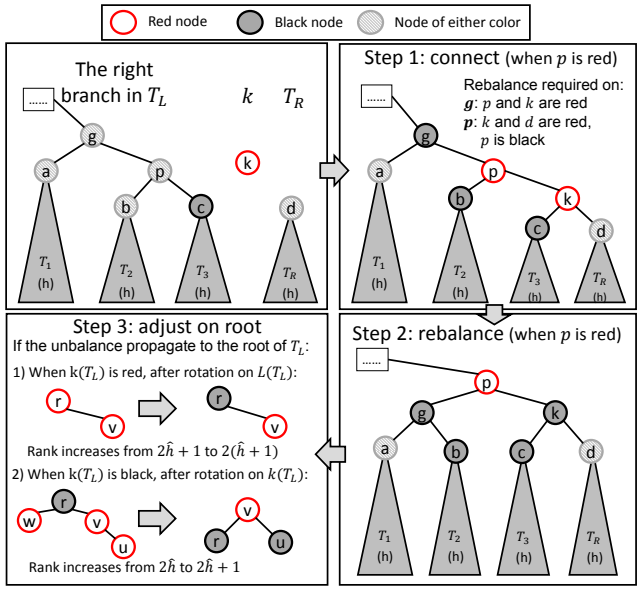

Figure 7: An example of JOIN on red-black trees ($\hat{h} = \hat{h}(T_L) > \hat{h}(T_R)$). We follow the right spine of T_L until we find a black node with the same black height as T_R (i.e., c). Then a new red $\texttt{Node}(c,k,T_R)$ is created, replacing c (Step 1). The only invariant that can be 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).

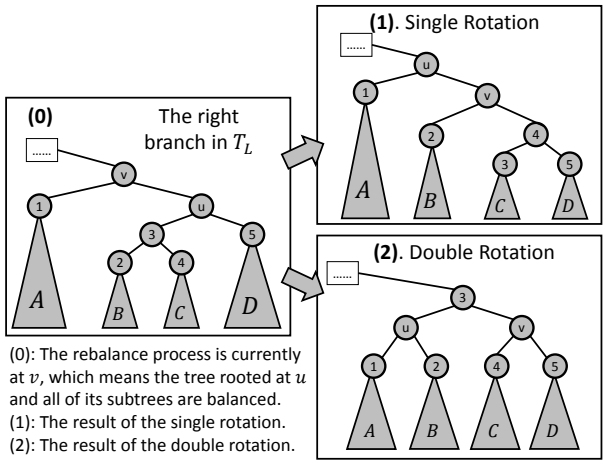

Figure 8: An illustration of single and double rotations possibly needed to rebalance weight-balanced trees. In the figure the subtree rooted at u has become heavier due to joining in T_L and its parent v now violates the balance invariant.

the two input tree sizes are balanced, and false otherwise. If T_L and T_R have like weights the algorithm returns a new $\texttt{Node}(T_L,k,T_R)$. 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 $\texttt{Node}(c,k,T_R)$ 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 8) at each node assuming α is within the bounds given in Section 2.

LEMMA 3. *For two α weight-balanced trees T_L and T_R and $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$, the weight-balanced* JOIN *algorithm works correctly, runs with $O(|\log(w(T_L)/w(T_R))|)$ work, and returns a tree satisfying the α weight-balance invariant and with rank at most $1 + \max(r(T_L), r(T_R))$.*

The proof is shown in the full version of our paper (on arXiV) [6].

Treaps. The treap JOIN algorithm (as in Figure 5) 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 the we can return $\texttt{Node}(T_L,k,T_R)$. Otherwise, WLOG, assume $k(T_L)$ has a higher priority. In this case $k(T_L)$ will be the root of the result, $L(T_L)$ will be the left tree, and $R(T_L)$, k and T_R will form the right tree. Thus JOIN recursively calls itself on $R(T_L)$, k and T_R and uses 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, if a key has the t^{th} highest priority among the keys, then its expected depth in a treap is $O(\log t)$ (also w.h.p.). If it is the highest priority, for example, then it remains at the root.

LEMMA 4. *For two treaps T_L and T_R, if the priority of 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 w.h.p., and returns a tree satisfying the treap invariant with rank at most $1 + \max(r(T_L), r(T_R))$.*

From the above lemmas we can get the following fact for JOIN.

THEOREM 1. *For AVL, RB and WB trees* JOIN(T_L, k, T_R) *does* $O(|r(T_L) - r(T_R)|)$ *work. For treaps* JOIN *does* $O(\log t)$ *work in expectation if* k *has the* t*-th highest priority among all keys. For AVL, RB, WB trees and treaps,* JOIN *returns a tree* T *for which the rank satisfies* $\max(r(T_L), r(T_R)) \leq r(T) \leq 1 + \max(r(T_L), r(T_R))$.

4. OTHER FUNCTIONS USING JOIN

In this section, we describe algorithms for various functions that use just JOIN. The algorithms are generic across balancing schemes. The pseudocodes for the algorithms in this section is shown in Figure 1.

Split. 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 less (larger) than k, and b is a flag indicating whether $k \in T$. The 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. Figure 9 gives an example.

THEOREM 2. *The work of* SPLIT(T, k) *is* $O(\log |T|)$ *for all balancing schemes described in Section 3 (w.h.p. for treaps). The two resulting trees* T_L *and* T_R *will have rank at most* $r(T)$.

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 $T_1, T_2, \ldots T_l$. We have that $r(T_1) \leq r(T_2) \leq \cdots \leq r(T_l)$. 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.

For AVL trees, red-black trees and weight-balanced trees, recall Theorem 1 that we have $r(T_i') \leq r(T_i) + 1$, so $r(T_i') \leq r(T_i) + 1 \leq r(T_{i+1}) + 1$. According to Theorem 1, the work of a single operation is $O(|r(T_{i+1}) - r(T_i')|)$. The overall complexity is $\sum_{i=1}^{l} |r(T_{i+1}) - r(T_i')| \leq \sum_{i=1}^{l} r(T_{i+1}) - r(T_i') + 2 = O(r(T)) = O(\log |T|)$.

For treaps, each JOIN uses the key with the highest priority since the key is always on a upper level. Hence by Lemma 4, the complexity of each JOIN is $O(1)$ and the work of split is at most $O(\log |T|)$ w.h.p.

Also notice that when getting the final result T_L and T_R, the last step is a JOIN on two trees, the larger one of which is a subtree of the original T. Thus the rank of the two trees to be joined is of rank at most $r(T) - 1$, according to Theorem 1 we have $r(T_L)$ and $r(T_R)$ at most $r(T)$. □

Join2. JOIN2(T_L, T_R) returns a binary tree for which the in-order values is 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. JOIN2 first finds the last element k (by following the right spine) in T_L and on the way back to root, joins the subtrees along the path, which is similar to SPLIT T_L by k. We denote the result of dropping k in T_L as T_L'. Then JOIN(T_L', 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 3. *The work of* JOIN2(T_L, T_R) *is* $O(r(T_L)+r(T_R))$ *for all balancing schemes described in Section 3 (bounds are w.h.p for treaps).*

Union, Intersect and Difference. 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 $k(T_2) \notin T_1$), 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 $L(T_2)$ with L_1, returning T_L, and the other one unions $R(T_2)$ 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 INTERSECT(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 is different. For INTERSECT, JOIN2 is used instead of JOIN if the root of the first *is not* found in the second. For DIFFERENCE, 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.

THEOREM 4 (MAIN THEOREM). *For all four balance schemes mentioned in Section 3, the work and span of the algorithm (as shown in Figure 1) of* UNION, INTERSECT *or* DIFFERENCE *on two balanced BSTs of sizes* m *and* n $(n \geq m)$ *is* $O\Big(m \log\Big(\frac{n}{m} + 1\Big)\Big)$ *and* $O(\log n \log m)$ *respectively (the bound is in expectation for treaps).*

A generic proof of Theorem 4 working for all the four balancing schemes will be shown in the next section.

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 $\log \binom{m+n}{n} = \Theta\big(m \log(\frac{n}{m} + 1)\big)$ [17].

Other Functions. Many other functions can be implemented with JOIN. Pseudocode for INSERT and DELETE was given in Figure 1. For a tree of size n they both take $O(\log n)$ work.

5. THE PROOF OF THE MAIN THEOREM

In this section we prove Theorem 4, for all the four balance schemes (AVL trees, RB trees, WB trees and treaps) and all three set algorithms (UNION, INTERSECT, DIFFERENCE) from Figure 1. 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 SPLITs and the JOINs (or JOIN2s), 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 SPLITs—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 2. 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*

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

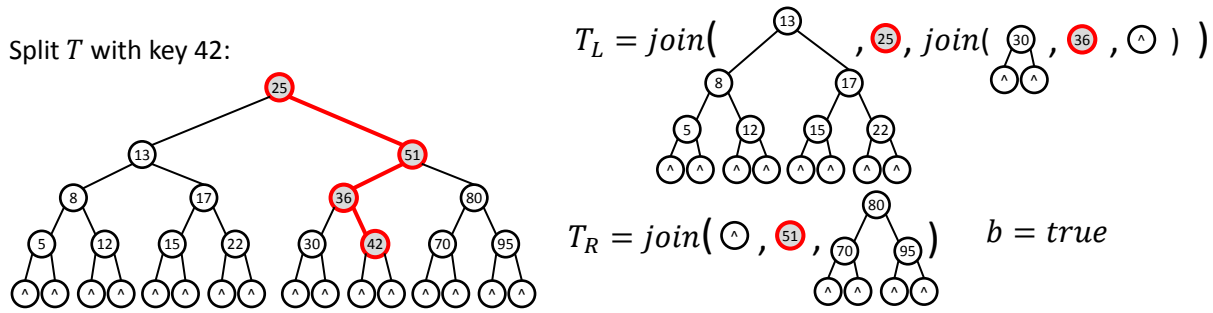

Figure 9: An example of SPLIT in a BST with key 42. We first search for 42 in the tree and split the tree by the searching path, then use JOIN to combine trees on the left and on the right respectively, bottom-top.

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[6]
s_i	The number of nodes in layer i
v_{kj}	The j-th node on layer k in T_p
$d(v)$	The number of nodes attached to a layer root v in a treap

Table 2: Descriptions of notations used in the proof.

tree and the second the *pivot tree*, denoted as T_d and T_p respectively. The tree that is returned is denoted as T_r. 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 that v splits as $T_d(v)$ (i.e., $\text{SPLIT}(v, T_d(v))$ is called at some point in the algorithm). For $v \in T_p$, we refer to $|T_d(v)|$ as its *splitting size*.

Figure 10 (a) illustrates the pivot tree with the splitting size annotated on each node. Since SPLIT has logarithmic work, we have,

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

which we analyze in Theorem 6. We first, however, show that the join work is bounded by the split work. We use the following Lemma, which is proven in the appendix.

LEMMA 5. *For T_r =UNION(T_p, T_d) on AVL, RB or WB trees, if $r(T_p) > r(T_d)$ then $r(T_r) \le r(T_p) + r(T_d)$.*

THEOREM 5. *For each function call to* UNION, INTERSECT *or* DIFFERENCE *on trees $T_p(v)$ 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 INTERSECT or DIFFERENCE, the cost of JOIN (or JOIN2) is $O(\log(|T_r| + 1))$. Notice that DIFFERENCE returns the keys in $T_d \backslash T_p$. Thus for both INTERSECT and DIFFERENCE we have $T_r \subseteq T_d$. The join work is $O(\log(|T_r| + 1))$, which is no more than $O(\log(|T_d| + 1))$ (the split work).

For UNION, if $r(T_p) \le r(T_d)$, the JOIN will cost $O(r(T_d))$, which is no more than the split work.

Consider $r(T_p) > r(T_d)$ for AVL, RB or WB trees. The rank of $L(T_p)$ and $R(T_p)$, which are used in the recursive calls, are at least $r(T_p) - 1$. Using Lemma 5, the rank of the two trees returned by the two recursive calls will be at least $(r(T_p) - 1)$ and at most $(r(T_p) + r(T_d))$, and differ by at most $O(r(T_d)) = O(\log |T_d| + 1)$. Thus the join cost is $O(\log |T_d| + 1)$, which is no more than the split work.

Consider $r(T_p) > r(T_d)$ for treaps. If $r(T_p) > r(T_d)$, then $|T_p| \ge |T_d|$. The root of T_p has the highest priority among all $|T_p|$ keys, so on expectation it takes at most the $\frac{|T_p|+|T_d|}{|T_p|} \le$ 2-th place among all the $|T_d| + |T_p|$ keys. From Lemma 4 we know that the cost on expectation is $\mathbb{E}[\log t] + 1 \le \log \mathbb{E}[t] + 1 \le \log 2 + 1$, 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 define layers based on the ranks and denote the size of layer i as s_i. We show that s_i shrinks geometrically, which helps us prove our bound on the split work. For AVL and RB trees, we group the nodes with rank i in

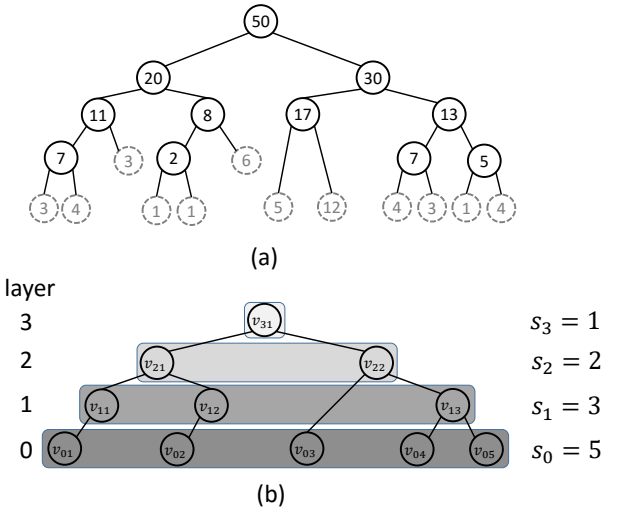

Figure 10: An illustration of splitting tree and layers. The tree in (a) is T_p, the dashed circle are the exterior nodes. The numbers on the nodes are the sizes of the tree from T_d to be split by this node, i.e., the "splitting size" $|T_d(v)|$. In (b) is an illustration of layers on an AVL tree.

layer i. For WB trees and treaps, we put a node v in layer i iff v has rank i and v's parent has rank strictly greater than i. Figure 10 (b) shows an example of the layers of an AVL tree.

DEFINITION 1. *In a BST, a set of nodes V is called a* disjoint set *if and only if for any two nodes v_1, v_2 in V, v_1 is not the ancestor of v_2.*

LEMMA 6. *For any disjoint set $V \subseteq T_p$, $\sum_{v\in V} |T_d(v)| \le |T_d|$.*

The proof of this Lemma is straightforward.

LEMMA 7. *For an AVL, RB, WB tree or a treap of size N, each layer is a disjoint set, and $s_i \le \frac{N}{c^{\lfloor i/2 \rfloor}}$ holds for some constant $c > 1$.*

PROOF. For AVL, RB, WB trees and treaps, a layer is obviously a disjoint set: a node and its ancestor cannot lie in the same layer.

For AVL trees, consider a node in layer 2, it must have at least two descendants in layer 0. Thus $s_0 \ge 2s_2$. Since an AVL tree with its leaves removed is still an AVL tree, we have $s_i \ge 2s_{i+2}$. Since s_0 and s_1 are no more than N, we can get that $s_i < \frac{N}{2^{\lfloor i/2 \rfloor}}$.

For RB trees, the number of black nodes in layer $2i$ is more than twice as many as in layer $2(i+1)$ and less than four times as many as in layer $2(i+1)$, i.e., $s_{2i} \ge 2s_{2i+2}$. Also, the number of red nodes in layer $2i+1$ is no more than the black nodes in layer $2i$. Since s_0 and s_1 are no more than N, $s_i < \frac{N}{2^{\lfloor i/2 \rfloor}}$.

For WB trees and treaps, the rank is defined as $\lceil \log_2(w(T)) \rceil - 1$, which means that a node in layer i has weight at least $2^i + 1$. Thus $s_i \le (N+1)/(2^i+1) \le N/2^i$. □

Not all nodes in a WB tree or a treap are assigned to a layer. We call a node a *layer root* if it is in a layer. We attach each node u in the tree to the layer root that is u's ancestor and has the same rank as u. We denote $d(v)$ as the number of descendants attached to a layer root v.

LEMMA 8. *For WB trees and treaps, if v is a layer root, $d(v)$ is less than a constant (in expectation for treaps). Furthermore, the random variables $d(v)$ for all layer roots in a treap are i.i.d. (See the proof in the Appendix.)*

By applying Lemma 7 and 8 we prove the split work. In the following proof, we denote v_{kj} as the j-th node in layer k.

THEOREM 6. *The split work in* UNION, INTERSECT *and* DIFFERENCE *on two trees of size m and n is $O\big(m \log\big(\frac{n}{m}+1\big)\big)$.*

PROOF. The total work of SPLIT is the sum of the log of all the splitting sizes on the pivot tree $O\Big(\sum_{v\in T_p} \log(|T_d(v)|+1)\Big)$. 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 $s_i \le |T_d|$. We calculate the dominant term $\sum_{v\in T_p} \log(|T_d(v)|+1)$ in the complexity by summing the work across layers:

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

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

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

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

$$\le 2\sum_{k=0}^{l/2} \frac{m}{c^k} \log\frac{n}{m/c^k} \le 2\sum_{k=0}^{l/2} \frac{m}{c^k} \log\frac{n}{m} + 2\sum_{k=0}^{l/2} k\frac{m}{c^k} = O\Big(m\log\frac{n}{m}\Big) + O(m) = O\Big(m\log\Big(\frac{n}{m}+1\Big)\Big) \quad (3)$$

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 $s_k \le |T_d|$, and we have:

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

$$\le \sum_{k=0}^{2\log_c \frac{n}{m}} |T_d| \log(1+1) + \sum_{k=2\log_c \frac{n}{m}}^{l} \frac{n}{c^{\lfloor k/2 \rfloor}} \log\left(\frac{m}{n/c^{\lfloor k/2 \rfloor}}+1\right) \quad (5)$$

$$\le O\Big(m\log\frac{n}{m}\Big) + 2\sum_{k'=0}^{\frac{l}{2}-\log_c \frac{m}{n}} \frac{m}{c^{k'}} \log c^{k'} = O\Big(m\log\frac{n}{m}\Big) + O(m) = O\Big(m\log\big(\frac{n}{m}+1\big)\Big) \quad (6)$$

From (1) to (2) and (4) to (5) we apply Lemma 7 and the fact that $f(x) = x\log(\frac{n}{x}+1)$ is monotonically increasing when $x \le n$.

For WB trees and treaps, the calculation above only includes the log of splitting size on layer roots. We need to further prove the total sum of the log of all splitting size is still $O\big(m\log\big(\frac{n}{m}+1\big)\big)$. Applying Lemma 8, the expectation is less than:

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

For WB trees $d(v_{kj})$ is no more than a constant, so we can also come to the same bound.

To conclude, the split work on all four balancing schemes of all three functions is $O\big(m\log\big(\frac{n}{m}+1\big)\big)$. □

THEOREM 7. *The total work of* UNION, INTERSECT *or* DIFFERENCE *of all four balancing schemes on two trees of size m and n ($m \geq n$) is $O\big(m \log\big(\frac{n}{m}+1\big)\big)$.*

This directly follows Theorem 5 and 6.

THEOREM 8. *The span of* UNION *and* INTERSECT *or* DIFFERENCE *on all four balancing schemes is $O(\log n \log m)$. Here n and m are the sizes of the two tree.*

PROOF. For the span of these algorithms, we denote $D(h_1, h_2)$ as the span on UNION, INTERSECT or DIFFERENCE on two trees of height h_1 and h_2. According to Theorem 5, 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 $D(h(T_p), h(T_d)) \leq 2h(T_p)h(T_d) = O(\log n \log m)$. □

Combine Theorem 7 and 8 we come to Theorem 4.

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.

Experiment setups and baseline algorithms.

For the experiments we use a 64-core machine with 4 x AMD Opteron(tm) Processor 6278 (16 cores, 2.4GHz, 1600MHz bus and 16MB L3 cache). Our code was compiled using the `g++` 4.8 compiler with the Cilk Plus extensions for nested parallelism. The only compilation flag we used was the `-O2` optimization flag. In all our experiments we use keys of the double data type. The size of each node is about 40 bytes, including the two child pointers, the key, the balance information, the size of the subtree, and a reference count. We generate multiple sets varying in size from 10^4 to 10^8. Depending on the experiment the keys are drawn either from an uniform or a Gaussian distribution. We use μ and σ to denote the mean and the standard deviation in Gaussian distribution. Throughout this section n and m represent the two input sizes for functions with two input sets ($n \geq m$).

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) [22] 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 [14] and the MCSTL library [15], both supporting array-tree UNION.

Comparing the balancing schemes and functions.

To compare the four balancing schemes we choose UNION as the representative operation. Other operations give similar results. We compare the schemes across varying thread counts and sizes.

Figure 11 (a) shows the runtime of UNION for varying tree sizes and all four balancing schemes on 64 cores. The times are very similar across the balancing schemes, differing by no more than 10%.

Figure 11 (b) shows the speedup curves for UNION on a varying number of cores with both inputs of size 10^8. All balancing schemes achieve a speedup of about 45 on 64 cores, and about 30 on 32 cores. The less-than-linear speedup beyond 32 cores is not due to lack of parallelism, since when we ran the same experiments on significantly smaller input (and hence less parallelism) we get very similar curves (not shown). Instead it seems to be due to saturation of the memory bandwidth.

We use the AVL tree as the representative tree to compare different functions. Figure 11 (c) compares time for the UNION, INTERSECT and DIFFERENCE functions. The size of the larger tree is fixed (10^8), while the size of the smaller tree varies from 10^4 to 10^8. As the plot indicates, the three functions have very similar performance.

The experiments are a good indication of the performance of different balancing schemes and different functions, while controlling other factors. The conclusion is that all schemes perform almost equally on all the set functions. 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. The AVL tree is always slightly faster than the other trees and this is likely due to the fact that they maintain a slightly stricter balance than the other trees, and hence the paths that need to be traversed are slightly shorter. For different set functions the similar performance is also as expected given the similarity of the code.

Comparing to sequential implementations.

The STL supports `set_union`, `set_intersection`, and `set_difference` on any container class, including sets based on red-black 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 and m, $m \leq n$, it takes $O(m+n)$ time on `std::vectors` by moving from left to right on the two inputs, comparing the current values, and writing the lesser to the end of the output, and $O((n+m)\log(n+m))$ time on `std::set` by inserting the elements in one set into the other. In the case of `std::set` we can do better by inserting elements from the smaller set into the larger, leading a time of $O(m \log(n+m))$. This is also what we do in our experiments. For vectors we stick with the available `set_union` implementation.

Figure 11 (d) gives a comparison of times for UNION. For equal lengths our implementation is about a factor of 3 faster than the set variant (red-black trees), and about 8 times slower than the vector variant. This is not surprising since we are asymptotically faster than their red-black tree implementation, and their 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. 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 ($O(m+n)$ and $O(m \log(m+n))$ is worse than our $O(m \log(n/m+1))$, which is optimal in the comparison model.

Comparing to parallel implementations..

To see how well our algorithm performs in a parallel setting, we compare it to parallel WBB-trees [14] and the MCSTL library [15]. WBB-trees, as well as the MCSTL, offer an interface for bulk insertions and deletions, which take a tree and a sorted array and either insert the elements from the array into the tree, or delete them. Bulk effectively gives a UNION, which we refer to this as an array-tree UNION as opposed to our symmetric tree-tree UNION. If the array is the smaller input, array-tree unions have an inherent advantage over tree-tree unions since accessing an array is much more cache efficient than accessing an tree. Also, the WBB-tree itself has a more cache-aware layout (8 keys per cache line as opposed to 1), leading to a better cache utilization compared to both the MCSTL and our implementation. In this section, we test the

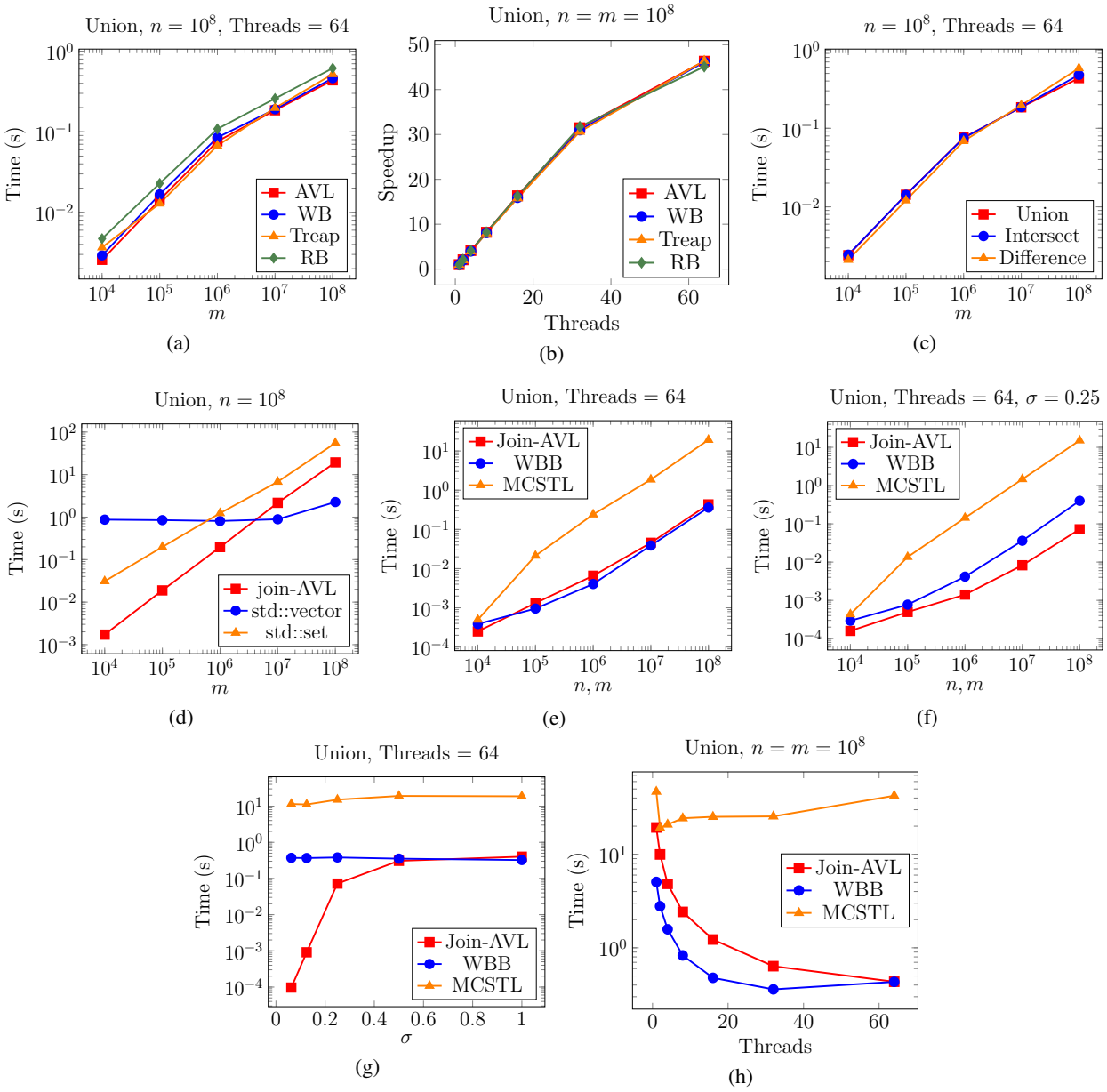

Figure 11: (a) Times for UNION as a function of size ($n = 10^8$) for different BBSTs; (b) speed up of UNION for different BBSTs; (c) times for various operations on AVL trees as a function of size ($n = 10^8$); (d) comparing STLs `set_union` with our UNION; (e, f, g, h) comparing our UNION to other parallel search trees; (e, h) input keys are uniformly distributed doubles in the range of [0, 1]; (f, g) inputs keys follow a normal distribution of doubles - the mean of the main tree is always $\mu_1 = 0$, while the mean of the bulk is $\mu_2 = 1$. Figure (f) uses a standard deviation of $\sigma = 0.25$, while Figure (g) shows the performance across different standard deviations.

performance of our implementation of JOIN-based UNION, WBB-trees and the MCSTL library on different input distributions.

In Figure 11 (e) we show the result of UNION on uniformly distributed doubles in the range of [0,1] across 64 cores. We set the input size to $n = m = 10^i$, i from 4 to 8. The three implementations have similar performance when $n = m = 10^4$. As the input size increases, MCSTL shows much worse performance than the other two because of the lack of parallelism (Figure 11 (h) is a good indication), and the WBB-tree implementation is slightly better than ours. This is likely due to better cache performance as discussed above.

Figure 11 (f) shows the result of a Gaussian distribution with doubles, also on all 64 cores with set sizes of $n = m = 10^i$ for $i = 4$ through 8. The distributions of the two sets have means at 0 and 1 respectively, and both having a standard deviation of 0.25, meaning that the data in the two sets have less overlap comparing to a uniform distribution (as in (e)). In this case our code achieves better performance than the other two implementations. For our algorithms less overlap in the data means more parts of the trees will be untouched, and therefore less nodes will be operated on. This in turn leads to less processing time.

We also do a study on how the overlap of the data sets affects the performance of each algorithm. We generate two sets of size $n = m = 10^8$, each from a Gaussian distribution. The distributions of the two sets have means at 0 and 1 respectively, and both have an equal standard deviation varying in $\{1, 1/2, 1/4, 1/8, 1/16\}$. The different standard deviations are to control the overlap of the two sets, and ideally less overlap should simplify the problem. Figure 11 (g) shows the result of the three parallel implementations on a Gaussian distribution with different standard deviations. From the figure we can see that MCSTL and WBB-tree are not affected by different standard deviations, while our join-based union takes advantage of less overlapping and achieves a much better performance when σ is small. This is not surprising since when the two sets are less overlapped, e.g., totally disjoint, our UNION will degenerate to a simple JOIN, which costs only $O(\log n)$ work. This behavior is consistent with the "adaptive" property (not always the worst-case) in [13]. This indicates that our algorithm is the only one among the three parallel implementations that can detect and take advantage of less overlapping in data, hence have a much better performance when the two operated sets are less overlapped.

We also compare the parallelism of these implementations. In Figure 11 (h) we show their performance across 64 cores. The inputs are both of size 10^8, and generated from an uniform distribution of doubles. It is easy to see that MCSTL does not achieve good parallelism beyond 16 cores, which explains why the MCSTL always performs the worst on 64 cores in all settings. As we mentioned earlier, the WBB-tree are slightly faster than our code, but when it comes to all 64 cores, both algorithms have similar performance. This indicates that our algorithm achieves better parallelism.

To conclude, in terms of parallel performance, our code and WBB-trees are always much better than the MCSTL because of MCSTL's lack of parallelism. WBB-trees achieve a slightly better performance than ours on uniformly distributed data, but it does not improve when the two sets are less overlapped. Thus our code is much better than the other two implementations on less overlapped data.

Comparing to a concurrent data structure.

We also compare our algorithm to an implementation of concurrent AVL trees [11]. UNION can be implemented by concurrent insertions of the smaller set into a larger set, and hence requires $\Omega(m \log n)$ work. We could not get their open-source implementation to run efficiently, so we compare to their previously reported times. As reported in Figure 15, top left of [11], they process 7 million keys per second (insertions and deletions) on 16 threads (for 1 million keys). Ours does 80 million keys per second when unioning two sequences each of length 1-10 million, again on 16 threads. This is 11x faster than theirs.[7] We note, however, that this comparison is not really fair since they support concurrent operations (searches, insertions, deletions, etc.), which we do not.

7. CONCLUSIONS

In this paper, we study ordered sets implemented with balanced binary search trees. We show for the first time that a very simple "classroom-ready" set of algorithms due to Adams' are 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, and even the JOINs are quite simple, as simple as INSERT or DELETE. It seems it is not sufficient to give a time bound to JOIN and base analysis on it. Indeed if this were the case it would have been done years ago. Instead our approach defines the notion of a rank (differently for different trees) and shows invariants on the rank. It is important that the cost of JOIN is proportional to the difference in ranks. It is also important that when joining two trees the resulting rank is only a constant bigger than the larger rank of the inputs. This insures that when joins are used in a recursive tree, as in UNION, the ranks of the results in a pair of recursive calls does not differ much on the two sides. This then ensures that the set functions are efficient.

We also test the performance of our algorithm. Our experiments show that our sequential algorithm is about 3x faster for union on two maps of size 10^8 compared to the STL red-black tree implementation. In parallel settings our code is much better than the two baseline algorithms (MCSTL and WBB-tree) on less overlapped data, while still achieves similar performances with WBB-tree when the two sets are more intermixed. Our code also achieves 45x speedup on 64 cores.

[7]Although on different processors, their processors are faster (2.66Ghz vs. 2.4Ghz).

Acknowledgments

This research was supported in part by NSF grants CCF-1314590 and CCF-1533858, and the Intel Science and Technology Center for Cloud Computing.

8. REFERENCES

[1] S. Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992.

[2] S. Adams. Efficient sets—a balancing act. *Journal of functional programming*, 3(04):553–561, 1993.

[3] G. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proc. of the USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962.

[4] Y. Akhremtsev and P. Sanders. Fast parallel operations on search trees. *arXiv preprint arXiv:1510.05433*, 2015.

[5] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[6] G. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. *arXiv preprint arXiv:1602.02120*, 2016.

[7] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, 1998.

[8] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, 1980.

[9] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1):202–229, 1998.

[10] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974.

[11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[12] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979.

[13] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000.

[14] S. Erb, M. Kobitzsch, and P. Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *Experimental Algorithms*, pages 111–122. Springer, 2014.

[15] L. Frias and J. Singler. Parallelization of bulk operations for STL dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007*, pages 49–58, 2007.

[16] Y. Hirai and K. Yamamoto. Balancing weight-balanced trees. *Journal of Functional Programming*, 21(03):287–307, 2011.

[17] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. on Computing*, 1(1):31–39, 1972.

[18] J. Katajainen. Efficient parallel algorithms for manipulating sorted sets. In *Proceedings of the 17th Annual Computer Science Conference*. University of Canterbury, 1994.

[19] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[20] K. S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, 2000.

[21] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[22] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[23] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, 2014.

[24] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973.

[25] H. Park and K. Park. Parallel algorithms for red–black trees. *Theoretical Computer Science*, 262(1):415–435, 2001.

[26] W. J. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries in 2-3 trees. In *Proc. Intl. Colloq. on Automata, Languages and Programming (ICALP)*, pages 597–609, 1983.

[27] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[28] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[29] M. Straka. Adams' trees revisited. In *Trends in Functional Programming*, pages 130–145. Springer, 2012.

[30] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

APPENDIX

A. PROOFS FOR SOME LEMMAS

A.1 Proof for Lemma 8

PROOF. One observation in WB trees and treaps is that all nodes attached to a single layer root form a chain. This is true 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 $i+1$.

For a layer root v in a WB tree on layer k, $w(v)$ is at most 2^{k+1}. Considering the balance invariant that its child has weight at most $(1-\alpha)w(v)$, the weight of the t-th generation of its descendants is no more than $2^{k+1}(1-\alpha)^t$. This means that after $t^* = \log_{\frac{1}{1-\alpha}} 2$ generations, the weight should decrease to less than 2^k. Thus $d(v) \leq \log_{\frac{1}{1-\alpha}} 2$, which is a constant.

For treaps consider a layer root v on layer k 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:

$$P = \frac{1}{2^k} \sum_{i=2^k+1}^{N} \frac{i-2^k}{i} \tag{7}$$

$$\leq \frac{1}{2^k} \sum_{i=2^k+1}^{2^{k+1}} \frac{i-2^k}{i} \tag{8}$$

$$\approx 1 - \ln 2 \tag{9}$$

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 shrink geometrically as $d(v)$ increase. Thus the expected value of $d(v)$ is a constant.

Since treaps come from a random permutation, all $s(v)$ are i.i.d. □

A.2 Proof for Lemma 5

PROOF. We are trying to show that for $T_r = \text{UNION}(T_p, T_d)$ on AVL, RB or WB trees, if $r(T_p) > r(T_d)$ then $r(T_r) \leq r(T_p) + r(T_d)$.

For AVL and RB trees we use induction on $r(T_p)+r(T_d)$. When $r(T_d)+r(T_p) = 1$ the conclusion is trivial. If $r = r(T_p) > r(T_d)$, T_p will be split into two subtrees, with rank at most $r(T_p)-1$ since we remove the root. T_d will be split into two trees with height at most $r(T_d)$ (Theorem 2). Using the inductive hypothesis, the two recursive calls will return two trees of height at most $r(T_p)-1+r(T_d)$. The result of the final JOIN is therefore at most $r(T_p)+r(T_d)$.

For WB trees, $|T| \leq |T_p| + |T_d| \leq 2|T_p|$. Thus $r(T) \leq r(T_p) + 1 \leq r(T_p) + r(T_d)$. □