Functional Set Operations with Treaps

Dan Blandford and Guy Blelloch

July 23, 2001

1 Abstract

We present a parallel functional algorithm for finding the union of two ordered sets using a variant of random balanced binary search trees (treaps). This algorithm is optimal with respect to the Block Metric bound of Carlsson et al. [7] which states that, given a structure which supports split and join operations in $O(\lg(\min(a,b)))$ work, where a and b are the sizes of the pieces involved, it is optimal to find the union of two lists in $O(k\lg(\frac{n}{k}+1))$ work, where k is the least possible number of blocks that we can break the two lists into before reforming them into one list. The parallel depth of our algorithm is $O(\lg^2(n))$.

In order to obtain a data structure with the required work bounds for split and join, we apply Tarjan's technique of "heterogeneous finger search trees" to a functional implementation of treaps. This results in a particularly simple implementation, although it does mean that our bounds are all expected-case. If, for example, the functional finger search trees developed by Kaplan and Tarjan were used instead, we believe this would lead to worst-case rather than expected-case work bounds.

A full implementation of our algorithms is available on the web at http://www.cs.cmu.edu/(location).

2 Introduction

A useful technique for developing parallel algorithms is functional programming [13]. In functional programming, data is never mutated; instead, when a data structure needs to be changed, a new structure is created which shares memory with the original. Such a structure is automatically fully persistent - all versions of that data structure since it was created can coexist at the same time, sharing memory where appropriate. In fact, it is possible for a data structure to have "multiple futures" - one copy of the structure can be modified in different ways to create new structures, all of which share memory with the original so as to conserve space. This is a great advantage when working with multiple processors - rather than copying the entire data structure multiple times so that each processor can have its own copy, the processors merely need to copy the parts of the data structure that they modify.

An interesting application for persistent data structures is the construction of search engines. For each word in the domain of a search engine there may be a great number of corresponding documents. If a user enters a query that is an AND or OR of two or more words, the search engine needs to find the union or intersection of the corresponding sets. The search engine cannot modify its original list of documents when carrying out this computation, and copying both lists might be prohibitively expensive. If the lists are stored in a persistent data structure, the new list can share data with the old ones in order to use as little memory as possible.

Our goal, then, is to find a persistent data structure that supports the set operations (union, difference, and intersection) as efficiently as possible.

In general, performing set operations on two sorted lists X and Y of size n and m (with $n \geq m$) requires $O(m \lg(\frac{n}{m} + 1))$ comparisons. If we make some assumptions about the inputs, though, these bounds can be improved. Carlsson, Levcopoulos and Petersson [7] showed that, given a data structure which supports split and join in $O(\lg(\min(|T_1|, |T_2|)))$ time, a lower bound for the list merging problem is $O(k \lg(\frac{n}{k} + 1))$, where $k = Block(X, Y)$ is the number of blocks that the lists need to be broken into before being recombined into one list. $Block(X, Y)$ is defined more formally in Section 3.1. An example of a serial algorithm that meets that bound is described in Section 4.1.

In order to meet the $O(k \lg(\frac{n}{k} + 1))$ bound of Carlsson et al. we need a functional representation of a sorted list that supports split and join in $O(\lg(\min(|T_1|, |T_2|)))$ time. Kaplan and Tarjan [16] developed such a structure using 2-3 trees based on their previous work involving recursive slow-down [17]. (In fact, they have developed functional double-ended queues that support join in $O(1)$ time [15, 18], but unfortunately these structures do not admit an $O(\lg(\min(|T_1|, |T_2|)))$ time split operation.)

Kaplan and Tarjan's data structure is described in some detail in Section 9. With some modifications it can be used to implement our union-finding algorithms. However, the implementation is somewhat complex.

In this paper we develop a functional data structure based on Treaps [2]. For the serial union-find algorithm presented in Section 4.1 we require only the trivial technique of flipping the pointers along the left spine of a Treap. Our parallel algorithms require that we flip the pointers along both spines. All of our algorithms require $O(k \lg(\frac{n}{k} + 1))$ expected work, and our parallel algorithms operate in $O(\lg^2(n))$ depth.

2.1 Related Work

The technique of performing searches starting at the top or bottom of a data structure is a form of finger searching [11]. The idea of finger searching is that, given a pointer (or "finger") to any key in a data structure, it should be possible to find nearby keys in faster than the usual $O(\lg(n))$ time.

Many types of balanced trees permit finger searches using parent pointers. Unfortunately, parent pointers imply pointer cycles, which are impossible to construct using purely functional programming. The solution we use for this problem is to flip the pointers along both spines of the tree - each node along the left spine points to its parent instead of its left child, and each node along the right spine points to its parent instead of its right child. Tarjan and Van Wyck [23] describe this data structure as a "heterogeneous finger search tree".

Cole et al. [8] showed that splay trees can be used to implement finger searches. Splay trees can easily be made functional, and this technique is much simpler than the others mentioned, but the bounds are amortized, which is a weak bound for persistent data structures that may have multiple futures.

The problem of merging two lists of size n and m $(n \geq m)$ has been well studied. Brown and Tarjan gave the first $\theta(m \lg(\frac{n}{m} + 1))$ algorithm that outputs the result in the same format as the inputs [5]. Their algorithm was based on finger searching in AVL-trees and they later showed a variant based 2-3 trees [6]. The same bounds can also be achieved with skip-lists [21].

In the parallel setting, previous work has focused either on merging algorithms that take $O(n)$ work [1, 9, 12, 24, 10, 19] and are optimal when the two sets have nearly equal sizes or on multi-insertion algorithms that take $O(m \lg(n + 1))$ work and are optimal when the input values to be inserted are not presorted [20, 14, 22, 3].

More recently, Blelloch and Reid-Miller [4] demonstrated a non-functional parallel merging algorithm which operates in $O(k \lg(\frac{n}{k} + 1))$ work and $O(\lg(m) \lg(n))$ depth using parent pointers. A functional version of this algorithm requires $O(m \lg(\frac{n}{m} + 1))$ work, though the depth does improve to $O(\lg(n))$.

3 Definitions

3.1 The Block Metric

Carlsson, Levcopoulos, and Petersson [7] define the Block Metric measure of the presortedness of two lists as follows:

Definition 1 *Let X and Y be two sorted sequences of length n and m, respectively, and let $Z = < z_1, ..., z_{n+m} >$ be the resulting merged sequence. Then*

$$Block(X, Y) = ||\{i | 1 \leq i < n + m \text{ and } z_i \in X \text{ and } z_{i+1} \in Y \text{ or } z_i \in Y \text{ and } z_{i+1} \in X\}|| + 1$$

Thus *Block* measures the number of blocks in the input sequences that appear in the merged sequence. Note that, if X and Y share some elements, those elements will appear multiple times in Z and will be counted as belonging to both lists. For example, if $X = \{1, 2\}$ and $Y = \{2, 3, 4\}$ then $Z = \{1, 2, 2, 3, 4\}$. Thus $z_2 \in X$ and $z_2 \in Y$, and $Block(X, Y) = 4$. In other words, any duplicate element is considered to belong to a block of its own.

Carlsson et al. showed that, given a data structure which supports splits and joins in $O(\min(d, n - d))$ and $O(\min(|T_1|, |T_2|))$, a lower bound for the list merging problem is $O(k \lg(\frac{n}{k} + 1))$, where $k = Block(X, Y)$. We will present a serial algorithm that meets that bound in Section 4.1 and a parallel algorithm that meets that bound in Section 6. First, though, we must develop the appropriate data structures.

3.2 Treaps

A treap is a kind of balanced tree in which each key has an associated random priority. In addition to the standard binary-tree ordering of the keys, a treap maintains the invariant that every parent has a higher priority than either of its children. When both the keys and priorities in a treap are unique, there is a unique treap containing those keys and priorities, regardless of the order in which those keys were inserted.

We use two operations to manipulate treaps. The first operation, SPLIT, takes a treap T and a key v, and returns a triple (T_1, v', T_2), such that all keys in T_1 are less than v, all keys in T_2 are greater than v, and v' is a duplicate of v if v was in T. The second operation, JOIN, takes two treaps T_1 and T_2 such that all keys in T_1 are less than any key in T_2. It returns a single treap T containing the keys in T_1 and T_2. Both of these operations run in $O(\lg(n))$ expected work.

3.3 Biased and Unbiased Treaps

Since all of the priorities in a treap T are drawn from the same distribution, any key in a treap has an equal chance to be the root. When we split a treap T on a key v, if we have not looked at T beforehand then we know nothing about the priorities of the keys in the resulting treaps T_1 and T_2. Thus those priorities are still random, and so all of the properties of treaps still hold for T_1 and T_2. We say that such a treap is *unbiased*.

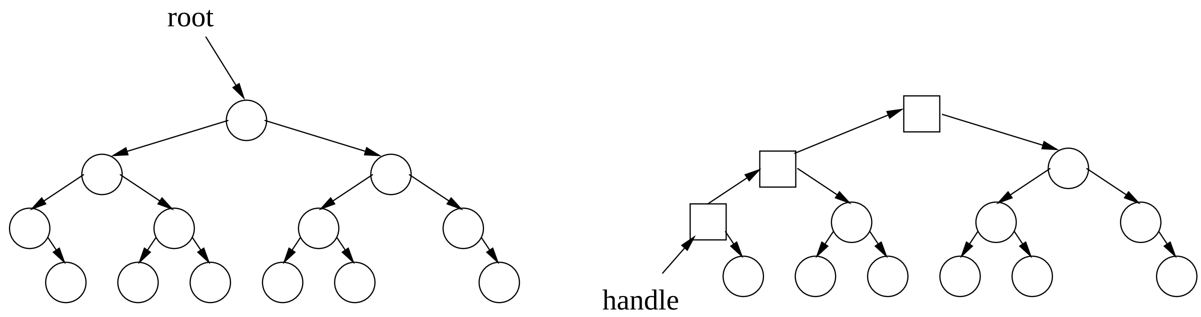

Figure 1: An LTreap is a Treap in which the pointers along the left spine are flipped.

Definition 2 *A treap is* unbiased *if the priorities of each of its keys are drawn from the same distribution. (That is, the priorities are identically and independently distributed.)*

However, if we look at the priorities in T before choosing our split, we have to be careful not to introduce bias into the result. For example, if we use a rule in which we always split T on the rightmost possible key which is still to the left of the root, then the root of our result treap T_2 will always be the leftmost element of that treap. In our analysis we can assume that the initial input treaps to our algorithms are unbiased, but whenever we choose where to split a treap based on knowledge of its priorities we will need to show that the result is unbiased before making any assumptions about its balance.

4 The Left Spinal Treap

The first new data structure we define is the left spinal treap, abbreviated LTreap. An LTreap is a treap in which the pointers along the left spine are flipped - that is, each node on the left spine has a pointer to its right child and a pointer to its parent. (We present an example in Figure 1.) We maintain a handle into the structure at the leftmost node, and all searches in the structure are made from that node. As we will show, this allows us to make finger searches from the leftmost point of the LTreap.

We use four functions to manipulate our LTreap data structures. The first function, TOLTREAP, converts a standard treap into an LTreap; the second, FROMLTREAP, converts an LTreap into a Treap. The third function, JOINLEFT, joins a Treap onto an LTreap provided that the largest key in the Treap is less than the smallest key in the LTreap. The fourth function, SPLITLEFT, splits an LTreap on a given key v. It returns a triple consisting of a Treap containing keys less than v, the key v if it was found, and an LTreap containing keys greater than v. Pseudocode for these functions can be found in figures 2, 3, 4, and 5.

For convenience, throughout our pseudocode we will use T, T', T_1, etc, to refer to Treaps, and L, L', L_1, etc, to refer to LTreaps.

```
toLTreapHelper(T, L)
  if T = null then
    return L
  else
    L′ ← new LTreap(T.key, T.right, L)
    return toLTreapHelper (T.left, L′)

toLTreap(T)
  return toLTreapHelper(T,null)
```

Figure 2: Pseudocode for toLTreap.

```
fromLTreapHelper(T, L)
  if L = null then
    return T
  else
    T′ ← new Treap(T, L.key, L.child)
    return fromLTreapHelper (T′, L.parent)

fromLTreap (L)
  return fromLTreapHelper(null,L)
```

Figure 3: Pseudocode for fromLTreap.

```
splitLeftHelper (T, v, L):
  T′ ← new Treap(T, L.key, L.child)
  if L.parent ≠ null and k ≥ L.parent.key then
    return splitLeftHelper (T′, v, L.parent)
  else
    (T_l, m, T_r) ← split(T′, v)
    return (T_l, m, toLTreapHelper (T_r, L.parent))

splitLeft(v,L)
  if L = null then
    return (null, null, null)
  else
    return splitLeftHelper (null, v, L)
```

Figure 4: Pseudocode for splitLeft.

```
joinLeftHelper(T, T2, L)
  if L ≠ null and T.priority > L.priority then
    T2' ← new Treap(T2, L.key, L.child)
    return joinLeftHelper(T, T2', L.parent)
  else
    T' ← join(T, T2)
    return toLTreapHelper(T', L)

joinLeft(T, L)
  if T = null then
    return L
  else
    return joinLeftHelper(T, null, L)
```

Figure 5: Pseudocode for joinLeft.

We need to analyze the work required for these algorithms. toLTreap touches only the nodes along the left spine of the treap; the left spine of a treap contains expected $O(\lg(n))$ nodes, so the expected work required for toLTreap is $O(\lg(n))$.

Seidel and Aragon [2] showed that, if nodes a and b are separated in a treap by d nodes, then the distance from a to b in that treap is expected to be $O(\lg(d))$. Our splitLeft algorithm touches only the nodes on the path from the leftmost node in the LTreap to the predecessor of v in that LTreap. If there are d nodes which are less than v in the LTreap, then that path has length $O(\lg(d))$, so the expected work required for splitLeft is $O(\lg(d))$.

Using joinLeft to join a Treap and an LTreap requires the same amount of work as is needed to split them apart again using splitLeft. Thus the expected work needed for joinLeft is $O(\lg(d))$ where d is the number of nodes to the left of the split - that is, the number of nodes in the Treap.

We also wish to define a "Right Spinal Treap" (RTreap), which is just like an LTreap except that the pointers are flipped along the right spine rather than the left.

4.1 A Serial Union-Find Algorithm

As an example of how we can use the LTreap we present a serial union-finding algorithm:

```
toLRTreap(T)
  if T = null then
    return new LRTreap(null, null, null)
  L ← toLTreap(T.left)
  R ← toRTreap(T.right)
  return new LRTreap(L, T.key, R)
```

Figure 6: Pseudocode for toLRTreap .

```
serialUnion(L1, L2)
  if L1 = null then
    return L2
  if L2 = null then
    return L1
  (T, v, L1') ← splitLeft(L1, L2.key)
  L ← serialUnion(L2, L1')
  return joinLeft(T, L)
```

Using split and join on a given block costs at most $O(\lg(s))$ expected work where s is the size of that block. The logarithm is convex, so in the worst case each block is the same size and the work is $O(k \lg(\frac{n}{k} + 1))$ where $k = Block(L_1, L_2)$, which is optimal.

Note that we never use the v return value from a split. If a split returns a v which is non-null, then that v was a duplicate key and we can safely discard it.

Unfortunately, this algorithm cannot be run in parallel. In order to create a parallelizable algorithm we need to implement some form of divide-and-conquer.

5 The LRTreap

We have defined data structures which permit finger searches from either the left or the right sides. We will now create a data structure which combines the two types, allowing finger searches to be done from either side. This data structure, called an LRTreap, is represented by a triple (L, root, R) where L is an LTreap containing the keys to the left of the root, and R is an RTreap containing the keys to the right. We can convert an LRTreap to a standard treap or vice versa in $O(\lg(n))$ expected time just by traversing the spines. (See Figure 6.) In our pseudocode we refer to LRTreaps using the variable Υ.

We define two operations, splitLR and joinLR, which can be used on LRTreaps. splitLR is a function which splits an LRTreap on a given key, producing two LRTreaps; it works by testing the key against the root and then calling either splitLeft or splitRight as appropriate. joinLR does the opposite: it joins two LRTreaps into one by converting one of them to a standard treap and calling either joinLeft or joinRight. Pseudocode for these operations can be found in Figures 7 and 8.

```
splitLR(Υ, v)
  if Υ.root = null then
    return (Υ, null, Υ)
  else if Υ.root > k then
    (T, m, L') ← splitLeft(Υ.L,v)
    Υ_1 ← toLRTreap(T)
    Υ_2 ← new LRTreap(L',Υ.root,Υ.R)
    return (Υ_1,m,Υ_2)
  else if Υ.root == v then
    T_1 ← fromLTreap(Υ.L)
    T_2 ← fromRTreap(Υ.R)
    return (toLRTreap(T_1), Υ.root, toLRTreap(T_2))
  else \\ Υ.root < v
    (R', m, T) ← splitRight(Υ.R,v)
    Υ_1 ← new LRTreap(Υ.L,Υ.root,R')
    Υ_2 ← toLRTreap(T)
    return (Υ_1,m,Υ_2)
```

Figure 7: Pseudocode for splitLR.

```
joinLR(Υ_1, Υ_2)
  if isEmpty(Υ_1) then
    return Υ_2
  else if isEmpty(Υ_2) then
    return Υ_1
  else if Υ_1.root.priority > Υ_2.root.priority then
    T_2 ← fromLRTreap(Υ_2)
    R' ← joinRight(Υ_1.R, T_2)
    Υ ← new LRTreap(Υ_1.L,Υ_1.root, R')
    return Υ
  else
    T_1 ← fromLRTreap(Υ_1)
    L' ← joinLeft(Υ_2.L, T_1)
    Υ ← new LRTreap(L', Υ_2.root, Υ_2.R)
    return Υ
```

Figure 8: Pseudocode for joinLR.

Henceforth, when discussing a specific key v, we will denote the number of keys less than v by a and the number of keys greater than v by b. We wish to show that the SPLITLR operation requires $O(\lg(\min(a, b)))$ expected work.

We would like to directly use the result of Seidel and Aragon [2] which states that, if nodes v and v' are separated in a treap by d nodes, then the expected distance from v to v' is $O(\lg(d))$. Unfortunately, we are hindered in this approach by the fact that we cannot search along any path that includes the root. The bound still holds, however:

Theorem 5.1 *The expected work required to split an LRTreap on a key v is $O(\lg(\min(a, b)))$.*

Proof. There are three possibilities, depending on where in the treap the root is.

First, the root could be to the left of v. (This corresponds to the case on Line 5 in the SPLITLR code.) This occurs with probability (a/n), and the expected work required to find v is $O(\lg(b))$.

Similarly, the root could be to the right of v. (This is Line 14.) This occurs with probability (b/n), and the expected work required to find v is $O(\lg(a))$.

Lastly, v could be the root. (This is Line 10.) In this case, we have to convert the whole LRTreap into a standard treap, perform the operation, and convert it back, which takes $O(\lg(a + b))$. Fortunately, this only happens with probability $(1/n)$.

So the total expected work to find v is

$$O\Big(\frac{a \lg(b) + b \lg(a) + \lg(a + b)}{n}\Big)$$

But when $a < b$, $a \lg(b) < b \lg(a)$ and $\lg(a + b) < b$, so this is equivalent to $O(b \lg(a)/n)$, which is $O(\lg(a))$. Likewise, when $b < a$, this is equivalent to $O(\lg(b))$. Thus the total expected work is $O(\lg(\min(a, b)))$. ∎

We can use the SPLITLR function to search for a key as well, so these work bounds also apply for searching.

Likewise, joining two LRTreaps Υ_1 and Υ_2 (where all keys in Υ_1 are less than all keys in Υ_2) requires touching no more nodes than would be required to split them apart after they were joined. Thus this bound applies to the join operation as well.

We note that the code as described makes two passes along the spines of the smaller LRTreap. A more practical implementation could gain a constant factor of efficiency by merging those two passes into one.

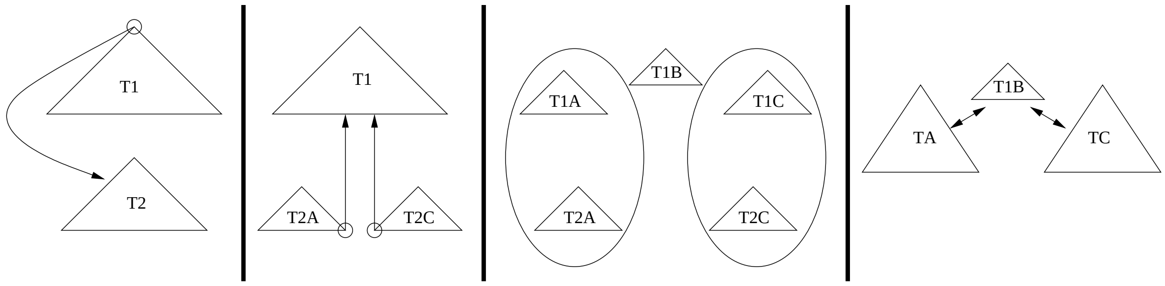

Figure 9: The four steps of our algorithm: split Υ_2, split Υ_1, recurse, rejoin.

```
GREATESTVALUE(Υ)
  if ISEMPTY(Υ) then
    return NEGATIVE_INFINITY
  if Υ.R ≠ null then
    return Υ.R.key
  else
    return Υ.root
```

Figure 10: Pseudocode for GREATESTVALUE. The LEASTVALUE code is similar.

6 Union-Find

We now turn our attention to the problem of finding the union of the elements of two treaps. Given two sorted sets of keys represented by LRTreaps, we wish to find an LRTreap containing the union of the two sets.

Our algorithm is shown in Figure 9, and can be described in four steps. Given two LRTreaps Υ_1 and Υ_2, with Υ_1.root.priority $\geq \Upsilon_2$.root.priority, we first split Υ_2 on the root of Υ_1. We then split Υ_1 on the two keys of Υ_2 that were nearest the previous split. This separates from Υ_1 a maximal block containing the root. In the third step, we recursively find the union of the pieces of the two LRTreaps; and in the fourth step, we join the results back together.

Pseudocode for our algorithm is shown in Figure 11. Note that, should Υ_{2A} be empty (that is, if the first split made is trivial), then GREATESTVALUE(Υ_{2A}) will return NEGATIVE_INFINITY (see Figure 10). Thus Υ_{1A} will also be empty, and UNION(Υ_{1A},Υ_{2A}) will immediately return an empty LRTreap. A similar case applies if Υ_{2C} is empty - so the possibility of trivial splits is handled without the necessity of checking for them in the code.

Note also that we never use the v, v', and v'' values returned by our calls to SPLITLR. If one of those values is non-null, then it is a duplicate key and can be safely discarded.

UNION(Υ_1, Υ_2)
if ISEMPTY(Υ_1) **then**
   return Υ_2
if ISEMPTY(Υ_2) **then**
   return Υ_1
if Υ_2.root.priority $>$ Υ_1.root.priority **then**
   $(\Upsilon_1, \Upsilon_2) \leftarrow (\Upsilon_2, \Upsilon_1)$
$(\Upsilon_{2A}, v, \Upsilon_{2C}) \leftarrow$ SPLITLR(Υ_2, Υ_1.root)
$(\Upsilon_{1A}, v', \Upsilon_1') \leftarrow$ SPLITLR(Υ_1, GREATESTVALUE(Υ_{2A}))
$(\Upsilon_{1B}, v'', \Upsilon_{1C}) \leftarrow$ SPLITLR(Υ_1', LEASTVALUE(Υ_{2C}))
$\Upsilon_A \leftarrow$ UNION(Υ_{1A}, Υ_{2A})
$\Upsilon_C \leftarrow$ UNION(Υ_{1C}, Υ_{2C})
return JOINLR(Υ_A, JOINLR(Υ_{1B}, Υ_C))

Figure 11: Pseudocode for UNION.

7 Analysis

In this section, all expectations are calculated over the random assignment of priorities to keys in Υ.

An important property of our algorithm is that, at any recursive step, we decide which splits to make based only on the location of the highest-priority root among the two LRTreaps. We then use splits to remove that root from future recursive steps. This ensures that we do not introduce bias: at any recursive step, every key is equally likely to be the root.

We formalize this property with a definition:

Definition 3 *An "almost-unbiased" split of an LRTreap is one which is chosen with no knowledge about that LRTreap other than its root node.*

Almost-unbiased splits have an interesting property that we will make use of in our analysis:

Lemma 7.1 *In an LRTreap Υ of size n, let N be the number of nodes visited by any almost-unbiased split. If that split visits N_{ns} non-spinal nodes and N_s spinal nodes, then $E[N] \leq 2E[N_{ns}] + 1$.*

Proof. Our SPLITLEFT and SPLITRIGHT algorithms visit nodes of two categories: spinal nodes and non-spinal nodes. They first travel up the spine of Υ until they reach a place to begin the split; then, they enter the interior of Υ and splits the two LRTreaps apart. Thus we have $N = N_s + N_{ns}$.

Consider any LRTreap Υ' that has been split off from Υ with an almost-unbiased split. The split was chosen without regard for the priorities of the keys in Υ', so all of those priorities

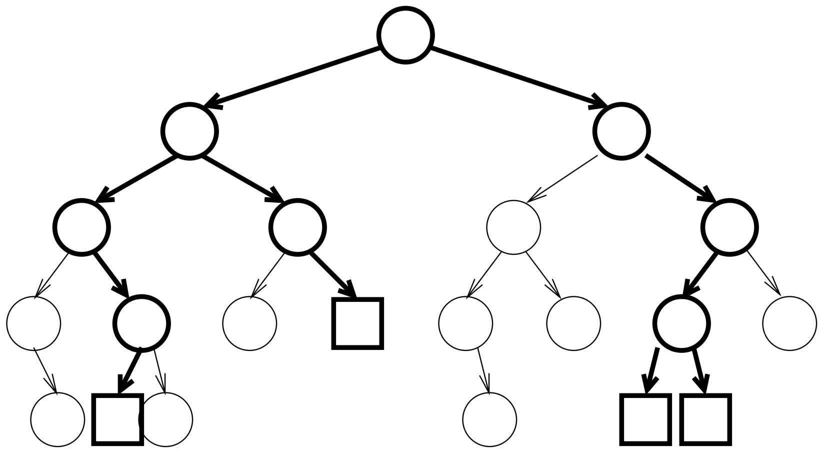

Figure 12: The path (bold) connecting any k leaves (squares) of a treap to each other and to the root contains expected $O(k \lg(\frac{n}{k}+1))$ nodes.

now are drawn from the same distribution. Thus Υ' is unbiased, so each of its spines has the same expected length, independent of the root of Υ. We know that one of those spines contains exactly N_s nodes; the other spine contains at most $N_{ns}+1$ nodes. (Some of the non-spinal nodes involved in the split may have stayed in Υ. The root of Υ', however, counts as a spinal node but still contributes 1 to the length of each spine of Υ'.) Thus we have $E[N_s] \le E[N_{ns}]+1$. We know $E[N] = E[N_s] + E[N_{ns}]$, so we have $E[N] \le 2E[N_{ns}]+1$. ∎

We can now begin to analyze the work required by our algorithm. To do this we will need the following lemma:

Lemma 7.2 *Given any unbiased treap T, and given k leaves of that treap, there is a unique minimal set of nodes P (called a "path") which connects those k leaves to each other and to the root of T, and $E[\,|P|\,]$ is $O(k \lg(\frac{n}{k}+1))$.*

Proof. The path is unique because the treap is acyclic. To show the size of the path, we consider an algorithm which traverses exactly that path.

Suppose that we take each of the $k-1$ leaves that the path touches and finger-search for them in order starting from the root. For each pair of adjacent leaves the finger search will travel up the tree until it finds the common ancestor of the two, then travel back down to the target leaf. Seidel and Aragon [2] showed that such a finger search takes $O(\lg(d))$ expected time per search, where d is the number of keys separating two endpoints. The logarithm is convex, so in the worst case all the spaces between the endpoints will be equal, and the finger-search algorithm will take expected time $O(\lg(n))$ for the search from the root to the first endpoint and $O(\lg(\frac{n}{k}+1))$ for each other search, for total expected time $O(k \lg(\frac{n}{k}+1))$.

The finger-searches described above touch exactly the nodes which are on the path we describe, so there can be at most $O(k \lg(\frac{n}{k}+1))$ of those nodes. ∎

We can now state the following theorem:

Theorem 7.3 *The expected work required for the* UNION *algorithm is $O(k \lg(\frac{n}{k}+1))$, where $k = Block(\Upsilon_1, \Upsilon_2)$ and $n = |\Upsilon_1| + |\Upsilon_2|$.*

Proof. We first consider Υ_1. We know that Υ_1 will eventually be split into $O(k)$ pieces by calls to SPLITLR. (A single call to SPLITLR might produce two splits of Υ_1, if the key being split on is a duplicate key. But both of those splits are counted in our *Block* metric, so this is still within our work bound.) We can trace the path P that those splits will take through Υ_1, and by Lemma 7.2 we know that $E[\,|P|\,]$ is $O(k \lg(\frac{n}{k}+1))$. Now observe that, regardless of the order in which we perform the splits, each node in the path will be used as a non-spinal node by exactly one split. (After a node is used as a non-spinal node, it becomes a spinal node of whatever LRTreap it ends up in.) If we denote the work required for split i by $N(i)$, then the total work required is:

$$W = \sum_{i=1}^{k} N(i)$$

$$\begin{aligned} E[W] &= E\left[\sum_{i=1}^{k} N(i)\right] && (1) \\ &\leq E\left[\sum_{i=1}^{k} (2N_{ns}(i)+1))\right] && (2) \\ &= 2E\left[\sum_{i=1}^{k} N_{ns}\right] + k \\ &= 2E[\,|P|\,] + k \\ &= O(k \lg(\frac{n}{k}+1)) \end{aligned}$$

where (2) follows from (1) and Lemma 7.1 and the fact that all the splits made by our algorithm are almost-unbiased.

An identical argument applies to Υ_2.

To see that the joins are within our work bound, consider the final product LRTreap Υ after all joins have been made, and imagine reversing the join-order to split it back up into its component blocks. We would first perform two splits to remove the root block from Υ. We would then recursively perform splits to remove the root blocks from the two pieces of Υ, and so on. The splits made in this fashion are determined only by the root of the component that is being split. Thus these splits are almost-unbiased splits, so we can again apply the argument above to see that the splits cost $O(k\lg(\frac{n}{k}+1))$ expected work. But joining two LRTreaps together takes the same amount of work as splitting them apart, so $E[W_{join}] = O(k\lg(\frac{n}{k}+1))$ as well.

We still need to show that the costs of the trivial splits and joins made are within our work bound. We define a "trivial" call to UNION to be one in which either Υ_1 or Υ_2 is empty. Observe that such calls immediately return without generating any further calls, so the total number of such calls can be no more than twice the number of nontrivial calls.

Now, each nontrivial call that is made results in at least one nontrivial join operation. Our algorithm makes at most $2k$ nontrivial joins in all, so there can be at most $2k$ nontrivial calls and $6k$ total calls. No call can contain more than a constant number of trivial splits and joins, so the work done on trivial splits and joins is $O(k)$.

Thus the total expected work required is $O(k\lg(\frac{n}{k}+1))$. ∎

7.1 Parallel Performance

We still need to consider the parallel performance of this algorithm. We can easily make our algorithm parallel by spawning separate threads for each recursive call. Since each call deals with a separate portion of the LRTreap, each thread will read and write from a separate part of the memory, so the algorithm will run on an EREW PRAM.

We would like to analyze the depth of our algorithm. Observe that, every time the algorithm recurses, the root is cut out of one of the LRTreaps. Thus the depth of the part of that LRTreap in the next recursion will be one less. That means that the sum of the depths of the LRTreaps decreases at each recursion. The expected depth of an LRTreap is $O(\lg(n))$, so the depth of the recursion is $O(\lg(n))$.

At each step of the recursion, the algorithm does at most $O(\lg(n))$ expected work, so the total depth of the algorithm is $O(\lg^2(n))$. To map this onto an EREW PRAM we need to deal with load-balancing. This can be done with a primitive scan operation or by including the depth of a scan in our time bound.

8 Intersection and Difference

With only slight modifications, we can also use our UNION algorithm to find the intersection or difference of two sets.

For INTERSECT, we want to include only duplicate keys in the final result, so we let Υ_{1B}

```
INTERSECT(Υ1, Υ2)
if ISEMPTY(Υ1) or ISEMPTY(Υ2) then
   return new LRTreap(null, null, null)
if Υ2.root.priority > Υ1.root.priority then
   (Υ1, Υ2) ← (Υ2, Υ1)
(Υ2A, v, Υ2C) ← SPLITLR(Υ2, Υ1.root)
(Υ1A, v′, Υ′1) ← SPLITLR(Υ1, GREATESTVALUE(Υ2A))
(Υ′1B, v″, Υ1C) ← SPLITLR(Υ′1, LEASTVALUE(Υ2C))
Υv ← new LRTreap(null, v, null)
Υv′ ← new LRTreap(null, v′, null)
Υv″ ← new LRTreap(null, v″, null)
Υ1B ← JOINLR(Υv, JOINLR(Υv′, Υv″)
ΥA ← INTERSECT(Υ1A, Υ2A)
ΥC ← INTERSECT(Υ1C, Υ2C)
return JOINLR(ΥA, JOINLR(Υ1B, ΥC))
```

Figure 13: Pseudocode for INTERSECT.

contain only the three duplicate keys v, v', and v''. We convert those keys to LRTreaps and then use JOINLR to form Υ_{1B} in order to handle the possibility that those keys might be null. Pseudocode for INTERSECT is shown in Figure 13.

For DIFFERENCE, the algorithm is a bit more complicated as we need to keep track of a boolean b that indicates which of Υ_1 and Υ_2 is the subtrahend. Also, if we encounter duplicate keys, we may need to explicitly remove the originals from whichever set they were in. To split off the leftmost key from Υ_{2C}, for example, we use the code $(\Upsilon_0,\ v'',\ \Upsilon_{2C}) \leftarrow$ SPLITLR(Υ_{2C}, v'') where we know from before that v'' was that leftmost key. Note that we never use the value Υ_0 as it will always be empty. Pseudocode for DIFFERENCE is shown in Figure 14.

9 Discussion

We have already mentioned the data structure of Kaplan and Tarjan [16] which allows splits and joins in worst-case $O(\lg(\min(a, b)))$ time. We used treaps to implement a similar data structure; for comparison, we will describe their structure here to provide an indication of its complexity.

We start by defining a 6-list to be a list that can contain no more than 6 elements. We can then recursively represent a sorted list L as follows. A list L contains three pieces: a prefix, a suffix, and a sub-list. The prefix and suffix are 6-lists; the sub-list $c(L)$ is a list whose elements are all either doubles or triples of elements from L. If L contains 6 or less elements then $c(L)$ is empty and all of the elements are stored in the prefix and suffix.

If we refer to the i-th level list with $c^i(L)$, then a list can be represented as a stack $S(L)$ in

```
\\ b is true if Υ_2 is the subtrahend, false otherwise.

DIFFERENCE(Υ_1, Υ_2, b)
if ISEMPTY(Υ_1) or ISEMPTY(Υ_2) then
  return b ? Υ_1 : Υ_2
if Υ_2.root.priority > Υ_1.root.priority then
  (Υ_1, Υ_2) ← (Υ_2, Υ_1)
  b ← not b
(Υ_2A, v, Υ_2C) ← SPLITLR(Υ_2, Υ_1.root)
(Υ_1A, v', Υ'_1) ← SPLITLR(Υ_1, GREATESTVALUE(Υ_2A))
(Υ'_1B, v'', Υ_1C) ← SPLITLR(Υ'_1, LEASTVALUE(Υ_2C))
if b then
  if v = null then
    Υ_1B ← Υ_1B'
  else
    (Υ_1BL, v, Υ_1BR) ← SPLITLR(Υ'_1B, v)
    Υ_1B ← JOINLR(Υ_1BL,Υ_1BR)
else
  Υ_1B ← new LRTreap(null, null, null)
  if v' ≠ null then
    (Υ_2A, v', Υ_0) ← SPLITLR(Υ_2A, v')
  if v'' ≠ null then
    (Υ_0, k'', Υ_2C) ← SPLITLR(Υ_2C, v'')
Υ_A ← DIFFERENCE(Υ_1A, Υ_2A, b)
Υ_C ← DIFFERENCE(Υ_1C, Υ_2C, b)
return JOINLR(Υ_A, JOINLR(Υ_1B, Υ_C))
```

Figure 14: Pseudocode for DIFFERENCE.

which the i-th element of $S(L)$ contains pointers to the prefix and suffix of $c^i(L)$.

An element of $c^i(L)$ can be represented as a 2-3 tree of depth i. For ease of searching, the leftmost key of such a 2-3 tree is stored with the tree itself.

A prefix or suffix is defined to be *green* if it contains two to four elements, *yellow* if it contains one or five elements, and *red* if it contains zero or six elements. The color of a list is defined to be the "worst" of the colors of its prefix and suffix, where red is defined to be "worse" than yellow, which is "worse" than green.

Kaplan and Tarjan maintain the following invariant over every list L:

"If $c^i(L)$ is a red list then $i > 0$ and there exists a green list $c^j(L)$, $j < i$, such that any list $c^k(L)$, $j < k < l$, is yellow."

In a functional setting, a list L is not actually represented by $S(L)$ but by a different stack $S'(L)$ in which every maximal sequence of yellow lists is replaced by a single element which contains a pointer to the sequence. The sequence is stored as a stack with the lowest-order

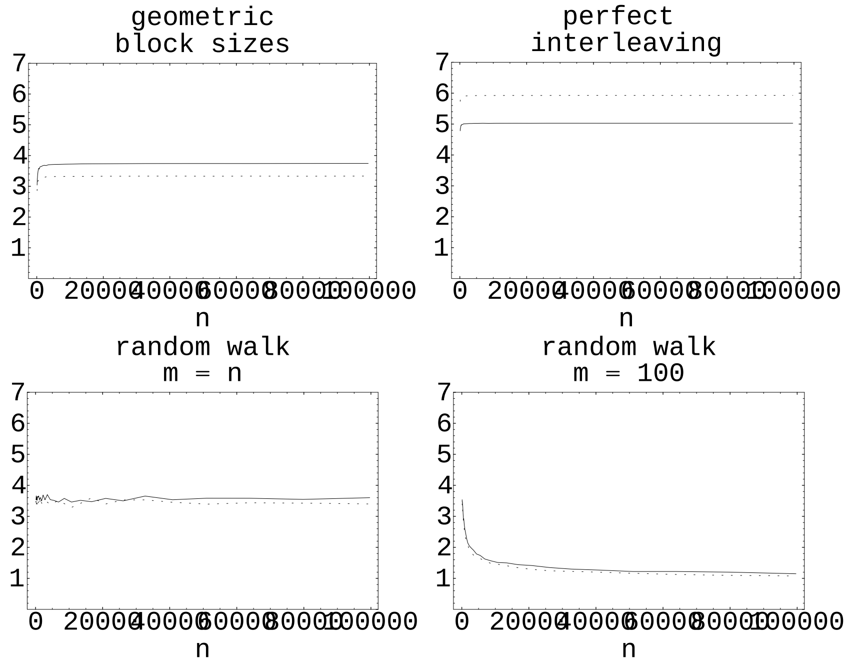

Figure 15: The performance constants of our algorithm using 2-3 trees (solid line) and treaps (dashed line).

list on top. This allows constant-time access to the topmost non-yellow list in any stack.

The structure described above permits implementations of SPLIT and JOIN which take $O(\lg(\min(a, b)))$ time, where a and b are the sizes of the pieces involved. With minor modifications we can allow the structure to keep track of a key which is near the midpoint of the list. SPLIT, JOIN, and GETMIDPOINT are the only operations that are required for our UNION, INTERSECT, and DIFFERENCE algorithms, so we can use the structure described above as a substitute for our treap data structures.

10 Analysis

We have implemented our algorithms in Java using both our Treap-based structures and Kaplan and Tarjan's 2-3-tree-based structures. (Our implementation of the 2-3-tree-based structures was only partial: it did not compress maximum sequences of yellow lists into one stack entry as described above. This affects the time bound of the structure but does not alter the number of comparisons it performs.) We conducted tests of our algorithms under varying conditions. We used four different distributions of random data. Our tests are shown in Figure 15.

For the first method the block sizes were geometrically distributed with mean 10. Blocks were created and assigned to sets until the total size of the two sets reached $2n$.

For the second method the keys were perfectly interleaved - that is, each block had size 1. This represents worst-case performance for our algorithm, since each set had to be completely broken down into keys before being reformed. The x-axis shows the number of keys in each set.

For the third method we generated the data through random walks. Datapoints were generated as a list in which the difference between any datapoint and its successor was a random variable uniformly distributed on the range $[-.5, .5)$. The lists were then sorted and converted to trees. The x-axis shows the number of keys in each set.

For the fourth method we also generated the data through random walks, but one of the sets was fixed to a size of 100 keys. Our performance factor improved with n in this case because the average size of each block increased with n. The x-axis shows the number of keys in the set whose size was not fixed.

In all tests we performed 100 trials at each datapoint. For each trial, we counted the number of comparisons made during the merge and divided by $k \lg(\frac{n+m}{k} + 1)$ to find the constant performance factor involved in our algorithm. The total included comparisons to the POSITIVE_INFINITY and NEGATIVE_INFINITY values returned when invoking GREATESTVALUE or LEASTVALUE on an empty structure; these values were independent of the underlying implementation, so this did not bias our results.

11 Future Work

One interesting avenue for future work would be to improve the parallel depth of this algorithm. Suppose we let n, m be the sizes of the two inputs, with $n \geq m$. At present our algorithm always chooses the higher-priority root among the two treaps for the first split. If instead we let our algorithm choose the lower-priority root, then the recursive call depth drops to $O(\lg(m))$, so the total depth should be $O(\lg(n)\lg(m))$. Unfortunately, this leads to the intermediate treaps becoming biased, which violates our work bound for this algorithm.

A possible solution to the above problem would be to switch the underlying balanced tree structures from treaps to 2-3 trees. The structures of Kaplan and Tarjan described in our discussion support the same split and join operations as our LRTreap structures, although the implementation is more complex. Using that implementation, bias would no longer be a concern, so we could make the change described above and decrease the depth to $O(\lg(n)\lg(m))$. Additionally, our bounds should become worst-case rather than expected-case.

References

[1] R. J. Anderson, E. W. Meyer, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82:262–277, 1989.

[2] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[3] A. Bäumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[4] G. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages ???–???, June 1998.

[5] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the Association for Computing Machinery*, 26(2):211–226, Apr. 1979.

[6] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9(3):594–614, Aug. 1980.

[7] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In *Proceedings of the International Symposium on Algorithms SIGAL'90*, pages 251–260, Tokyo, Japan, Aug. 1990.

[8] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. i. splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[9] E. Dekel and I. Azsvath. Parallel external merging. *Journal of Parallel and Distributed Computing*, 6:623–635, 1989.

[10] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, 40:592–602, 1991.

[11] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. of the 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[12] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[13] K. Hammond. Parallel functional programming: An introduction. In *First International Symposium on Parallel Symbolic Computation PASCO*, pages xiii+431, 181–93, Sept. 1994.

[14] L. Highan and E. Schenk. Maintaining B-trees on an EREW PRAM. *Journal of Parallel and Distributed Computing*, 22:329–335, 1994.

[15] H. Kaplan, C. Okasaki, and R. Tarjan. Simple confluently persistent catenable lists. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 119–130, July 1998.

[16] H. Kaplan and R. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 202–211, May 1996.

[17] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[18] R. E. Kaplan, H.; Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46:577–603, 1999.

[19] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. In *Proceedings of the 4th International PARLE Conference (Parallel Architectures and Languages Europe)*, volume 605 of *Lecture Notes in Computer Science*, pages 37–49, 1992.

[20] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Lecture Notes in Computer Science 143: Proceedings of the 10th Colloquium on Automata, Languages and Programming, Barcelona, Spain*, pages 597–609, Berlin/New York, July 1983. Springer-Verlag.

[21] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, University of Maryland, June 1990.

[22] A. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 127–137, San Diego, CA, June-July 1992.

[23] R. E. Tarjan and C. J. V. Wyck. An O(n log log n)-time algorithm for triangulating a simple polygon. *Siam J. Computing*, 17:143–173, 1988.

[24] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.