Functional Set Operations with Treaps

Dan Blandford and Guy Blelloch
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{blandford,blelloch}@cs.cmu.edu

August 8, 2001

Abstract

Purely functional algorithms have the advantages of leading to full persistence of all partial results, being easier to parallelize, and being easier to formally analyze. We present sequential and parallel functional algorithms for computing the union, intersection and difference of ordered sets using a variant of random balanced binary search trees (treaps). The algorithms are optimal with respect to the Block Metric bound of Carlsson et al.. In particular for two sets of length n and m, with $m \leq n$, the algorithms run in $O(k \lg(\frac{n}{k} + 1))$ expected 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 expected parallel depth of the parallel algorithms is $O(\log^2(n))$. The sequential bounds can also be achieved using the purely functional catenable list data structures suggested by Kaplan and Tarjan. However, we believe our are structure is significantly simpler—it only involves reversing the spines of a standard treap. We also present experimental results that analyze the constant factors involved in the algorithm.

A full implementation of our algorithms is available on the web at `http://www.cs.cmu.edu/~pscico/fingertrees.html` in both JAVA and SML.

1 Introduction

A purely functional algorithm is one that involves no side effects, *i.e.* no overwriting of existing data. Such algorithms have several advantages over imperative (side-effecting) algorithms, including the strongest form of persistence (data can never be overwritten), a more direct path to parallelism (independent threads cannot interact through memory writes), and a much better chance of formally or automatically reasoning about them. The purely functional model, however, is provably less powerful than imperative model [24], and it is not well understood what problems can be solved in the functional model with the same bounds as in the imperative model. To better understand the power of functional programs, there has been significant recent interest in developing efficient purely functional algorithms, both in the sequential [21, 18, 17, 22, 4] and in the parallel [6, 5] context. We note that although purely functional algorithms are typically discussed in the context of languages that are considered functional, such as ML or Haskell, these algorithms can also be programmed in languages that are considered imperative, such as JAVA.

In this paper we are interested in implementing set operations (union, intersection and difference) using purely functional algorithms. As motivation for this interest, consider the following application to search engines. Almost all search engines use an "inverted-lists" structure for storing the mapping of terms to documents [28]. This structure is simply a collection of "lists", one for each term in the database, and each list contains all the documents in which that term appears. Abstractly each list is better thought of as a set since the order has no importance. When a query is made involving and expression of AND and ORs on terms, this can be translated to intersections and unions on the corresponding sets (set difference can be used for ANDNOT queries). It is well known that these set operations can be implemented in time that is sublinear in the sum of the input sizes. Brown and Tarjan showed that merging two lists X and Y of size n and m (with $n \geq m$) takes $O(m \lg(\frac{n}{m} + 1))$ comparisons using AVL trees to represent the lists [7]. Assuming a total order on the set elements, it is not hard to extend the results to implement set union, intersection and difference in the same bounds. This sublinear time is particularly important when one list is much longer than the other, which happens frequently in practice (consider the query "red AND coeloglossum").

The imperative algorithm of Brown and Tarjan modifies its input—it inserts the elements from one list into the other. This is clearly undesirable in the suggested application since it will destroy the original term lists, which presumably need to be reused. Copying the inputs before the operation is not efficient since the set operations take sublinear time. If the set operations can be implemented purely functionally, however, the result set will share data with the input without modifying the input (the operations are fully persistent). This would not only save the original set for each term, but it would allow caching of partial results. Figure 1 shows an example of such caching.

For general inputs, the $O(m \lg(\frac{n}{m} + 1))$ bound is a lower bound for the list merging problem.

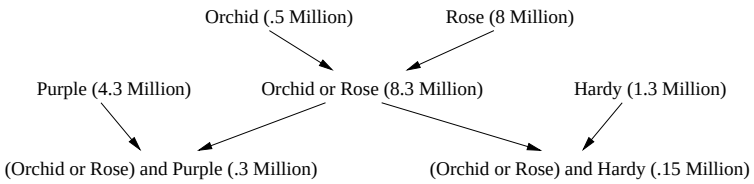

Figure 1: An example set of searches that reuse the cached result "orchid or rose". The numbers are the approximate size of each set based on the AltaVista search engine.

If we make some assumptions about the inputs, however, the bounds can be improved. Carlsson, Levcopoulos and Petersson [9] considered a block metric $k = Block(X, Y)$ which represents the minimum number of blocks that the lists need to be broken into before being recombined into one list. Given this metric they present an algorithm for static merging with bounds $O(k \lg(\frac{n}{k}+1))$. The static merging problem only generates cross pointers that show where the blocks from one list can be inserted into the other, but does not actually merge the results. It is not hard to show, however, that given split and join operations that run in $O(\log(\min(|S_1|, |S_2|)))$ time, regular merging and all the set operations can be implemented in $O(k \lg(\frac{n}{k} + 1))$ time [20]. Demaine, López-Ortiz and Munro extended this to methods for taking the union, intersection and difference on multiple sets. These methods do not parallelize easily since they involve sequentially pulling elements off of each input from one end.

The $O(k \lg(\frac{n}{k} + 1))$ bound can be met in the purely functional context with a functional representation of a sorted list that supports split and join in $O(\log(\min(|T_1|, |T_2|)))$ time. Kaplan and Tarjan [18, 17] developed such a structure based on 2-3 trees. The structure is described in Section B, in the Appendix. They point out that the structure can be used to implement the list-merging algorithm of Brown and Tarjan [8] with an optimal $O(m \log(\frac{n}{m}+1))$ time bound. However, the data structure they use is somewhat complex, the Brown and Tarjan list-merging algorithm does not give the stronger $O(k \lg(\frac{n}{k} + 1))$ time bounds, and the extension based on split and join mentioned in the previous paragraph does not parallelize.

In this paper we develop very simple functional data structures for representing sets based on Treaps [2]. For the sequential set operations presented in Section 3 we require only the trivial technique of flipping the pointers along the left spine of a Treap. Our parallel set operations 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 $O(\log^2 n)$ expected depth. We present all the code for all our algorithms—the only exception is for some symmetric routines for left and right spines in which we only show the left version.[1] We also present some experimental results that demonstrate that the constants in the work bounds are reasonably small. The experiments show that our algorithm does slightly better than the Kaplan and Tarjan algorithm in terms of number of comparisons.

Related Work. All the techniques we know of for implementing fast set operations originate from work on finger searching [14]. The idea of finger searching is that, given a pointer (or "finger") to any key in a search structure, it should be possible to find nearby keys in faster than the usual $O(\log(n))$ time. Many types of balanced trees permit finger searches using parent pointers. Unfortunately, parent pointers imply pointer cycles, which cannot be constructed using purely functional programming. The solution we use for this problem is to flip the pointers along one or 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 [26] describe this data structure as a "heterogeneous finger search tree".

Cole et al. [10] 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. Amortized bounds are weak for data structures that may have multiple futures since an application could perform many operations on the same version of a structure (consider the two operations on "Orchid or Rose" in Figure 1). If that version of the structure had a high potential value, the amortized bounds might not hold.

[1] We have also left the parallel intersection and difference code from this submitted version.

Demaine, López-Ortiz, and Munro [12] considered information-theoretic lower-bounds on the set operations which are stronger than the Block Metric bounds in that they take the sizes of the individual blocks into account. These bounds were shown earlier in the context of binary set operations [9], but they use some nice analysis tools to extend the bounds to operations on an arbitrary number of sets (*i.e.*, taking the union of m sets, or taking the intersection of m sets). They also proposed a sequential algorithm based on persistent B-trees which is optimal with respect to this bound. In Section 3 we show how to implement a version of their multi-set union operation using our sequential data structures. We believe it is significantly simpler than their original algorithm and also believe it is optimal with respect to their stronger bound.

In the parallel setting, previous work has focused either on merging algorithms that take $O(n)$ work [1, 11, 15, 27, 13, 19] and are optimal when the two sets have nearly equal sizes, or on multi-insertion algorithms that take $O(m \log(n+1))$ work and are optimal when the input values to be inserted are not presorted [23, 16, 25, 3]. More recently, Blelloch and Reid-Miller [5] demonstrated a non-functional parallel merging algorithm which operates in $O(k \log(\frac{n}{k}+1))$ work and $O(\log(m)\log(n))$ depth using parent pointers. A functional version of this algorithm requires $O(m \log(\frac{n}{m}+1))$ work, though the depth does improve to $O(\log(n))$.

2 Definitions

The Block Metric. Carlsson, Levcopoulos, and Petersson [9] 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 a lower bound for the list merging problem is $O(k \lg(\frac{n}{k}+1))$, where $k = Block(X,Y)$. We will present sequential set-operation algorithms that meet that bound in Section 3 and parallel algorithms that meets that bound in Section 5 and Appendix 7. First, though, we must develop the appropriate data structures.

Treaps. A treap is a balanced tree in which each key has an associated random priority [2]. 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, independent 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

```
JOIN(T1, T2)
  if T1 = null then
    return T2
  if T2 = null then
    return T1
  if T1.priority > T2.priority then
    T ← JOIN(T1.right, T2)
    return new Treap(T1.left, T1.key, T)
  else
    T ← JOIN(T1, T2.left)
    return new Treap(T, T2.key, T2.right)
```

```
SPLIT(T, v)
  if T = null then
    return (null, null, null)
  if v = T.key then
    return (T.left, T.key, T.right)
  if v < T.key then
    (T1, m, T2) ← SPLIT(T.left, v)
    T2' ← new Treap(T2, T.key, T.right)
    return (T1, m, T2')
  else \\ v > T.key
    (T1, m, T2) ← SPLIT(T.right, v)
    T1' ← new Treap(T.left, T.key, T1)
    return (T1', m, T2)
```

Figure 2: Pseudocode for JOIN and SPLIT.

containing the keys in $T_1 \cup T_2$. Both of these operations run in $O(\log(n))$ expected work. Code for these operations is shown in Figure 2.

Seidel and Aragon showed two properties of treaps which we will make use of in this paper. First, they showed that the expected depth of a treap of size n is $O(\log(n))$. Second, they showed the Finger Search property: if two keys in a treap are separated by d other keys, then the expected length of the path between them is $O(\log(d))$. These properties will be useful in proving bounds on our algorithms.

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.*

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. This will prove important in the analysis of our parallel union algorithm in Section 6.

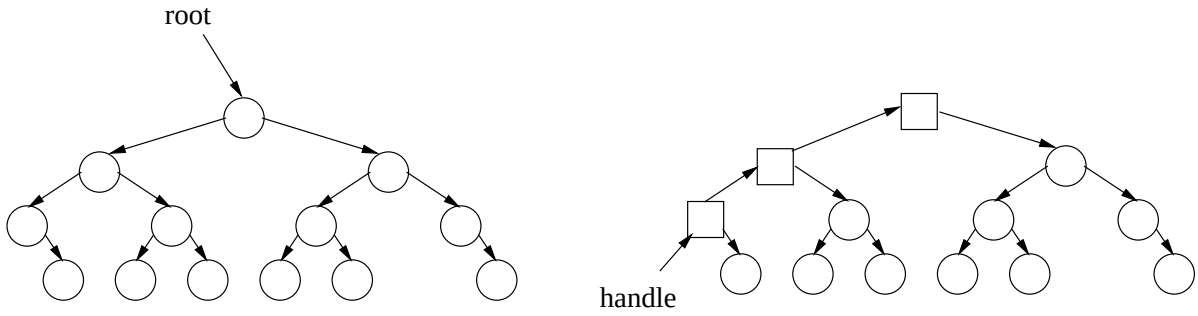

Figure 3: An LTreap is a Treap in which the pointers along the left spine are flipped.

3 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 3.) We maintain a handle into the structure at the leftmost node, and all searches in the structure start from that node.

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 Figure 4.

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.

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

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 $O(\log(d))$, so the expected work required for splitLeft is $O(\log(d))$.

Using joinLeft requires the same amount of work as is needed by splitLeft to split the result into back into its components. Thus the expected work needed for joinLeft is $O(\log(d))$ where d is the number of nodes to the left of the split—that is, the number of nodes in the Treap.

We can also 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.

Sequential Set Operations. As an example of how we can use the LTreap we present sequential algorithms which perform the set operations (See Figure 5). Each of our algorithms uses one split and one join on each block in the input. Using split and join on a given block costs at most $O(\log(s))$ expected work where s is the size of that block. Thus the total work is $O(\sum_{i=1}^{k} \log(s_i))$. 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.

```
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)
```

```
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)
```

```
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)
```

```
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
    (Tl, m, Tr) ← split(T', v)
    L' ← toLTreapHelper (Tr, L.parent)
    return (Tl, m, L')

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

Figure 4: Pseudocode for toLTreap, fromLTreap, joinLeft and splitLeft.

We can also use our LTreap data structures to implement the algorithm of Demaine et al. [12] for finding the union of multiple sets. This simplifies the code somewhat—we can construct the data structure at the same time as we locate the blocks, rather than performing the two tasks in separate phases. Pseudocode is shown in Figure 11, in the Appendix.

Unfortunately, these algorithms do not parallelize. To create parallelizable algorithms we need a data structure that allows some form of divide-and-conquer.

4 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 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 $O(\log(n))$ expected time just by traversing the spines. 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

```
seqIntersect(L1, L2)
  if L1 = null then
    return null
  (T, v, L2') ← splitLeft(L1.key, L2)
  L ← seqIntersect(L2', L1)
  if v = null then
    return L
  else
    Tv ← new Treap(null, v, null)
    return joinLeft(Tv, L)

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

seqDiff1(L1, L2)
  if L1 = null then
    return null
  (T, v, L2') ← splitLeft(L1.key, L2)
  if v ≠ null then
    (T0, v', L1') ← splitLeft(L1.key, L1)
    return seqDiff2(L2', L1')
  else
    return seqDiff2(L2', L1)

seqDiff2(L1, L2)
  if L1 = null then
    return L2
  (T, v, L2') ← splitLeft(L1.key, L2)
  L ← seqDiff1(L2', L1)
  return joinLeft(T, L)
```

Figure 5: Simple set-operation algorithms. seqDiff1 subtracts L_2 from L_1; seqDiff2 subtracts L_1 from L_2.

in Figure 12, in the Appendix.

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 $O(\log(\min(a,b)))$ expected work.

We know that, if nodes v and v' are separated in a treap by d nodes, then the expected distance from v to v' is $O(\log(d))$. Unfortunately, our algorithm may not always be able to take the shorter path to the target key because it cannot search along any path that includes the root. The bound still holds, however:

Theorem 3
The expected work required to split an LRTreap on a key v is $O(\log(\min(a,b)))$.

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 occurs with probability (a/n), and the expected work required to find v is $O(\log(b))$.

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

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

So the total expected work to find v is

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

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

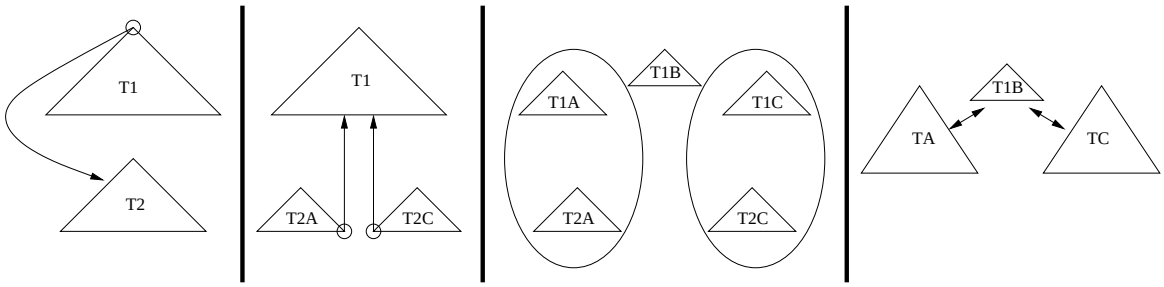

Figure 6: The four steps of our algorithm: split Υ_2, split Υ_1, recurse, rejoin.

work is $O(\log(\min(a, b)))$. ■

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.

Likewise, we can use SPLITLR and JOINLR to implement search, insert, and delete functions, so the work bounds for SPLITLR and JOINLR also apply to these functions.

We note that the code as shown in Figure 12 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.

5 Set Operations: Union

We now turn our attention to the problem of computing the union of the elements of two treaps in parallel. 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 illustrated in Figure 6, 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 compute 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 7. We use a $||$ symbol to mark places where a new thread can be spawned to deal with a recursive call. 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 7). 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.

6 Analysis

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

```
UNION(Υ1, Υ2)
  if Υ1.root = null then
    return Υ2
  if Υ2.root = null then
    return Υ1
  if Υ2.root.priority > Υ1.root.priority then
    (Υ1, Υ2) ← (Υ2, Υ1)
  (Υ2A, v, Υ2C) ← SPLITLR(Υ2, Υ1.root)
  (Υ1A, v', Υ1') ← SPLITLR(Υ1, GREATESTVALUE(Υ2A))
  (Υ1B, v'', Υ1C) ← SPLITLR(Υ1', LEASTVALUE(Υ2C))
  ΥA ← UNION(Υ1A, Υ2A) ||
  ΥC ← UNION(Υ1C, Υ2C)
  return JOINLR(ΥA, JOINLR(Υ1B, ΥC))
```

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

Figure 7: Pseudocode for UNION and for GREATESTVALUE . LEASTVALUE is similar to GREATESTVALUE.

In Section 2 we defined an "unbiased" treap to be one in which all of the priorities are identically and independently distributed. This is important, because our bounds on a treap's expected depth only apply when the treap is known to be unbiased. If we make decisions based on the priorities of the keys in a treap, we may bias the structure. For example, suppose we adopt the rule that we always split a treap immediately to the left of the highest-priority key in the structure. One of the result treaps would then be biased—the location of its root would no longer be random. Our parallel algorithm does make decisions based on the priorities of the keys, and so we need to show that we do not cause bias in the intermediate treaps.

How can we bound the bias introduced by our algorithm? 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 the following definition:

Definition 4
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 5
In an unbiased 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 split 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 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

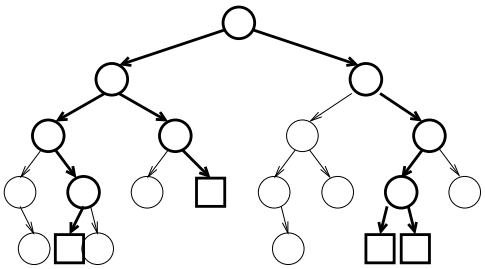

Figure 8: The path (bold) connecting any k leaves (squares) of a treap to each other and to the root contains expected $O(k \lg(\frac{n}{k}+1))$ nodes.

length of each spine of Υ'.) Thus we have $E[N_s] \leq E[N_{ns}] + 1$. We know $E[N] = E[N_s] + E[N_{ns}]$, so we have $E[N] \leq 2E[N_{ns}] + 1$. ■

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

Lemma 6
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. The first search takes $O(\log(n))$ expected work, and the rest take $O(\log(d))$ expected time per search, where d is the number of keys separating two endpoints. Thus the total work is $O(\log(n) + \sum_{i=1}^{k} d_i)$. 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 $O(k \lg(\frac{n}{k}+1))$.

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

We defer the proof to Appendix A.

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 $O(\log(n))$, so the depth of the recursion is $O(\log(n))$.

At each step of the recursion, the algorithm does at most $O(\log(n))$ expected work, so the total depth of the algorithm is $O(\log^2(n))$. 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.

7 Set Operations: Intersect and Difference

With only slight modifications, we can also use our UNION algorithm to find the intersection or difference of two sets. Code, description and analysis for these operations will be included in the final version of our paper.

8 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. This results in a parallel depth of $O(\log^2(n))$. If instead we let our algorithm choose the lower-priority root, then the recursive call depth drops to $O(\log(m))$, so the total depth drops to $O(\log(n)\log(m))$. Unfortunately, this leads to the intermediate treaps becoming biased, which violates our work bound for this algorithm. If we use Kaplan and Tarjan's structure, though, bias ceases to be a problem, and the modification we described above will work.

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, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. To appear in the Journal of Functional Programming, 2001.

[5] G. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 16–26, June 1998.

[6] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, Jan. 2000.

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

[14] 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.

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

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

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

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

[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] A. Moffat, O. Petersson, and N. C. Wormald. A tree-based mergesort. *Acta Informatica*, 35(9):775–793, 1998.

[21] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[22] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[23] 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.

[24] N. Pippenger. Pure versus impure lisp. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):223–238, 1997.

[25] 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.

[26] 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.

[27] 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.

[28] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images (Second Edition)*. Morgan Kaufmann Publishing, San Francisco, 1999.

A Proof of Theorem 7

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 6 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 5 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 $E[W_{joins}] = O(k \lg(\frac{n}{k}+1))$ 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 $O(k \lg(\frac{n}{k}+1))$. ■

B Discussion

We have already mentioned the data structure of Kaplan and Tarjan [17] which allows splits and joins in worst-case $O(\log(\min(a,b)))$ 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 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 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 operate in $O(\log(\min(a,b)))$ 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.

C Implementation

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 maximal 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, merging disjoint lists of sizes n and m. We used four different distributions of random data. Our tests are shown in Figure 9.

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 this distribution the value recorded for n is an average, not an exact value.

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.

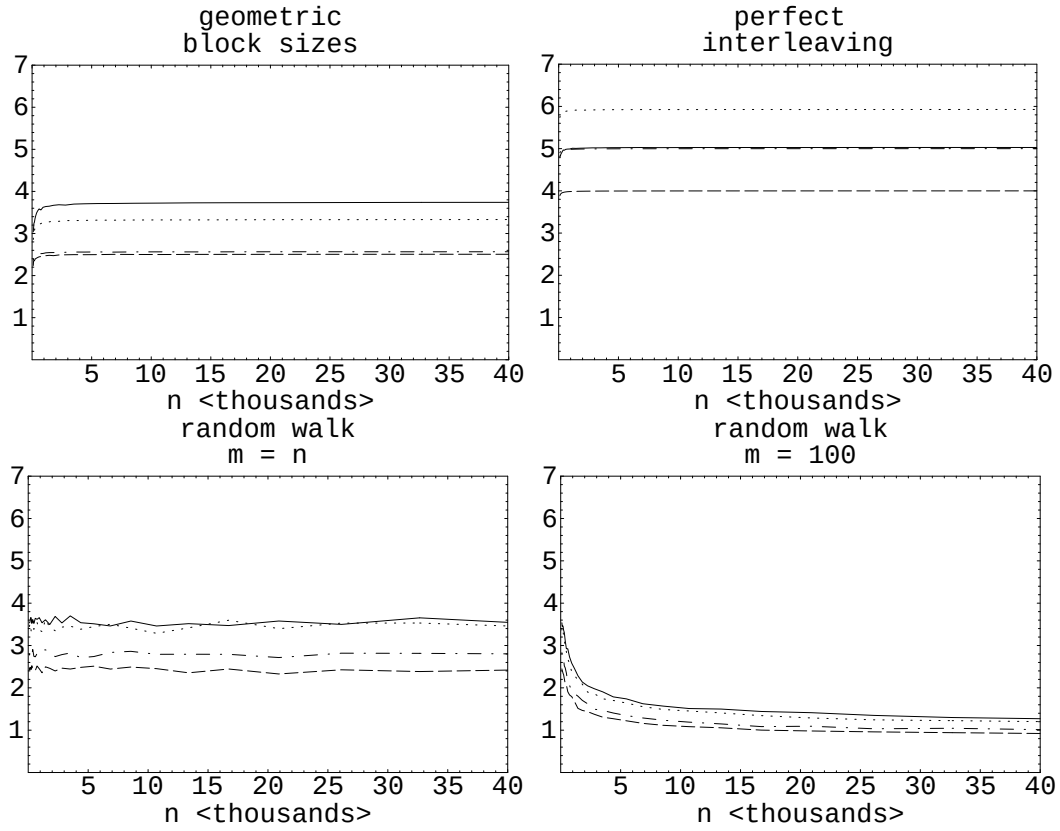

Figure 9: The performance constants of our algorithm using parallel 2-3 trees (solid line), parallel treaps (dotted line), sequential 2-3 trees (dot-dashed line), and sequential treaps (dashed line).

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 $k \log(\frac{n+m}{k}+1)$ 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.

Although the performance constants appear to depend on mean block size, they are independent of list size for sufficiently large n. Noting this, we created some simulations in which we fixed $n \simeq 10000$ and varied the block size. The results are shown in Figure 10.

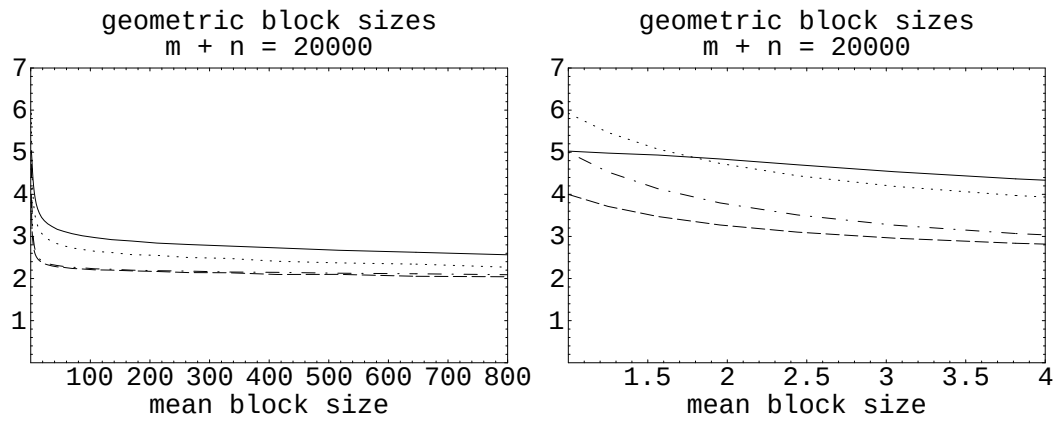

Figure 10: The performance constants of our algorithm using parallel 2-3 trees (solid line), parallel treaps (dotted line), sequential 2-3 trees (dot-dashed line), and sequential treaps (dashed line). The right plot is the same as the left, but with a different scale.

```
multiUnionHelper(L,Q)
  if isEmpty(Q) then
    return L
  (v, L') ← deleteMin(Q)
  (T, v, L) ← splitLeft(v, L)
  if L ≠ null then
    Q ← insert(Q,L.key,L)
  L ← multiUnionHelper(L', Q)
  return joinLeft(T, L)

multiUnion(Ls)
  Q = new Queue()
  foreach L in Ls
    Q ← insert(Q,L.key,L)
  return multiUnionHelper(null, Q)
```

Figure 11: A simple variant of the multiunion algorithm of Demaine et al. [12] using LTreaps. The insert(Q, k, d) routine places the data d into a priority queue Q based on the key k. We assume that multiple consecutive deleteMins of the same key take constant time after the first deleteMin.

```
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)
```

```
fromLRTreap(Υ)
  if Υ.root = null then
    return null
  T1 ← fromLTreap(Υ.L)
  T2 ← fromRTreap(Υ.R)
  return new Treap(T1, Υ.root, T2)
```

```
joinLR(Υ1, Υ2)
  if Υ1.root = null then
    return Υ2
  if Υ2.root = null then
    return Υ1
  if Υ1.root.priority > Υ2.root.priority then
    T2 ← fromLRTreap(Υ2)
    R' ← joinRight(Υ1.R, T2)
    Υ ← new LRTreap(Υ1.L,Υ1.root, R')
    return Υ
  else
    T1 ← fromLRTreap(Υ1)
    L' ← joinLeft(Υ2.L, T1)
    Υ ← new LRTreap(L', Υ2.root, Υ2.R)
    return Υ
```

```
splitLR(Υ, v)
  if Υ.root = null then
    return (Υ, null, Υ)
  if Υ.root == v then
    Υ1 ← toLRTreap(fromLTreap(Υ.L))
    Υ2 ← toLRTreap(fromRTreap(Υ.R))
    return (Υ1, Υ.root, Υ2)
  if Υ.root > v then
    (T, m, L') ← splitLeft(v, Υ.L)
    Υ1 ← toLRTreap(T)
    Υ2 ← new LRTreap(L',Υ.root,Υ.R)
    return (Υ1,m,Υ2)
  else \\ Υ.root < v
    (R', m, T) ← splitRight(v, Υ.R)
    Υ1 ← new LRTreap(Υ.L,Υ.root,R')
    Υ2 ← toLRTreap(T)
    return (Υ1,m,Υ2)
```

Figure 12: Pseudocode for toLRTreap, fromLRTreap, joinLR and splitLR.