COMPACT REPRESENTATIONS OF ORDERED SETS *

Daniel K. Blandford
dkb1@cs.cmu.edu

Guy E. Blelloch
blelloch@cs.cmu.edu

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We consider the problem of efficiently representing sets S of size n from an ordered universe $U = \{0, \ldots, m-1\}$. Given any ordered dictionary structure (or comparison-based ordered set structure) D that uses $O(n)$ pointers, we demonstrate a simple blocking technique that produces an ordered set structure supporting the same operations in the same time bounds but with $O(n \log \frac{m+n}{n})$ bits. This is within a constant factor of the information-theoretic lower bound. We assume the unit cost RAM model with word size $\Omega(\log |U|)$ and a table of size $O(m^\alpha \log^2 m)$ bits, for some constant $\alpha > 0$. The time bound for our operations contains a factor of $1/\alpha$.

We present experimental results for the STL (C++ Standard Template Library) implementation of Red-Black trees, and for an implementation of Treaps. We compare the implementations with blocking and without blocking. The blocking variants use a factor of between 1.5 and 10 less space depending on the density of the set.

1 Introduction

Memory considerations are a serious concern in the design of search engines. Some web search engines index over a billion documents, and even this is only a fraction of the total number of pages on the Internet. Most of the space used by a search engine is in the representation of an *inverted file* index, a data structure that maps search terms to lists of documents containing those terms. Each entry (or *posting list*) in an inverted file index is a list of the document numbers of documents containing a specific term. When a query on multiple terms is entered, the search engine retrieves the corresponding posting lists from memory, performs some set operations to combine them into a result, and reports them to the user. It may be desirable to maintain the documents ordered, for example, by a ranking of the pages based on importance [13]. Typically using difference coding these lists can be compressed into an array of bits using 5 or 6 bits per edge [17, 12, 3], but such representations are not well suited for merging lists of different sizes.

Here we are interested in a data structure to compactly represent an individual posting list, represented as an ordered set $S = \{s_1, s_2, \ldots, s_n\}, s_i < s_{i+1}$, from a universe $U = \{0, \ldots, m-1\}$. This data structure should support dynamic operations including set union and intersection, and it should operate in a purely functional setting [10] since it is desirable to reuse the original sets for multiple queries. In a purely functional setting data cannot be overwritten. This means that all data is fully persistent.

There has been significant research on succinct representation of sets taken from U. An information-theoretic bound shows that representing a set of size n (for $n \leq \frac{m}{2}$) requires $\Omega(\log \binom{m}{n}) = \Omega(n \log \frac{m+n}{n})$ bits. Brodnik and Munro [5] demonstrate a structure that is optimal in the high-order term of its space usage and supports lookup in $O(1)$ worst-case time and insert and delete in $O(1)$ expected amortized time. Pagh [14] simplifies the structure and improves the space bounds slightly. These structures, however, are based on hashing and do not support ordered access to the data: for example, they support searching for a precise key, but not searching for the next key greater (or less) than the search key. Pagh's structure does support `Rank` but only statically, *i.e.*, without allowing insertions and deletions. As with our work they assume the unit cost RAM model with word size $\Omega(\log |U|)$.

The set union and intersection problems are directly related to the list merging problem, which has received significant study. Carlsson, Levcopoulos, and Petersson [7] considered a block metric $k = Block(S_1, S_2)$

*This work was supported in part by the National Science Foundation as part of the Aladdin Center (`www.aladdin.cmu.edu`) and Sangria Project (`www.cs.cmu.edu/~sangria`) under grants ACI-0086093, CCR-0085982, and CCR-0122581.

which represents the minimum number of blocks that two ordered lists S_1, S_2 need to be broken into before being recombined into one ordered list. Using this metric they show an information-theoretic lower bound of $\Omega(k \log \frac{|S_1|+|S_2|}{k})$ on the time complexity of list merging in the comparison model.

Moffat, Petersson, and Wormald [11] show that the list merging problem can be solved in $O(k \log \frac{|S_1|+|S_2|}{k})$ time by any structure that supports *fast split and join* operations. A split operation is one that, given an ordered set S and a value v, splits the set into sets S_1 containing values less than v and S_2 containing values greater than v. A join operation is one that, given sets S_1 and S_2, with all values S_1 less then the least value in S_2, joins them into one set. These operations are said to be *fast* if they run in $O(\log(\min(|S_1|, |S_2|)))$ time. In fact, the actual algorithm requires only that the split and join operations run in $O(\log |S_1|)$ time.

Here we demonstrate a compression technique which improves the space efficiency of structures for ordered sets taken from U. We consider the following operations:

- `Search`$^-$ (`Search`$^+$): Given x, return the greatest (least) element of S that is less than or equal (greater than or equal) to x.
- `Insert`: Given x, return the set $S' = S \cup \{x\}$.
- `Delete`: Given x, return the set $S' = S \setminus \{x\}$.
- `FingerSearch`$^-$(`FingerSearch`$^+$): Given a handle (or "finger") for an element y in S, perform `Search`$^-$ (`Search`$^+$) for x in $O(\log d)$ time where $d = |\{s \in S \mid y < s < x \ \vee \ x < s < y\}|$.
- `First`, `Last`: Return the least (or greatest) element in S.
- `Split`: Given an element x, return two sets $S' : \{y \in S \mid y < x\}$ and $S'' : \{y \in S \mid y > x\}$, plus x if it was in S.
- `Join`: Given sets S', S'' such that $\forall x \in S', \forall y \in S'', x < y$, return $S = S' \cup S''$.
- `(Weighted)Rank`: Given an element y and weight function w on S, find $r = \sum_{x \in S, x<y} w(x)$. In the unweighted variant, all weights are considered to be 1.
- `(Weighted)Select`: Given r and a weight function w, find the greatest y such that $\sum_{x \in S, x<y} w(x) \leq r$. Return both y and the associated sum. In the unweighted variant, all weights are considered to be 1.

Given any ordered dictionary structure D supporting certain subsets of these operations and using $O(n \log m)$ bits to store n values from $U = \{0, \ldots, m-1\}$ we demonstrate a simple blocking technique that produces an ordered set structure using $O(n \log \frac{m+n}{n})$ bits. This is within a constant factor of the information-theoretic lower bound. Our technique requires that the target machine have a word size of $\Omega(\log m)$. This is reasonable since $\log m$ bits are required to distinguish the elements of U. Our technique also makes use of a lookup table of size $O(m^\alpha \log^2 m)$ for any $\alpha > 0$.

Our data structure works as follows. Elements in the structure are *difference coded* [17] and stored in fixed-length blocks of size $\Theta(\log m)$. The first element of every block is kept uncompressed. The blocks are kept in a dictionary structure (with the first element as the key). The data structure needs to know nothing about the actual implementation of the dictionary. A query consists of first searching for the appropriate block in the dictionary, and then searching within that block. We provide a framework for dynamically updating blocks as inserts and deletes are made to ensure that no block becomes too full or too empty. For example, inserting into a block might overflow the block. This requires it to be split and a new block to be inserted into the dictionary. The operations we use on blocks correspond almost directly to the operations on the tree as a whole. We use table-lookup to implement the block operations efficiently.

Our structure can support a wide range of operations, depending on the operations the dictionary D supports. In all cases the cost of our operations is $O(1)$ instructions and $O(1)$ operations on D.

If the input structure D supports the `Search`$^-$, `Search`$^+$, `Insert`, and `Delete` operations, then our structure supports those operations.

If D supports `FingerSearch` and supports `Insert` and `Delete` at a finger, then our structure supports those operations.

If D supports `First`, `Last`, `Split`, and `Join`, then our structure supports those operations. If the bounds for `Split` and `Join` are $O(\log \min(|D_1|, |D_2|))$, then our structure meets these bounds (despite the $O(1)$ calls to other operations).

If D supports `WeightedRank`, then our structure supports `Rank`. If D supports `WeightedSelect`, then our structure supports `Select`. Our algorithms need the weighted versions so that they can use the number of entries in a block as the weight.

The catenable-list structure of Kaplan and Tarjan [10] can be adapted to support all of these operations. The time bounds (all worst-case) are $O(\log n)$ for `Search`$^-$, `Search`$^+$, `Insert`, and `Delete`; $O(\log d)$

for `FingerSearch`, where d is as defined above; $O(1)$ for `First` and `Last`; and $O(\log \min(|D_1|, |D_2|))$ for `Split` and `Join`. Our structure meets the same bounds. As another example, our representation based on a simpler dictionary structure based on Treaps [16] supports all these operations in the time listed in the expected case. Both of these can be made purely functional. As a third example, our representation using a skip-list dictionary structure [15] supports these operations in the same time bounds (expected case) but is not purely functional.

In Section 5 we present experimental results for red-black trees and treaps. For red-black trees we consider insertions, deletions and searches. For treaps, in addition to insertion, deletions and searches we present results for a merge routine based on split and join operations.

2 Block structure

Our representation consists of two structures, nested using a form of structural bootstrapping [6]. The base structure is the *block*. In this section we describe our block structure and the operations supported on blocks. Then, in Section 3, we describe how blocks are kept in an ordered-dictionary structure to support efficient operations.

The block structure, the given dictionary structure and our combined structure all implement the same operations except that the block structure has an additional `BMidSplit` operation, and only the given dictionary supports the weighted versions of `Rank` and `Select`. For clarity, we refer to operations on blocks with the prefix B (*e.g.*, `BSplit`), operations on the given dictionary structure with the prefix D (*e.g.*, `DSplit`), and operations on our combined structure with no prefix.

Logarithmic codes. Our data structures are compressed using *logarithmic codes*. A logarithmic code is any variable-length prefix code that uses $O(\log v)$ bits to represent a value v. An example of a logarithmic code is the *gamma code* [8]. The gamma code represents a positive integer v with $\lfloor \log v \rfloor$ zeroes, followed by the $\lfloor \log v \rfloor + 1$-bit binary representation of v, for a total of $2\lfloor \log v \rfloor + 1$ bits. Gamma codes for multiple values can be concatenated to form a sequence of bits. Since the codes are prefix codes, this sequence can be uniquely decoded.

We use M to denote the maximum possible length of a difference code. In the case of gamma codes, $M = 2\lfloor \log m \rfloor + 1$ bits. Throughout Sections 2 and 3 we will assume the use of gamma codes. For the experiments we use a different logarithmic code.

{306, 309, 312, 314, 315, 319}

306	3	3	2	1	4

0100110010	011	011	010	1	00100

Figure 1: The encoding of a block of size 15. In this case the universe has size 1024, so the head is encoded with 10 bits.

Block encoding. A *block* B_i is an encoding of a series of values (in increasing order) $v_1, v_2, \ldots, v_k$. The block is encoded as a $\log m$-bit representation of v_1 (called the "head") followed by difference codes for $v_2 - v_1, v_3 - v_2, \ldots, v_k - v_{k-1}$. (See Figure 1 for an example.) We say that the *size* of a block `size`(B) is the total length of the difference codes contained in that block. In particular we are interested in blocks of size $O(\log m)$ bits.

It is important for our time bounds that the operations on blocks are fast—they cannot take time proportional to size of the block. Operations such as `BFingerSearch` have to take time proportional to d and a block can have up to $\log m$ entries.

Lemma 2.1 *If a block contains s values and has size b bits, then using a lookup table of size $O(m^{2\alpha} \log m)$ it is possible to search the block for a value v in $O(\frac{b}{\alpha \log m})$ time in the worst case. In particular, if b is $O(\alpha \log m)$ then it is possible to search in $O(\frac{1}{\alpha})$ worst-case time.*

Proof. To search a block of size b for a key k, our algorithm makes use of an auxiliary decoding table which maps *chunks* of $\alpha \log m$ bits of code to the result when those bits are decoded. The chunk may contain multiple difference codes but the first code must be aligned with the start of the chunk. Each entry in the lookup table contains two arrays.

Each entry i in the the first array contains two values: the greatest value encoded in the chunk that is less than i and the least value in the chunk that is greater than i. If i is in the chunk, the entry also lists the bit offset from the start of the chunk to the end of the difference code for i. Note that this array describes *values* rather than *differences*: if the chunk contains gamma codes for the sequence $3, 1, 4, 1$, then the array describes the decoded values $3, 4, 8, 9$. We assume the part of the block before the chunk has been decoded so we know the start value v for the chunk. In searching for a key k we therefore look in array position $i = k - v$ (if in bounds).

The second array contains one entry for each bit-position in the chunk. This entry gives the ending bit-position of the difference encoded there, if it is in the chunk. This is needed for the `BMidSplit` operation.

The first array in each entry contains space for $O(2^{\alpha \log m})$ values, each using $O(\log m)$ bits. The second array contains space for $\alpha \log m$ values, each using $O(\log m)$ bits. In all the table contains $2^{\alpha \log m}$ entries using $O(2^{\alpha \log m}) \log m$ bits each, for a total of $O(2^{2\alpha \log m} \log m) = O(m^{2\alpha} \log m)$ bits. By choosing an appropriate $\alpha < .5$ we can control the size of the table.

When decoding using table lookup it may occur that a chunk contains no full gamma codes—that is, that the first gamma code in the chunk is longer than the chunk size. In this case, our algorithm can decode the gamma code in constant time. The length of a gamma code is $2z+1$ where z is the number of zeroes preceding the first one; the value of the gamma code is simply the last $z+1$ bits of the code. Our algorithm finds the location of the first 1 using table lookup (the table size is $O(m^\alpha \log \log m)$), then reads the last $z+1$ bits using shift operations.

Consider one "decoding step" on a chunk to consist of a table-lookup operation followed (if necessary) by decoding an oversized gamma code as described above. At the end of a decoding step, the algorithm moves to the end of the final gamma code that was decoded and performs another step. Any two decoding steps must decode at least $\alpha \log m$ bits, so at most $O(\frac{b}{\alpha \log m})$ decoding steps are required in the worst case.

The search algorithm keeps track of the value of the last element in each chunk it decodes. It performs decoding steps until the last element of the current chunk is greater than the target value v. The algorithm then examines the array at the appropriate offset for v and returns the result. Each decoding step takes $O(1)$ time, so the total time used is $O(\frac{b}{\alpha \log m})$; if b is $O(\log m)$ then the total time used is $O(\frac{1}{\alpha})$. ■

In our application constant-time search will not be needed since dictionary operations will use $O(\log n)$ time for regular searches or $O(\log d)$ time for finger searches. For this bound the lookup table can be simplified: each lookup table entry can be an array containing $O(\alpha \log m)$ elements, one for each value encoded in the chunk. This reduces the lookup table size to $O(m^\alpha \log^2 m)$ bits, which implies that larger values of α are practical. The array can be searched using binary search in $O(\log n)$ worst-case time, or using doubling search in $O(\log d)$ worst-case time for `FingerSearch` type operations.

We define the following operations on blocks. All operations require constant time assuming constant α and that the blocks have size $O(\log m)$. Some operations increase the size of the blocks and we describe in Section 3 how the block sizes are bounded.

`BSearch`$^-$ (`BSearch`$^+$): Given a value v and a block B, these operations return the greatest (least) value in B that is less than or equal (greater than or equal) to v. This is just an application of the search method above.

`BInsert`: Given a value v and a block B, this operation inserts v into B. If v is less than the head for B, then our algorithm encodes that head by its difference from v and adds that code to the block. Otherwise, our algorithm searches B for the value v_j that should precede v. The gamma code for $v_{j+1} - v_j$ is deleted and replaced with the gamma codes for $v - v_j$ and $v_{j+1} - v$. (Some shift operations may be needed to make room for the new codes. Since each shift affects $O(\log m)$ bits, this requires constant time.)

`BDelete`: Given a block B and a value v_j contained in B, this operation deletes v_j from B. If v_j is the head for B, then its successor is decoded and made into the new head for B. Otherwise, our algorithm searches B for v_j. It deletes the gamma codes for $v_j - v_{j-1}$ and for $v_{j+1} - v_j$ and replaces them with the gamma code for $v_{j+1} - v_{j-1}$. (Part of the block may need to be shifted. As in the Insert case, this requires a constant number of shifts.)

`BMidSplit`: Given a block B of size b bits (where $b > 2M$), this operation splits off a new block B' such that B and B' each have size at least $b/2 - M$. It searches B for the first code c that starts after position $b/2 - M$ (using the second array stored with each table entry). Then c is decoded and made into the head for B'. The codes after c are placed in B', and c and its successors are deleted from B. B now contains at most $b/2$ bits of codes, and c contained at most M bits, so B' contains at least $b/2 - M$ bits. This takes constant time since codes can be copied $\Omega(\log m)$ bits at a time.

`BFirst`: Given a block B, this operation returns the head for B.

`BLast`: Given a block B, this operation scans to the end of B and returns the final value.

`BSplit`: Given a block B and a value v, this operation splits a new block B' off of B such that all values in B' are greater than v and all values in B are less than v. This is the same as `BMidSplit` except that c is chosen by a search rather than by its position in B. This operation returns v if it was in B.

`BJoin`: The join operation takes two blocks B and B' such that all values in B' are greater than the

greatest value from B. It concatenates B' onto B. To do this it first finds the greatest value v in B. It represents the head v' from B' with a gamma code for $v'-v$ and appends this code to the end of B. It appends the remaining codes from B' to B. This takes constant time since codes can be copied $\Omega(\log m)$ bits at a time.

`BRank`: To support the `BRank` operation the lookup table needs to be augmented: with each value is stored that value's rank within its chunk. To find the rank of an element v within a block B, our algorithm searches for the element while keeping track of the number of elements in each chunk skipped over.

`BSelect`: To support the `BSelect` operation the lookup table needs to be augmented: in addition to the decoding table, each chunk has an array containing its values. (This adds $O(m^\alpha \log^2 m)$ bits to the table, which does not alter its asymptotic space complexity.) To find the element with a given rank, our algorithm searches for the chunk containing that element, then accesses the appropriate index of the array.

3 Representation

To represent an ordered set $S = \{s_1, s_2, \ldots, s_n\}, s_i < s_{i+1}$, our approach maintains S as a set of blocks B_i where $B_i = \{s_{b_i}, s_{b_i+1}, \ldots, s_{b_{i+1}-1}\}$. The values $b_1 \ldots b_k$ are maintained such that the size of each block is between M and $4M$. The first block and the last block are permitted to be smaller than M. (Recall that $M = 2\lfloor \log m \rfloor + 1$ is the maximum possible length of a gamma code.) This property is maintained through all operations performed on S.

Lemma 3.1 *Given any set S from $U = \{0, \ldots, m-1\}$, let $|S| = n$. Given any assignment of b_i such that $\forall B_i, M \leq \texttt{size}(B_i) \leq 4M$, the total space used for the blocks is $O(n \log \frac{n+m}{n})$.*

Proof. We begin by bounding the space used for the gamma codes. The cost to gamma code the differences between every pair of consecutive elements in S is

$$\sum_{i=2}^{n} (2\lfloor \log(s_i - s_{i-1}) \rfloor + 1).$$

Since the logarithm is concave, this sum is maximized when the values are evenly spaced in the interval $1 \ldots m$; at that point the sum is $\sum_{i=2}^{n} (2 \log \frac{m}{n} + 1)$, which is $O(n \log \frac{m}{n} + n) = O(n \log \frac{m+n}{n})$.

The gamma codes contained in the blocks are a subset of the ones considered above (since the head of each block is not gamma coded). For every $\log m$ bits used by a head there are at least M bits used by gamma codes; since $M > 2 \log m$ the amount of additional space used by heads is at most half that used by gamma codes. ■

The blocks B_i are maintained in an ordered-dictionary structure D. The key for each block is its head. We refer to operations on D with a prefix D to differentiate them from operations on blocks and from the interface to our representation as a whole. D may use $O(\log m)$ bits to store each value. Since each value stored in D contains $\Theta(\log m)$ bits already, this increases our space bound by at most a constant factor. Our representation, as a whole, supports the following operations. They are not described as functional but can easily be made so: rather than change a block, our algorithm could delete it from the structure, copy it, modify the copy, and reinsert it into the structure.

`Search`$^-$: First, our algorithm calls `DSearch`$^-(k)$, returning the greatest block B with head $k' \leq k$. If $k' = k$, return k'. Otherwise, call `BSearch`$^-(k)$ on B and return the result.

`Search`$^+$: First, our algorithm calls `DSearch`$^-(k)$, returning the greatest block B with head $k' \leq k$. If $k' = k$, return k'. Otherwise, call `BSearch`$^+(k)$ on B. If this produces a value, return that value; otherwise, call `DSearch`$^+(k)$ and return the head of the result.

`Insert`: First, our algorithm calls `DSearch`$^-(k)$, returning the block B that should contain k. (If there is no block with head less than k, our algorithm uses `DSearch`$^+(k)$ to find a block instead.) Our algorithm then calls `BInsert`(k) on B. If `size`$(B) > 4M$, our algorithm calls `BMidSplit` on B and uses `DInsert` to insert the new block.

`Delete`: First, our algorithm calls `DSearch`$^-(k)$, returning the block B that contains the target element k. Then our algorithm calls `BDelete`(k) on B. If `size`$(B) < M$, our algorithm uses `DDelete` to delete B from D. It uses `DSearch`$^-$ to find the predecessor of B and `BJoin` to join the two blocks. This in turn may produce a block which is larger in size than $4M$, in which case a `BMidSplit` operation and a `DInsert` operation are needed as in the `Insert` case.

(Under rare circumstances, deleting a gamma-coded element from a block may cause it to grow in size by one bit. If this causes the block to exceed $4M$ in size, this is handled as in the `Insert` case.)

We define a "finger" to an element v to consist of a finger to the block B containing v in D.

`FingerSearch`: Our algorithm calls `DFingerSearch`(k) for the block B' which contains k. It then calls `BSearch`$^-(k)$ and returns the result.

`First`: Our representation calls `DFirst` and then `BFirst` and returns the result.

`Last`: Our representation calls `DLast` and then `BLast` and returns the result.

`Join`: Given two structures D_1 and D_2, our algorithm first checks the size of $B_1 = \texttt{DLast}(D_1)$ and $B_2 = \texttt{DFirst}(D_2)$. If $\texttt{size}(B_1) < M$, our algorithm uses `DSplit` to remove B_1 and its predecessor, `BJoin` to join them, and `BMidSplit` if the resulting block is oversized. It uses `DJoin` to join the resulting block(s) back onto D_1. If $\texttt{size}(B_2) < M$, our algorithm joins B_2 onto its successor using a similar method. Then our algorithm uses `DJoin` to join the two structures.

`Split`: Given an element k, our algorithm first calls $\texttt{DSplit}(k)$, producing structures D_1 and D_2. If the split operation returns a block B, then our algorithm uses `BDelete` on B to delete the head, uses `DJoin` to join B to D_2, and returns (D_1, k, D_2). Otherwise, our algorithm calls $\texttt{BSplit}(k)$ on the last block $\texttt{DLast}(D_1)$. If this produces an additional block, this block is inserted using `DJoin` into D_2.

`Rank`: The weighted rank of a block is defined to be the number of elements it contains. Our algorithm calls $\texttt{DSearch}^-(k)$ to find the block B that should contain k. It calls $\texttt{DWeightedRank}(B)$ and $\texttt{BRank}(k)$ and returns the sum.

`Select`: The size of a block is defined to be the number of elements it contains. Our algorithm uses $\texttt{DWeightedSelect}(r)$ to find the block B containing the target, then uses `BSelect` with the appropriate offset on B to find the target.

Lemma 3.2 *For an ordered universe $U = \{0, \ldots, m-1\}$, given an ordered dictionary structure (or comparison-based ordered set structure) D that uses $O(n \log m)$ bits to store n values, our blocking technique produces a structure that uses $O(n \log \frac{n+m}{n})$ bits.*

1. *If D supports* $\texttt{DSearch}^-$, $\texttt{DSearch}^+$, `DInsert`, *and* `DDelete`, *the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to operations of D.*
2. *If D supports* `DFingerSearch`, *the blocked set structure supports* `FingerSearch` *in $O(1)$ instructions and one call to* `DFingerSearch`. *If D supports* `DInsert` *and* `DDelete` *at a finger, then the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to* `DInsert` *and* `DDelete` *at a finger.*

```
union(S1,S2)
  if S1 = null then
    return S2
  if S2 = null then
    return S1
  (S2A,v,S2B) ← DSplit(S2,DFirst(S1))
  SB ← union(S2B,S1)
  return DJoin(S2A,SB)
```

Figure 2: Pseudocode for a `union` operation.

3. *If D supports the* `DFirst`, `DLast`, `DSplit`, *and* `DJoin` *operations, then the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to operations of D.*
4. *If D supports the* `DWeightedRank` *operation, then the blocked set structure supports the* `Rank` *operation in $O(1)$ instructions and one call to* `DWeightedRank`. *If D supports the* `DWeightedSelect` *operation, then the blocked set structure supports the* `Select` *operation using $O(1)$ instructions and one call to* `DWeightedSelect`.

The proof follows from the descriptions above.

4 Applications

By combining the `Split` and `Join` operations it is possible to implement efficient set union, intersection, and difference algorithms. An example implementation of `union` is shown in Figure 2. If `Split` and `Join` run in $O(\log |D_1|)$ time, then these set operation algorithms run in $O(k \log \frac{|D_1|+|D_2|}{k} + k)$ time, where k is the least possible number of blocks that we can break the two lists into before reforming them into one list. (This is the Block Metric of Carlsson et al. [7].)

As described in the introduction, the catenable ordered list structure of Kaplan and Tarjan [10] can be modified to support all of the operations described here in worst-case time. (To do this, we use `Split` as our search routine; to support `FingerSearch` we define a finger for k to be the result when the structure is split on k. To support weighted Rank and Select, we let each node in the structure store the weight of its subtree.) Thus our representation using their structure supports those operations in worst-case time using $O(n \log \frac{n+m}{n})$ bits. This structure may be somewhat unwieldy in practice, however.

If expected-case rather than worst-case bounds are acceptable, Treaps [16] are an efficient alternative. Treaps can be made to support the Split and Join operations by flipping the pointers along the left spine of the trees—each node along the left spine points to

$\|U\|$	$\|S\|$	Insert Times		Delete Times		Space Needed	
		Standard	Blocked	Standard	Blocked	Standard	Blocked
2^{20}	2^{10}	0.001	0.004	0.001	0.003	12	4.62
2^{20}	2^{12}	0.010	0.016	0.012	0.013	12	3.80
2^{20}	2^{14}	0.061	0.067	0.058	0.076	12	3.02
2^{20}	2^{16}	0.363	0.348	0.343	0.369	12	2.28
2^{20}	2^{18}	2.007	1.790	1.920	1.901	12	1.64
2^{25}	2^{10}	0.004	0.001	0.000	0.006	12	6.37
2^{25}	2^{12}	0.009	0.013	0.010	0.017	12	5.67
2^{25}	2^{14}	0.062	0.073	0.058	0.087	12	4.96
2^{25}	2^{16}	0.351	0.393	0.347	0.465	12	4.18
2^{25}	2^{18}	1.875	2.071	1.828	2.365	12	3.42
2^{30}	2^{10}	0.001	0.005	0.002	0.003	12	8.15
2^{30}	2^{12}	0.012	0.013	0.011	0.019	12	7.43
2^{30}	2^{14}	0.061	0.078	0.062	0.093	12	6.68
2^{30}	2^{16}	0.357	0.424	0.346	0.515	12	5.89
2^{30}	2^{18}	1.865	2.283	1.798	2.745	12	5.33

Table 1: Performance of a standard treap implementation versus our blocked treap implementation, averaged over ten runs. Time is in seconds; space is in bytes per value.

its parent instead of its left child. To split such a treap on a key k, an algorithm first travels up the left spine until it reaches a key greater than k, then splits the treap as normal. Seidel and Aragon showed that the expected path length of such a traversal is $O(\log |T_1|)$. By copying the path traversed this can be made purely functional.

5 Experimentation

We implemented our blocking technique in C using both treaps and red-black trees. Rather than the gamma code we use the nibble code, a code of our own devising which stores numbers using 4-bit "nibbles" [4]. Each nibble contains three bits of data and one "continue" bit. The continue bit is set to **0** if the nibble is the last one in the representation of that number, and **1** otherwise.

We decode blocks nibble-by-nibble rather than with a lookup table as described above. For very large problems, using such a table might improve performance.

We use a maximum block size of 46 nibbles (23 bytes) and a minimum size of 16 nibbles (8 bytes). We use one byte to store the number of nibbles in the block, for a total of 24 bytes per block.

We combined our blocking structure with two separate tree structures. The first is our own (purely functional) implementation of treaps [2]. Priorities are generated using a hash function on the keys. Each treap node maintains an integer key, a left pointer, and a right pointer, for a total of 12 bytes per node. In our blocked structure each node also keeps a pointer to its block. Since each block is 24 bytes, the total space usage is 40 bytes per treap node.

The second tree structure is the implementation of red-black trees [9] provided by the RedHat Linux implementation of the C++ Standard Template Library [1]. We used the `map<int, unsigned char*>` template for our blocked structure and the `set<int>` template for the unblocked equivalent. A red-black tree node includes a key, three pointers (left, right, and parent), and a byte indicating the color of the node. Since a C compiler allocates memory to data structures in multiples of 4, this requires a total of 20 bytes per node for the unblocked implementation, and 48 bytes for our blocked implementation.

We ran our simulations on a 1GHz processor with 1GB of RAM.

For each of our tree structures we tested the time needed to insert and delete elements. We used universe sizes of 2^{20}, 2^{25}, and 2^{30}, with varying numbers of elements. Elements were chosen uniformly from U. All elements in the set were inserted, then deleted. We calculated the time needed for insertion and deletion and the space required by each implementation, and computed the average over ten runs.

Results for the treap implementations are shown in are shown in Table 1. Our blocked version uses considerably less space than the non-blocked version: the improvement is between a factor of 1.45 and 7.3, depending on the density of the set. The slowdown caused by blocking varies but is usually less than 50%. (In fact, sometimes the blocked variant runs faster. We

$\|U\|$	$\|S\|$	Insert Times		Delete Times		Space Needed	
		Standard	Blocked	Standard	Blocked	Standard	Blocked
2^{20}	2^{10}	0.001	0.002	0.000	0.003	20	5.49
2^{20}	2^{12}	0.004	0.006	0.003	0.014	20	4.55
2^{20}	2^{14}	0.013	0.033	0.023	0.054	20	3.62
2^{20}	2^{16}	0.064	0.136	0.100	0.230	20	2.74
2^{20}	2^{18}	0.357	0.559	0.538	0.972	20	1.97
2^{25}	2^{10}	0.001	0.003	0.000	0.000	20	7.66
2^{25}	2^{12}	0.004	0.008	0.004	0.015	20	6.80
2^{25}	2^{14}	0.012	0.037	0.022	0.056	20	5.96
2^{25}	2^{16}	0.064	0.152	0.098	0.247	20	5.02
2^{25}	2^{18}	0.384	0.634	0.583	1.066	20	4.10
2^{30}	2^{10}	0.000	0.003	0.002	0.003	20	9.79
2^{30}	2^{12}	0.003	0.010	0.005	0.015	20	8.91
2^{30}	2^{14}	0.013	0.040	0.020	0.060	20	8.01
2^{30}	2^{16}	0.066	0.170	0.100	0.262	20	7.08
2^{30}	2^{18}	0.385	0.714	0.589	1.143	20	6.39

Table 2: Performance of a standard red-black tree implementation versus our blocked red-black tree implementation, averaged over ten runs. Time is in seconds; space is in bytes per value.

suspect this is because of caching and memory issues.)

Results for the red-black tree implementations are shown in Table 2. Here the space improvement is between a factor of 2 and 10. However the slowdown is sometimes as much as 150%.

Note that the STL red-black tree implementation is significantly faster than our treap implementation. In part this is because our treap structure is purely functional (and thus persistent). The red-black tree structure is not persistent.

For our treap data structure we also implemented the serial merge algorithm described in Section 4. We computed the time needed to merge sets of varying sizes in a universe of size 2^{20}. Results are shown in Figure 3. The slowdown caused by blocking was at most 150%.

$\|A\|$	$\|B\|$	Union Time	
		Standard	Blocked
2^{14}	2^{10}	0.003	0.011
2^{14}	2^{12}	0.015	0.036
2^{14}	2^{14}	0.036	0.086
2^{16}	2^{10}	0.005	0.014
2^{16}	2^{12}	0.028	0.048
2^{16}	2^{14}	0.067	0.157
2^{16}	2^{16}	0.151	0.370
2^{18}	2^{10}	0.006	0.015
2^{18}	2^{12}	0.043	0.059
2^{18}	2^{14}	0.119	0.208
2^{18}	2^{16}	0.293	0.703
2^{18}	2^{18}	0.616	1.540

Table 3: Performance of our serial merge algorithm implemented using standard treaps and blocked treaps. All values are averaged over ten runs. The universe size is 2^{20}. Time is in seconds.

References

[1] The C++ standard template library. `http://www.sgi.com/tech/stl/index.html`.

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

[3] D. Blandford and G. Blelloch. Index compression through document reordering. In *Data Compression Conference (DCC)*, pages 342–351, 2002.

[4] D. Blandford, G. Blelloch, D. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proc. 12th International Meshing Roundtable*, 2003.

[5] A. Brodnik and J. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[6] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 40–49, 1992.

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

[8] P. Elias. Universal codeword sets and representations

of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

[9] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[10] 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.

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

[12] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[14] R. Pagh. Low redundancy in static dictionaries with $O(1)$ worst case lookup time. *Lecture Notes in Computer Science*, 1644:595–??, 1999.

[15] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[16] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[17] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufman, 1999.