A Simple Linear-Space Data Structure for Constant-Time Range Minimum Query[☆]

Stephane Durocher[a,1,*], Robby Singh[a]

[a]*Department of Computer Science, University of Manitoba, Canada*

Abstract

We revisit the range minimum query problem and present a new $O(n)$-space data structure that supports range minimum queries in $O(1)$ time. The goal is to construct a static data structure that efficiently supports range minimum queries on a given list $A[0 : n-1]$ of n items drawn from a totally ordered set. Each range minimum query consists of an input pair of indices (i, j) for which the minimum element of the subarray $A[i : j]$ must be returned. Although previous data structures exist whose asymptotic bounds match ours, our goal is to introduce a new solution that is simple, intuitive, and practical without increasing asymptotic costs for query time or space. We analyze our new data structure theoretically and practically, the latter through an evaluation of its performance relative to implementations of four of the top range minimum query data structures.

Keywords: array range query, range minimum query, data structures, linear space, constant time

[☆]A preliminary version of this work appeared in the proceedings of the Conference on Space Efficient Data Structures, Streams and Algorithms [1].

[*]Corresponding author

Email addresses: `durocher@cs.umanitoba.ca` (Stephane Durocher), `singhr39@myumanitoba.ca` (Robby Singh)

[1]Work of the author is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

1. Introduction

1.1. Motivation

Along with the mean, median, and mode of a multiset, the minimum (equivalently, the maximum) is a fundamental statistic of data analysis for which efficient computation is necessary. Given a list $A[0 : n-1]$ of n items drawn from a totally ordered set, a *range minimum query (RMQ)* consists of an input pair of indices (i, j) for which the minimum element of $A[i : j]$ must be returned. The objective is to preprocess A to construct a data structure that supports efficient response to one or more subsequent range minimum queries, where the corresponding input parameters (i, j) are provided at query time.

Although the complete set of possible queries can be precomputed and stored using $\Theta(n^2)$ space, practical data structures require less storage while still enabling efficient response time. For all i, if $i = j$, then a range query must report $A[i]$. Consequently, any range query data structure for a list of n items requires $\Omega(n)$ words of storage space in the worst case [2]. This leads to a natural question: how quickly can an $O(n)$-space data structure answer a range minimum query?

Previous $O(n)$-space data structures exist that provide $O(1)$-time RMQ (e.g., [3, 4, 5, 6, 7], see Section 2). These solutions typically require a transformation or invoke a property that enables the volume of stored precomputed data to be reduced while allowing constant-time access and RMQ computation. Each such solution is a conceptual organization of the data into a compact table for efficient reference; essentially, the algorithm reduces to a clever table lookup. In this paper our objective is not to minimize the total number of bits occupied by the data structure (our solution is not succinct) but rather to present a simple and more intuitive method for organizing the precomputed data to support RMQ efficiently, i.e., an $O(n)$-space data structure that supports RMQ in $O(1)$ time. Our solution combines new ideas with techniques from various previous data structures: van Emde Boas trees [8], resizable arrays [9], range mode query [10, 11], one-sided RMQ [3], and a linear-space data structure that supports RMQ in $O(\sqrt{n})$ time. The resulting RMQ data structure stores efficient representations of the data to permit direct lookup without requiring the indirect techniques employed by previous solutions (e.g., [3, 4, 5, 6, 12, 13, 14]) such as transformation to a lowest common ancestor query, Cartesian trees, Eulerian tours, or the Four Russians speedup. The data structure's RMQ algorithm is astonishingly simple: it can be implemented as a single if statement with four branches, each of which returns the minimum of at most three values retrieved from precomputed tables (see the pseudocode for Algorithm 2 in Section 3.3).

The RMQ problem is sometimes defined such that a query returns only the index of the minimum element instead of the minimum element itself. In particular, this is the case for succinct data structures that support $O(1)$-time RMQ using only $O(n)$ bits of space [7, 15, 16, 17] (see Section 2). In order to return the actual minimum element, say $A[i]$, in addition to its index i, any such data structure must also store the values from the input array A, corresponding to a lower bound of $\Omega(n \log u)$ bits of space in the worst case when element are drawn from a universe of size u or, equivalently, $\Omega(n)$ *words* of space (this lower bound also applies to other array range query problems [2]). Therefore, a range query data structure that uses $o(n)$ words of space requires storing the input array A separately, resulting in total space usage of $\Theta(n)$ words of space in the worst case. In this paper we require that a RMQ return the minimum element. Our RMQ data structure stores all values of A internally and matches the optimal asymptotic bounds of $O(n)$ words of space and $O(1)$ query time.

1.2. Definitions and Model of Computation

We assume the RAM word model of computation with word size $\Theta(\log u)$, where elements are drawn from a universe $U = \{-u, \ldots, u-1\}$ for some fixed integer $u \geq n$. Unless specified otherwise, memory requirements are expressed in word-sized units. We assume the usual set of $O(1)$-time primitive operations: basic integer arithmetic (addition, subtraction, multiplication, division, and modulo), bitwise logic, and bit shifts. We do not assume $O(1)$-time exponentiation nor, consequently, radicals. When the base operand is a power of two and the result is an integer, however, these operations can be computed using bitwise left or right shifts. All arithmetic computations are on integers in U, and integer division is assumed to return the floor of the quotient. Finally, our data structure only requires finding the binary logarithm of integers in the range $\{0, \ldots, n\}$, i.e., the index of the most significant non-zero bit. The complete set of values can be precomputed and stored in a table of size $O(n)$ to provide $O(1)$-time reference for the log and $\log\log$ operations[2] at query time, regardless of whether logarithm computation is included in the RAM model's set of primitive operations.

A common technique used in array range searching data structures (e.g., [3, 10, 11]) is to partition the input array $A[0 : n-1]$ into a sequence of $\lceil n/b \rceil$ blocks, each of size b (except possibly for the last block whose size is $[(n-1) \bmod b] + 1$). A query range $A[i : j]$ spans between 0 and $\lceil n/b \rceil$ complete blocks. We refer to the sequence of complete blocks contained within $A[i : j]$ as the *span*, to the elements of $A[i : j]$ that precede the span as the *prefix*, and to the elements of $A[i : j]$ that succeed the span as the *suffix*. See Figure 1. One or more of the prefix, span, and suffix may be empty. When the span is empty, the prefix and suffix can lie either in adjacent blocks, or in the same block; in the latter case the prefix and suffix are equal (or one is empty).

We summarize the asymptotic resource requirements of a given RMQ data structure by the ordered pair $\langle s(n), t(n) \rangle$, where $s(n)$ denotes the storage space it requires in words and $t(n)$ denotes its worst-case RMQ time for an array of size n. Our discussion focuses primarily on these two measures of efficiency; other measures of interest include the preprocessing time and the update time. Note that similar notation is sometimes used to pair precomputation time and query time (e.g., [3, 6]).

[2]Throughout this manuscript, $\log a$ denotes the binary logarithm, $\log_2 a$.

2. Related Work

Multiple $\langle \omega(n), O(1) \rangle$ solutions are known, including precomputing RMQs for all query ranges in $\langle O(n^2), O(1) \rangle$, and precomputing RMQs for all ranges of length 2^k for some $k \in \mathbb{Z}^+$ in $\langle O(n \log n), O(1) \rangle$ (Sparse Table Algorithm) [3, 6]. In the latter case, a query is decomposed into two (possibly overlapping) precomputed queries. Similarly, $\langle O(n), \omega(1) \rangle$ solutions exist, including the $\langle O(n), O(\sqrt{n}) \rangle$ data structure described in Section 3.1.

Several $\langle O(n), O(1) \rangle$ RMQ data structures exist, many of which depend on the equivalence between the range minimum query and lowest common ancestor (LCA) problems. Harel and Tarjan [13] gave the first $\langle O(n), O(1) \rangle$ solution to LCA. Their solution was simplified by Schieber and Vishkin [14]. Berkman and Vishkin [5] showed how to solve the LCA problem in $\langle O(n), O(1) \rangle$ by transformation to RMQ using an Euler tour. This method was simplified by Bender and Farach-Colton [3] to give an ingenious solution which we briefly describe below. Comprehensive overviews of previous solutions are given by Davoodi [18] and Fischer [19].

The array $A[0 : n-1]$ can be transformed into a Cartesian tree $\mathcal{C}(A)$ on n nodes such that a RMQ on $A[i : j]$ corresponds to the LCA of the respective nodes associated with i and j in $\mathcal{C}(A)$. When each node in $\mathcal{C}(A)$ is labelled by its depth, an Eulerian tour on $\mathcal{C}(A)$ (i.e., the depth-first traversal sequence on $\mathcal{C}(A)$) gives an array $B[0 : 2n-2]$ for which any two adjacent values differ by ± 1. Thus, a LCA query on $\mathcal{C}(A)$ corresponds to a ± 1-RMQ on B. Array B is partitioned into blocks of size $(\log n)/2$. Separate data structures are constructed to answer queries that are contained within a single block of B and those that span multiple blocks, respectively. In the former case, the ± 1 property implies that the number of unique blocks in B is $O(\sqrt{n})$; all $O(\sqrt{n} \log^2 n)$ possible RMQs on blocks of B are precomputed (the Four Russians technique [20]). In the latter case, a query can be decomposed into a prefix, span, and suffix (see Section 1.2). RMQs on the prefix and suffix are contained within respective single blocks, each of which can be answered in $O(1)$ time as in the former case. The span covers zero or more blocks. The minimum of each block of B is precomputed and stored in $A'[0 : 2n/\log n - 1]$. A RMQ on A' (the minimum value in the span) can be found in $\langle O(n), O(1)\rangle$ using the $\langle O(n' \log n'), O(1)\rangle$ sparse table data structure [3, 6] mentioned above due to the shorter length of A' (i.e., $n' = 2n/\log n$).

Fischer and Heun [6] use similar ideas to give a $\langle O(n), O(1)\rangle$ solution to RMQ that applies the Four Russians technique to any array (i.e., it does not require the ± 1 property) on blocks of length $\Theta(\log n)$. Yuan and Atallah [21] examine RMQ on multidimensional arrays and give a new one-dimensional $\langle O(n), O(1)\rangle$ solution that uses a hierarchical binary decomposition of $A[0 : n-1]$ into $\Theta(n)$ canonical intervals, each of length 2^k for some $k \in \{1, \ldots, \log n\}$, and precomputed queries within blocks of length $\Theta(\log n)$ (similar to the Four Russians technique).

When only the minimum's index is required, Sadakane [17] gives a succinct data structure requiring $4n + o(n)$ bits that supports $O(1)$-time RMQ. Fischer and Heun [15, 16] and Davoodi et al. [7] reduce the space requirements to $2n + o(n)$ bits. Finally, the RMQ problem has been examined in the dynamic setting [22, 18], in two and higher dimensions [23, 24, 25, 17, 26, 21], and on trees and directed acyclic graphs [4, 22, 24].

3. A New $\langle O(n), O(1)\rangle$ RMQ Data Structure

The new data structure is described in steps, starting with a previous $\langle O(n), O(\sqrt{n})\rangle$ data structure, extending it to $\langle O(n \log\log n), O(\log\log n)\rangle$ by applying the technique recursively, eliminating recursion to obtain $\langle O(n \log\log n), O(1)\rangle$, and finally reducing the space to $\langle O(n), O(1)\rangle$. To simplify the presentation, suppose initially that the input array A has size $n = 2^{2^k}$, for some $k \in \mathbb{Z}^+$; as described in Section 3.5, removing this constraint and generalizing to an arbitrary n is easily achieved without any asymptotic increase in time or space.

3.1. $\langle O(n), O(\sqrt{n})\rangle$ Data Structure

The following $\langle O(n), O(\sqrt{n})\rangle$ data structure is known in RMQ folklore (e.g., [27]) and has similar high-level structure to the ± 1-RMQ algorithm of Bender

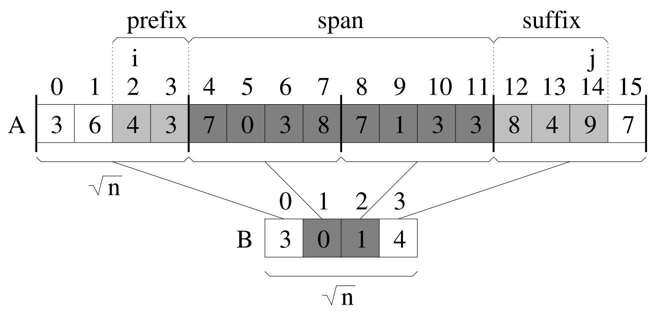

Figure 1: A $\langle O(n), O(\sqrt{n})\rangle$ data structure: the array A is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in array B. A range minimum query $A[2 : 14]$ is processed by finding the minimum of the respective minima of the prefix $A[2 : 3]$, the span $A[4 : 11]$ (determined by examining array B), and the suffix $A[12 : 14]$. In this example this corresponds to $\min\{3, 0, 4\} = 0$.

and Farach-Colton [3, Section 4]. While suboptimal on its own, and often overlooked in favour of more efficient solutions, this data structure forms the basis for our new $\langle O(n), O(1)\rangle$ data structure.

The input array $A[0 : n-1]$ is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in a table $B[0 : \sqrt{n}-1]$. See Figure 1. A query range spans between zero and $\sqrt{n}$ complete blocks. The minimum of the span is computed by iteratively examining the corresponding values in B. Similarly, the respective minima of the prefix and suffix are computed by iteratively examining their elements. The range minimum corresponds to the minimum of these three values. Since the prefix, suffix, and array B each contain at most $\sqrt{n}$ elements, the worst-case query time is $\Theta(\sqrt{n})$. The total space required by the data structure is $\Theta(n)$. Precomputation requires only a single pass over the input array in $\Theta(n)$ time. Updates (e.g., set $A[i] \leftarrow x$) require $\Theta(\sqrt{n})$ time in the worst case; whenever an array element equal to its block's minimum is increased, the block must be scanned to identify the new minimum.

3.2. $\langle O(n \log\log n), O(\log\log n)\rangle$ Data Structure

One-sided range minimum queries (where one endpoint of the query range coincides with one end of the array A) are trivially precomputed [3] and stored in arrays C and C', each of size n, where for each i,

$$C[i] = \begin{cases} \min\{A[i], C[i-1]\} & \text{if } i > 0, \\ A[0] & \text{if } i = 0, \end{cases}$$

$$C'[i] = \begin{cases} \min\{A[i], C'[i+1]\} & \text{if } i < n-1, \\ A[n-1] & \text{if } i = n-1. \end{cases} \tag{1}$$

Any subsequent one-sided RMQ on $A[0:j]$ or $A[j:n-1]$ can be answered in $O(1)$ time by referring to $C[j]$ or $C'[j]$.

The $\langle O(n), O(\sqrt{n})\rangle$ solution discussed in Section 3.1 includes three range minimum queries on subproblems of size $\sqrt{n}$, of which at most one is two-sided. In particular, if the span is non-empty, then the query on array B is two-sided, and the queries on the prefix and suffix are one-sided. Similarly, if the query range is contained in a single block, then there is a single two-sided query and no one-sided queries. Finally, if the query range intersects exactly two blocks, then there are two one-sided queries (one each for the prefix and suffix) and no two-sided queries.

Thus, upon adding arrays C and C' to the data structure, at most one of the three (or fewer) subproblems requires $\omega(1)$ time to identify its range minimum. This search technique can be applied recursively on two-sided queries. By limiting the number of recursive calls to at most one and by reducing the problem size by an exponential factor of $1/2$ at each step of the recursion, the resulting query time is bounded by the following recurrence (similar to that achieved by van Emde Boas trees [8]):

$$T(n) \leq \begin{cases} T(\sqrt{n}) + O(1) & \text{if } n > 2, \\ O(1) & \text{if } n \leq 2 \end{cases}$$
$$\in O(\log\log n). \tag{2}$$

Table 1: The xth level is a sequence of $b_x(n)$ blocks of size $s_x(n)$.

level	x	0	1	2	…	i	…	$\log\log n - 2$	$\log\log n - 1$	$\log\log n$
number of blocks	$b_x(n)$	$n/2$	$n/4$	$n/16$	…	$n/2^{2^i}$	…	$n^{3/4}$	$\sqrt{n}$	1
block size	$s_x(n)$	2	4	16	…	2^{2^i}	…	$n^{1/4}$	$\sqrt{n}$	n

Each step invokes at most one recursive RMQ on a subarray of size $\sqrt{n}$. Each recursive call is one of two types: i) a recursive call on array B (a two-sided query to compute the range minimum of the span) or ii) a recursive call on the entire query range (contained within a single block).

Recursion can be avoided entirely for determining the minimum of the span (a recursive call of the first type). Since there are $\sqrt{n}$ blocks, $\binom{\sqrt{n}+1}{2} \leq n$ distinct spans are possible. As is done in the range mode query data structure of Krizanc et al. [11], the minimum of each span can be precomputed and stored in a table D of size n. Any subsequent RMQ on a span can be answered in $O(1)$ time by reference to table D. Consequently, tables C, C', and D suffice, and table B can be eliminated.

The result is a hierarchical data structure containing $\log\log n+1$ levels which we number $0, \ldots, \log\log n$, where the xth level[3] is a sequence of $b_x(n) = n/2^{2^x}$ blocks of size $s_x(n) = n/b_x(n) = 2^{2^x}$. See Table 1. Thus, level 0 is a sequence of $n/2$ blocks of size 2, level 1 is a sequence of $n/4$ blocks of size 4, level 2 is a sequence of $n/16$ blocks of size 16, and so on, until level $\log\log n - 2$ is a sequence of $n^{3/4}$ blocks of size $n^{1/4}$, level $\log\log n - 1$ is a sequence of $\sqrt{n}$ blocks of size $\sqrt{n}$, and level $\log\log n$ is a single block of size n (i.e., the entire array A).

Generalizing (1), for each $x \in \{0, \ldots, \log\log n\}$ the new arrays C_x and C'_x

[3]Level $\log\log n$ is included for completeness since we refer to the size of the parent of blocks on level x, for each $x \in \{0, \ldots, \log\log n - 1\}$. The only query that refers to level $\log\log n$ directly is the complete array: $i = 0$ and $j = n - 1$. The global minimum for this singular case can be stored using $O(1)$ space and updated in $O(\sqrt{n})$ time as described in Section 3.1.

are defined by

$$C_x[i] = \begin{cases} \min\{A[i], C_x[i-1]\} & \text{if } i \neq 0 \bmod s_x(n), \\ A[i] & \text{if } i = 0 \bmod s_x(n), \end{cases}$$

$$C'_x[i] = \begin{cases} \min\{A[i], C'_x[i+1]\} & \text{if } (i+1) \neq 0 \bmod s_x(n), \\ A[i] & \text{if } (i+1) = 0 \bmod s_x(n). \end{cases}$$

We refer to a sequence of blocks on level x that are contained in a common block on level $x+1$ as *siblings* and to the common block as their *parent*. Each block on level $x+1$ is a parent to $s_{x+1}(n)/s_x(n) = s_x(n)$ siblings on level x. Thus, any query range contained in some block at level $x+1$ covers at most $s_x(n)$ siblings at level x, resulting in $\Theta(s_x(n)^2) = \Theta(s_{x+1}(n))$ distinct possible spans within a block at level $x+1$ and $\Theta(s_{x+1}(n) \cdot b_{x+1}(n)) = \Theta(n)$ total distinct possible spans at level $x+1$, for any $x \in \{0, \ldots, \log\log n - 1\}$. These precomputed range minima are stored in table D, such that for every $x \in \{0, \ldots, \log\log n - 1\}$, every $b \in \{0, \ldots, b_{x+1}(n) - 1\}$, and every $\{i, j\} \subseteq \{0, \ldots, s_x(n) - 1\}$, $D_x[b][i][j]$ stores the minimum of the span $A[b \cdot s_{x+1}(n) + i \cdot s_x(n) : b \cdot s_{x+1}(n) + (j+1)s_x(n) - 1]$.

This gives the following recursive algorithm whose worst-case time is bounded by (2):

Algorithm 1

```
   RMQ($i, j$)
1  if $i = 0$ and $j = n - 1$
2      return $minA$ // global minimum
3  else
4      return RMQ($\log\log n - 1, i, j$)
       // start recursion at the top level
   RMQ$(x, i, j)$
 1 $b_i \leftarrow \lfloor i/s_x(n) \rfloor$          // block indices
 2 $b_j \leftarrow \lfloor j/s_x(n) \rfloor$
 3 $b \leftarrow \lfloor i/s_{x+1}(n) \rfloor$        // parent block
 4 if $x = 0$                       // base case
 5     return $\min\{A[i], A[j]\}$  // range size $\leq 2$
 6 else if $b_i = b_j$
 7     return RMQ$(x - 1, i, j)$
       // two-sided recursive RMQ
 8 else if $b_j - b_i \geq 2$       // non-empty span
 9     return $\min\{C'_x[i], C_x[j], D_x[b][b_i + 1][b_j - 1]\}$
       // 2 one-sided RMQs + precomputed span
10 else                             // empty span
11     return $\min\{C'_x[i], C_x[j]\}$ // 2 one-sided RMQs
```

The space required by array D_x for each level $x < \log\log n$ is

$$O\left(s_x(n)^2 \cdot b_{x+1}(n)\right) = O\left(s_{x+1}(n) \cdot b_{x+1}(n)\right) = O(n).$$

Since arrays C_x and C'_x also require $O(n)$ space at each level, the total space required is $O(n)$ per level, resulting in $O(n \log\log n)$ total space for the complete data structure.

For each level $x < \log\log n$, precomputing arrays C_x, C'_x, and D_x is easily achieved in $O(n \cdot s_x(n)) = O(n \cdot 2^{2^x})$ time per level, or $O(n^{3/2})$ total time. Each update requires $O(s_x(n))$ time per level, or $O(\sqrt{n})$ total time per update. This gives the following lemma:

Lemma 1. *Given any $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$ and any array $A[0 : n-1]$, Algorithm 1 supports range minimum queries on A in $O(\log\log n)$ time using a data structure of size $O(n \log\log n)$.*

3.3. $\langle O(n \log\log n), O(1) \rangle$ Data Structure

Each step of Algorithm 1 described in Section 3.2 invokes at most one recursive call on a subarray whose size decreases exponentially at each step. Specifically, the only case requiring $\omega(1)$ time occurs when the query range is contained within a single block of the current level. In this case, no actual computation or table lookup occurs locally; instead, the result of the recursive call is returned directly (see Line 7 of Algorithm 1). As such, the recursion can be eliminated by jumping directly to the level of the data structure at which the recursion terminates, that is, the highest level for which the query range is not contained in a single block. Any such query can be answered in $O(1)$ time using a combination of at most three references to arrays C_x, C'_x, and D_x (see Lines 9 and 11 of Algorithm 1). We refer to the corresponding level of the data structure as the *query level*, whose index we denote by ℓ.

More precisely, Algorithm 1 makes a recursive call whenever $b_i = b_j$, where b_i and b_j denote the respective indices of the blocks containing i and j in the current level (see Line 7 of Algorithm 1). Thus, we seek to identify the highest level for which $b_i \neq b_j$. In fact, it suffices to identify the highest level $\ell \in \{0, \ldots, \log\log n - 1\}$ for which no query of size $j - i + 1$ can be contained within a single block. While the query could span the boundary of (at most) two adjacent blocks at higher levels, it must span at least two blocks at all levels less than or equal to ℓ. In other words, the size of the query range is bounded by

$$\begin{aligned} & & s_\ell(n) <& j - i + 1 \leq s_{\ell+1}(n) \\ \Leftrightarrow & & 2^{2^\ell} <& j - i + 1 \leq 2^{2^{\ell+1}} \\ \Leftrightarrow & \log\log(j-i+1) - 1 \leq & \ell <& \log\log(j-i+1) \\ \Rightarrow & & \ell =& \lfloor \log\log(j-i) \rfloor. \end{aligned}$$

As discussed in Section 1.2, since we only require finding binary logarithms of positive integers up to n, these values can be precomputed and stored in a table of size $O(n)$. Consequently, the value ℓ can be computed in $O(1)$ time at query time, where each logarithm is found by a table lookup. Similarly, the values $s_x(n)$ and $b_x(n)$ can be precomputed for all $x \in \{0, \ldots, \log\log n\}$.

This gives the following simple algorithm whose worst-case running time is constant (note the absence of loops and recursive calls):

Algorithm 2

```
   RMQ$(i, j)$
 1 $\ell \leftarrow \lfloor \log\log(j-i) \rfloor$          // level
 2 $b_i \leftarrow \lfloor i/s_\ell(n) \rfloor$             // block indices
 3 $b_j \leftarrow \lfloor j/s_\ell(n) \rfloor$
 4 $b \leftarrow \lfloor i/s_{\ell+1}(n) \rfloor$           // parent block
 5 if $i = 0$ and $j = n-1$
 6     return $minA$                 // global minimum
 7 else if $j - i < 2$               // range size $\leq 2$
 8     return $\min\{A[i], A[j]\}$
 9 else if $b_j - b_i \geq 2$         // non-empty span
10     return $\min\{C'_\ell[i], C_\ell[j], D_\ell[b][b_i+1][b_j-1]\}$
       // 2 one-sided RMQs + precomputed span
11 else                              // empty span
12     return $\min\{C'_\ell[i], C_\ell[j]\}$ // 2 one-sided RMQs
```

Although the query algorithm differs from Algorithm 1, the data structure remains unchanged except for the addition of precomputed values for logarithms which require $O(n)$ additional total space. As such, the space remains $O(n \log\log n)$ while the query time is reduced to $O(1)$ in the worst case. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively. This gives the following lemma:

Lemma 2. *Given any $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$ and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n \log\log n)$.*

3.4. $\langle O(n), O(1) \rangle$ Data Structure

The data structures described in Sections 3.2 and 3.3 store exact precomputed values in arrays C_x, C'_x, and D_x. That is, for each a and each x,

$C_x[a]$ stores $A[b]$ for some b (similarly for C'_x and D_x). If the array A is accessible during a query, then it suffices to store the relative index $b - a$ instead of storing $A[b]$. Thus, $C_x[a]$ stores $b - a$ and the returned value is $A[C_x[a] + a] = A[(b - a) + a] = A[b]$. Since the range minimum is contained in the query range $A[i : j]$ we get that $\{a, b\} \subseteq \{i, \ldots, j\}$ and, therefore,

$$|b - a| \leq j - i + 1 \leq s_{\ell+1}(n).$$

Consequently, for each level x, $\log(s_{x+1}(n)) = 2^{x+1}$ bits suffice to encode any value stored in C_x, C'_x, or D_x. Therefore, for each level x, each table C_x, C'_x, and D_x can be stored using $O(n \cdot 2^{x+1})$ bits. Observe that

$$\sum_{x=0}^{\log\log n - 1} n \cdot 2^{x+1} < 2n \log n. \tag{3}$$

Consequently, the total space occupied by the tables C_x, C'_x, and D_x can be compacted into $O(n \log n)$ bits or, equivalently, $O(n)$ words of space. We now describe how to store this compact representation to enable efficient access. For each $i \in \{0, \ldots, n - 1\}$, the values $C_0[i], \ldots, C_{\log\log n - 1}[i]$ can be stored in two words by (3). Specifically, the first word stores $C_{\log\log n - 1}[i]$ and for each $x \in \{0, \ldots, \log\log n - 2\}$, bits $2^{x+1} - 1$ through $2^{x+2} - 2$ store the value $C_x[i]$. Thus, all values $C_0[i], \ldots, C_{\log\log n - 2}[i]$ are stored using

$$\sum_{i=0}^{\log\log n - 2} 2^{x+1} = \log n - 2 < \log u$$

bits, i.e., a single word, where $\log u$ denotes the word size under the RAM model. The value $C_x[i]$ can be retrieved using a bitwise left shift followed by a right shift or, alternatively, a bitwise logical AND with the corresponding sequence of consecutive 1 bits (all $O(\log\log n)$ bit sequences can be precomputed). An analogous argument applies to the arrays C'_x and D, resulting in $O(n)$ space for the complete data structure.

To summarize, the query algorithm is unchanged from Algorithm 2 and the corresponding query time remains constant, but the data structure's required space is reduced to $O(n)$. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively. This gives the following lemma:

Lemma 3. *Given any $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$ and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

3.5. Generalizing to an Arbitrary Array Size n

To simplify the presentation in Sections 3.1 to 3.4 we assumed that the input array had size $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$. As we show in this section, generalizing the data structure to an arbitrary positive integer n while maintaining the same asymptotic bounds on space and time is straightforward.

Let m denote the largest value no larger than n for which Lemma 3 applies. That is,

$$\begin{aligned} & m = 2^{2^{\lfloor \log \log n \rfloor}} \\ \Rightarrow \quad & m \leq n < m^2 \\ \Rightarrow \quad & n/m < \sqrt{n}. \end{aligned} \tag{4}$$

Define a new array $A'[0 : n' - 1]$, where $n' = m\lceil n/m \rceil$, that corresponds to the array A padded with dummy data[4] to round up to the next multiple of m. Thus,

$$\forall i \in \{0, \ldots, n'-1\},\ A'[i] = \begin{cases} A[i] & \text{if } i < n \\ +\infty & \text{if } i \geq n. \end{cases}$$

Since $n' = 0 \bmod m$, partition array A' into a sequence of blocks of size m. The number of blocks in A' is $\lceil n/m \rceil < \lceil \sqrt{n} \rceil$.

By (4) and Lemma 3, for each block we can construct a data structure to support RMQ on that block in $O(1)$ time using $O(m)$ space per block. Therefore, the total space required by all blocks in A' is $O(\lceil n/m \rceil \cdot m) = O(n)$. Construct arrays C, C', and D as before on the top level of array A' using the blocks of size m. The arrays C and C' each require $O(n') = O(n)$ space. The array D

[4]For implementation, it suffices to store $u - 1$ (the largest value in the universe U) instead of $+\infty$ as the additional values.

requires $O(\lceil n/m \rceil^2) \subseteq O(n)$ space by (4). Therefore, the total space required by the complete data structure remains $O(n)$.

Each query is performed as in Algorithm 2, except that references to C, C', and D at the top level access the corresponding arrays (which are stored separately from C_x, C'_x, and D_x for the lower levels). Therefore, the query time is increased by a constant factor for the first step at the top level, and the total query time remains $O(1)$.

As discussed in Section 3.4, the number of bits required to store a value in C_x, C'_x or D_x doubles from levels x to $x+1$. Since $2n \log n$ bits suffice to store all data for levels $\ell \leq m$, it follows that $4n \log n$ bits suffice to store the complete data structure, i.e., levels 0 through $m+1$, including the new top level.

This gives the following theorem:

Theorem 4 (Main Result). *Given any $n \in \mathbb{Z}^+$, and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

4. Simulation and Performance Evaluation

We implemented our RMQ data structure along with four other efficient RMQ data structures, and ran a series of tests to compare the performance of these data structures on input arrays ranging in size from 2^7 to 2^{26} integers, measuring range minimum query times, memory usage (total space required by the data structure), and preprocessing times (time required to build the data structure on the given input array). The four additional data structures selected were those of Fischer and Heun [15], Bender and Farach-Colton [3], Sadakane [17], and Ohlebusch and Gog [28]. All of these data structures are $\langle O(n), O(1) \rangle$; that is, they can be stored using $O(n)$ total space and support $O(1)$-time range minimum queries in the worst case. The goal of this experiment was to evaluate and compare the constants in these respective linear and constant costs for each data structure. For simplicity, we refer to each RMQ data structures by its first author. See Section 2 for a discussion of these data structures.

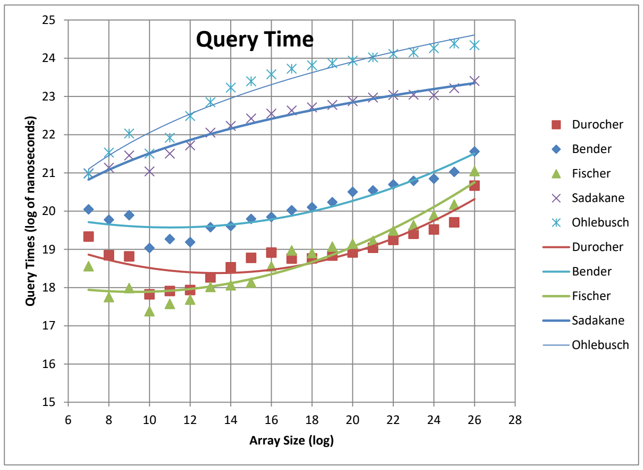

Figure 2: Average query times plotted as the log of time in nanoseconds, as a function of the input array size

In the remainder of this section we describe the parameters of the simulation and analyze its results. In brief, the two data structures with the fastest query times were ours and Fischer's, with Fischer providing faster query time for small data ($n \in \{2^7, \ldots, 2^{18}\}$) and ours providing faster query time for large data ($n \in \{2^{19}, \ldots, 2^{26}\}$). Following our data structure and Fischer's, in order of increasing query times were Bender, Sadakane, and Ohlebusch. With respect to space, the data structures that used the least space were Sadakane and Ohlebusch, followed by Fischer, ours, and Bender. Except for small data ($n \leq 2^9$), the relative ordering of data structures in terms of space use remained effectively identical throughout the tests. See Figures 2 through 4 for plots of average query times, space, and preprocessing times as functions of the input size.

4.1. Simulation Parameters

Our tests were conducted on a laptop computer running Windows 10 with an Intel Core i7-4720HQ CPU and 16GB of RAM. For each array size[5], 2^i, for all

[5]For $n = 2^{27}$, Bender's data structure used approximately 80% of the available 16GB of memory, resulting in significantly increased query times. To avoid biasing results, we tested

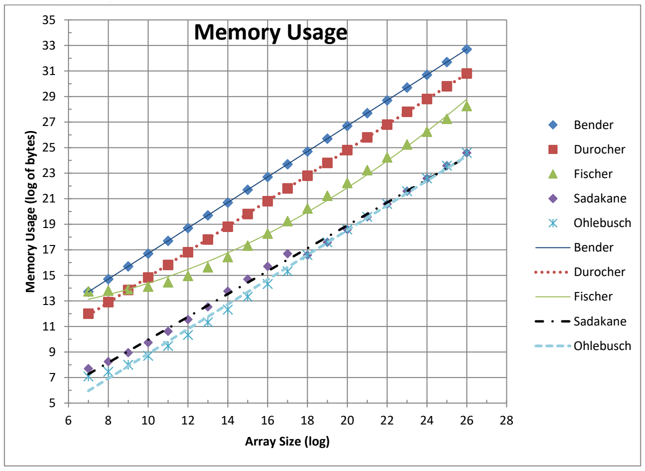

Figure 3: Memory use plotted as the log of space in bytes, as a function of the input array size

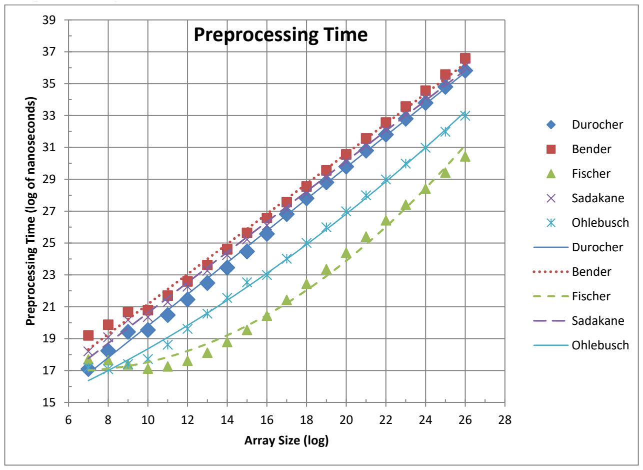

Figure 4: Similar to memory use, preprocessing times are nearly linear, again with the exception of Fischer for small n.

$i \in \{7, 8, \ldots, 26\}$, we performed nine[6] trials. For each trial we created an array of n integers, with each integer stored as a 32-bit word[7] selected uniformly at random from $\{0, \ldots, 2^{16} - 1\}$. The five data structures were constructed using this same input array, and the preprocessing time for constructing each was measured in nanoseconds. We recorded the memory used by each data structure in bytes. For each trial we generated a sequence of 1000 pairs of random query indices (i, j), where $0 \leq i \leq j \leq n - 1$; this same sequence of index pairs was applied to each of the five data structures. Memory used by the data structures was not freed until the end of the trial, with all five data structures existing in memory simultaneously during the querying phase to avoid any bias related to the amount of available RAM. For each data structure and each query index pair (i, j), the RMQ query on (i, j) was performed 1000 times and the time required measured in nanoseconds; this repetition was intended to reduce inaccuracies on the measurement of query times due to the extremely quick queries times. The time required for 1000 repetitions was recorded for each of the 1000 RMQs in the sequence, and the average of these 1000 values was recorded for each data structure. Nine trials were executed for each n to reduce any potential bias that particular data sets might prove cumbersome for one or more of the data structures being evaluated, with the median measurements of each of the nine trials recorded for each data structure and each n.

All five data structure were implemented in C++. The implementations of Sadakane and Ohlebusch were based on those in Gog's Succinct Data Structure Library [29], the implementation of Fischer was based on his own code [30], and the implementation of Bender was based on that by Walsh [31].

all data structures on array sizes up to 2^{26}.

[6]An odd number of trials was selected to ensure unique median values.

[7]Although larger integers could have been used, the maximum value $2^{16} - 1 = 65535$ minimized the need to modify existing code used for some of the data structures.

4.2. Analysis

Query times, which are theoretically constant, should correspond to horizontal plots; see Figure 2. Slightly increasing trends were measured, possibly due to storing and accessing larger tables. The data structures with the fastest query times were ours and Fischer, with Fischer providing faster query time for small data ($n \in \{2^7, \ldots, 2^{18}\}$) and ours providing faster query time for large data ($n \in \{2^{19}, \ldots, 2^{26}\}$). The relative differences in query times remained mostly similar for all n, with query times for Bender approximately 2 times slower than for ours and Fischer, and query times for Sadakana and Ohlebusch approximately 8 and 32 slower than for ours and Fischer.

All of the data structures implemented use $O(n)$ space, as confirmed by the clear linear growth in space use, except for Fischer, whose space use increases non-linearly when $n < 2^{12}$, likely due to the $o(n)$ overhead cost whose effect is more apparent for small n. See Figure 3. The data structures that used the least space were Sadakane and Ohlebusch, followed by Fischer, ours, and Bender. Except for small data ($n \leq 2^9$), the relative ordering of data structures in terms of space use remained effectively identical throughout the tests.

Finally, preprocessing times were nearly linear (with the exception of Fischer for $n \geq 2^{14}$, again likely due to overhead for small n), with Fischer measuring the fastest preprocessing times, followed by Ohlebusch, with preprocessing time approximately four times that of Fischer, followed by our data structure, with preprocessing time approximately eight that of Fischer, followed by Sadakane and Bender, with preprocessing times only marginally higher than ours. See Figure 4.

Query time and space are the critical performance constraints in typical RMQ applications. If query time is more important, then Fischer or our data structure are good choices, whereas if space is more important, then Ohlebusch and Sadakane are good choices. All of the data structures evaluated perform within constant factors of each other in terms of query time, space use, and preprocessing time. The choice of data structure may come down to ease of implementation. In terms of simplicity to code, our data structure and Bender are good choices, although this attribute is of course subjective.

5. Directions for Future Work

5.1. Succinctness

The data structure presented in this paper uses $O(n)$ words of space. It is not currently known whether its space can be reduced to $O(n)$ bits if a RMQ returns only the index of the minimum element. As suggested by Nicholson [32], each array C_x and C'_x can be stored using binary rank and select data structures in $O(n)$ bits of space (e.g., [33]). That is, we can support references to C_x and C'_x in constant time using $O(n)$ bits of space per level or $O(n \log\log n)$ total bits. It is not known whether the remaining components of the data structure can be compressed similarly, or whether the space can be reduced further to $O(n)$ bits.

5.2. Higher Dimensions

As shown by Demaine et al. [24], RMQ data structures based on Cartesian trees cannot be generalized to two or higher dimensions. The data structure presented in this paper does not involve Cartesian trees. Although it is possible that some other constraint may preclude generalization to higher dimensions, this remains to be examined.

5.3. Dynamic Data

As described, our data structure structure requires $O(\sqrt{n})$ time per update (e.g., set $A[i] \leftarrow x$) in the worst case. It is not known whether the data structure can be modified to support efficient queries and updates without increasing space.

Acknowledgements

The authors thank Joshua Hernandez who helped write code for some of the data structures used in the experiments described in Section 4. Stephane

Durocher thanks Timothy Chan and Patrick Nicholson with whom this paper's results were discussed. Stephane Durocher also thanks the students of his senior undergraduate class in advanced data structures at the University of Manitoba; preparing lecture material on range searching in arrays inspired him to revisit solutions to the range minimum query problem, leading to the results presented here.

[1] S. Durocher, A simple linear-space data structure for constant-time range minimum query, in: Proceedings of the Conference on Space Efficient Data Structures, Streams and Algorithms, Vol. 8066 of LNCS, Springer, 2013, pp. 48–60.

[2] P. Bose, E. Kranakis, P. Morin, Y. Tang, Approximate range mode and range median queries, in: Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS), Vol. 3404 of Lecture Notes in Computer Science, Springer, 2005, pp. 377–388.

[3] M. A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of the Latin American Theoretical Informatics Symposium (LATIN), Vol. 1776 of Lecture Notes in Computer Science, Springer, 2000, pp. 88–94.

[4] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, Journal of Algorithms 57 (2) (2005) 75–94.

[5] O. Berkman, U. Vishkin, Recursive star-tree parallel data structures, SIAM Journal on Computing 22 (2) (1993) 221–242.

[6] J. Fischer, V. Heun, Theoretical and practical improvements on the RMQ-problem with applications to LCA and LCE, in: Proceedings of the Symposium on Combinatorial Pattern Matching (CPM), Vol. 4009 of Lecture Notes in Computer Science, Springer, 2006, pp. 36–48.

[7] P. Davoodi, R. Raman, S. S. Rao, Succinct representations of binary trees for range minimum queries, in: Proceedings of the International Computing

and Combinatorics Conference (COCOON), Vol. 7434 of Lecture Notes in Computer Science, Springer, 2012, pp. 396–407.

[8] P. v. Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Information Processing Letters 6 (3) (1977) 80–82.

[9] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, R. Sedgewick, Resizable arrays in optimal time and space, in: Proceedings of the Workshop on Algorithms and Data Structures (WADS), Vol. 1663 of Lecture Notes in Computer Science, Springer, 1999, pp. 27–48.

[10] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, B. T. Wilkinson, Linear-space data structures for range mode query in arrays, in: Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), Vol. 14 of Leibniz International Proceedings in Informatics, 2012, pp. 291–301.

[11] D. Krizanc, P. Morin, M. Smid, Range mode and range median queries on lists and trees, Nordic Journal of Computing 12 (2005) 1–17.

[12] S. Alstrup, C. Gavoille, H. Kaplan, T. Rauhe, Nearest commmon ancestors: a survey and a new algorithms for a distributed environment, Theory of Computing Systems 37 (3) (2004) 441–456.

[13] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal on Computing 13 (2) (1984) 338–355.

[14] B. Schieber, U. Vishkin, On finding lowest common ancestors: Simplification and parallelization, SIAM Journal on Computing 17 (6) (1988) 1253–1262.

[15] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE), Vol. 4614 of Lecture Notes in Computer Science, Springer, 2007, pp. 459–470.

[16] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM Journal on Computing 40 (2) (2011) 465–492.

[17] K. Sadakane, Succinct data structures for flexible text retrieval systems, Journal of Discrete Algorithms 5 (2007) 12–22.

[18] P. Davoodi, Data structures: Range queries and space efficiency, Ph.D. thesis, Aarhus University (2011).

[19] J. Fischer, Optimal succinctness for range minimum queries, in: Proceedings of the Latin American Theoretical Informatics Symposium (LATIN), Vol. 6034 of Lecture Notes in Computer Science, Springer, 2010, pp. 158–169.

[20] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradžev, On economical construction of the transitive closure of a directed graph, Soviet Mathematics—Doklady 11 (5) (1970) 1209–1210.

[21] H. Yuan, M. J. Atallah, Data structures for range minimum queries, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 150–160.

[22] G. S. Brodal, P. Davoodi, S. S. Rao, Path minima queries in dynamic weighted trees, in: Proceedings of the Workshop on Algorithms and Data Structures (WADS), Vol. 6844 of Lecture Notes in Computer Science, Springer, 2011, pp. 290–301.

[23] G. S. Brodal, P. Davoodi, S. S. Rao, On space efficient two dimensional range minimum data structures, Algorithmica 63 (4) (2012) 815–830.

[24] E. Demaine, G. M. Landau, O. Weimann, On Cartesian trees and range minimum queries, in: Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP), Vol. 5555 of Lecture Notes in Computer Science, Springer, 2009, pp. 341–353.

[25] M. Golin, J. Iacono, D. Krizanc, R. Raman, S. S. Rao, Encoding 2D range maximum queries, in: Proceedings of the International Symposium on Algorithms and Computation (ISAAC), Vol. 7074 of Lecture Notes in Computer Science, Springer, 2011, pp. 180–189.

[26] A. Amir, J. Fischer, M. Lewenstein, Two-dimensional range minimum queries, in: Proceedings of the Symposium on Combinatorial Pattern Matching (CPM), Vol. 4580 of Lecture Notes in Computer Science, Springer, 2007, pp. 286–294.

[27] D. Pasailă, Range minimum query and lowest common ancestor, http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor.

[28] E. Ohlebusch, S. Gog, A compressed enhanced suffix array supporting fast string matching, in: Proceedings of the String Processing and Information Retrieval Symposium (SPIRE), Vol. 5721 of Lecture Notes in Computer Science, Springer, 2009, pp. 51–62.

[29] S. Gog, Succinct data structure library, http://github.com/simongog/sdsl-lite, retrieved August 2016.

[30] J. Fischer, http://www.bio.ifi.lmu.de/˜fischer, retrieved August 2016.

[31] L. Walsh, http://github.com/leifwalsh/rmq, retrieved August 2016.

[32] P. Nicholson, personal communication (2011).

[33] J. I. Munro, Tables, in: V. Chandru, V. Vinay (Eds.), Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Vol. 1180 of Lecture Notes in Computer Science, Springer, 1996, pp. 37–42.