SPACE-EFFICIENT PREPROCESSING SCHEMES FOR RANGE MINIMUM QUERIES ON STATIC ARRAYS[*]

JOHANNES FISCHER[†] AND VOLKER HEUN[‡]

Abstract. Given a static array of n totally ordered objects, the range minimum query problem is to build a data structure that allows to answer subsequent on-line queries of the form "what is the position of a minimum element in the sub-array ranging from i to j?" efficiently. We focus on two settings, where (1) the input array is available at query time, and (2) the input array is only available at construction time. In setting (1), we show new data structures (a) of size $\frac{2n}{c(n)} - \Theta\big(\frac{n \lg \lg n}{c(n) \lg n}\big)$ bits and query time $O(c(n))$ for any positive integer function $c(n) \in O\big(n^{\varepsilon}\big)$ for an arbitrary constant $0 < \varepsilon < 1$, or (b) with $O(nH_k) + o(n)$ bits and $O(1)$ query time, where H_k denotes the empirical entropy of k'th order of the input array. In setting (2), we give a data structure of size $2n + o(n)$ bits and query time $O(1)$. All data structures can be constructed in linear time and almost in-place.

Key words. range queries, lowest common ancestors, arrays, trees

AMS subject classifications. 05C05, 68R05, 68P05, 68W32

1. Introduction.

For an array $A[1, n]$ of n objects from a totally ordered universe and two indices i and j with $1 \le i \le j \le n$, a *Range Minimum Query*[1] $\text{RMQ}_A(i, j)$ returns the *position* of a minimum element in the sub-array $A[i, j]$; in symbols: $\text{RMQ}_A(i, j) = \text{argmin}_{i \le k \le j}\big\{A[k]\big\}$. Given the ubiquity of arrays and the fundamental nature of this question, it is not surprising that RMQs have a wide range of applications in various fields of computing: text indexing [23, 52], pattern matching [2, 12], string mining [21, 34], text compression [9, 45], document retrieval [42, 53, 59], trees [4, 6, 38], graphs [28, 49], bioinformatics [58], and in other types of range queries [10, 56], to mention just a few.

In almost all applications, the array A on which the RMQs are performed is static and known in advance, and there are several queries to be answered on-line (meaning that the queries are not available from the start). This is also the scenario considered in this article, and in such a case it makes sense to preprocess A into a (preprocessing-) *scheme* such that future RMQs can be answered quickly. We can hence formulate the following problem.

Definition 1.1 (RMQ-Problem).
Given: *a static array $A[1, n]$ of n totally ordered objects.*
Compute: *an (ideally small) data structure, called* scheme*, that allows to compute subsequent RMQs on A (in ideally constant time).*

The most naive preprocessing would be to store the answers to all $\binom{n}{2}$ proper RMQs in a table, and then simply look up the answers in (optimal) constant time. On the opposite side of the extremes, we could do *no* preprocessing at all, and scan the

[*]Parts of this work have already been presented at the 17th Annual Symposium on Combinatorial Pattern Matching [19], at the 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies [20], at the 2008 Data Compression Conference [22], and at the 9th Latin American Theoretical Informatics Symposium [17]. This work has been partially supported by the German Research Foundation (DFG).

[†]Institut für Theoretische Informatik, Karlsruher Institut für Technologie, Am Fasanengarten 5, 76131 Karlsruhe, Germany (johannes.fischer@kit.edu).

[‡]Institut für Informatik, Ludwig-Maximilians-Universität München, Amalienstr. 17, 80333 München (volker.heun@bio.ifi.lmu.de).

[1]Sometimes also called *Discrete Range Searching* [1] or, depending on the context, *Range Maximum Query.*

query interval $A[i, j]$ each time a new query $\text{RMQ}_A(i, j)$ arrives, resulting in $O(n)$ query time in the worst case. Both of these solutions are clearly far from being optimal, and indeed, it was noted already a quarter of a century ago [25] that a scheme of size $O(n)$ *words* suffices to answer RMQs in optimal constant time. This scheme is based on the idea that an RMQ-instance can be transformed into an instance of *lowest common ancestors* (LCAs) in the *Cartesian Tree* [60] of A (see § 2.2 for a formal definition of this tree). For constant-time LCA-queries, linear preprocessing schemes had already been discovered earlier [33].

The problem of Gabow et al.'s solution [25], and also that of subsequent simplifications [1, 4, 6, 61], is their space consumption of $O(n \lg n)$ *bits*, as they store $O(n)$ words occupying $\lceil \lg n \rceil$ bits each.[2] A recent trend in the theory of data structures is that of *succinct* and *compressed* data structures. A *succinct data structure* uses space that is close to the information-theoretic lower bound, in the sense that objects from a universe of cardinality L are stored in $(1 + o(1)) \lg L$ bits. An even stronger concept is that of *compressed data structures*, where it is tried to *surpass* the information-theoretic lower bound for instances that are in some sense *compressible.* Research on succinct and compressed data structures is very active, and we just mention some examples from the realm of trees [5, 13, 27, 36, 41, 55], dictionaries [46, 50], and strings [14, 15, 31, 32, 43, 51, 54], being well aware of the fact that this list is far from complete.

Our results for RMQs lie in the field of succinct and compressed data structures. We work with the standard word-RAM model of computation (which is also the model used in all LCA- and RMQ-schemes cited in this article), where it is assumed that we can do arithmetic and logical operations on w-bit wide words in $O(1)$ time, and $w = \Omega(\lg n)$. Before detailing our contributions, we first classify and summarize existing schemes for RMQs in constant time (called $O(1)$-RMQs henceforth).

1.1. Previous Solutions for RMQ. In accordance with common nomenclature [26], preprocessing schemes for $O(1)$-RMQs can be classified into two different types: *systematic* and *non-systematic* (also called *indexing* and *encoding* data structures, respectively). Systematic schemes must store the input array A verbatim along with the additional information for answering the queries. Systematic schemes are perhaps more natural than non-systematic ones, and not surprisingly, all early schemes [1, 4, 6, 25] are systematic. They are appropriate in the following situations:

1. If $|A|$, the number of bits to store A, is small enough to be dominated by the space for the RMQ-scheme (e.g., $|A| = O(n)$).
2. If $\omega(1)$ query time suffices, and whole blocks of the input array are to be *scanned* when answering the queries.
3. Perhaps most importantly, when the actual *values* of the minima matter, or if A is needed for different purposes.

In any of the situations mentioned above, some space for the scheme can in principle be saved, as the query algorithm can substitute "missing information" by consulting A when answering the queries; this is indeed what all systematic schemes make heavy use of.

On the contrary, non-systematic schemes must be able to obtain their final answer without consulting the array. This second type is also important, for at least the following two reasons:

[2]Throughout this article, space is measured in bits, lg denotes the binary logarithm, and $\lg^x n$ is short for $(\lg n)^x$.

Table 1.1

Preprocessing schemes for $O(1)$-RMQs, where $|A|$ denotes the space of the (read-only) input array A. (Set $c(n) = O(1)$ to obtain $O(1)$ query time in Thm. 3.7.) Space is measured in bits, and construction space is in addition *to the final space. In Thm. 4.1, σ denotes the number of* different *objects in A, and H_k the empirical entropy of A. Schemes marked $^{(*)}$ are only for a restricted class of arrays, where subsequent elements differ by exactly 1. Schemes marked $^{(**)}$ are non-systematic, meaning that they do not have to access A for answering the queries. All schemes can be constructed in $O(n)$ time.*

references	final space	construction space
[25] + [33], [4,6,61]	$\lvert A\rvert + O(n \lg n)$	$O(n \lg n)$
[1]	$\lvert A\rvert + O(n \lg n)$	$O(\text{polylg}\, n)$
Thm. 3.7	$\boldsymbol{\lvert A\rvert + \frac{2n}{c(n)} - \Theta\big(\frac{n \lg\lg n}{c(n) \lg n}\big)}$	$\boldsymbol{O(\lg^3 n)}$
Thm. 4.1	$\boldsymbol{\lvert A\rvert + nH_k + O\big(\frac{n(k \lg \sigma + \lg\lg n)}{\lg n}\big)}$	$\boldsymbol{O\big(\sqrt{n/\lg n}\big)}$
[52]$^{(*)}$	$n + O(n \lg^2 \lg n / \lg n)$	$O(1)$
[55]$^{(*)}$	$n + O(n/\text{polylg}\, n)$	$O(n)$
Thm. 5.7$^{(*)}$	$\boldsymbol{n + O(n \lg\lg n / \lg n)}$	$\boldsymbol{O(\lg^3 n)}$
[53]$^{(**)}$	$4n + O(n \lg^2 \lg n / \lg n)$	$\lvert A\rvert + O(n \lg n)$
Thm. 5.8$^{(**)}$	$\boldsymbol{2n + O(n \lg\lg n / \lg n)}$	$\boldsymbol{\lvert A\rvert + n + O\big(\frac{n \lg\lg n}{\lg n}\big)}$
Cor. 5.9$^{(**)}$	$\boldsymbol{2n + O(n/\text{polylg}\, n)}$	$\boldsymbol{\lvert A\rvert + O(n)}$

1. In some applications, e.g., in algorithms for document retrieval [42, 53] or position restricted substring matching [12], only the *position* of the minimum matters, but *not* the value of this minimum. In such cases it would be a waste of space to keep the input array in memory.

2. If the time to access the elements in A is $\omega(1)$, this slowed-down access time propagates to the time for answering RMQs if the query algorithm consults the input array, which may be undesirable. As a prominent example, in string processing RMQ is often used on the array of *longest common prefixes* of lexicographically consecutive suffixes, the so-called *LCP-array* [39]. However, storing the LCP-array efficiently in $2n + o(n)$ bits [52] or even less [18, 23] increases the access-time to the time needed to retrieve an entry from the corresponding *suffix array* [39], which is $\Omega(\lg^\varepsilon n)$ (constant $\varepsilon > 0$) at the very best if the suffix array is also stored in compressed form [31, 51]. Hence, with a systematic scheme the time needed for answering RMQs on LCP could never be $O(1)$ in this case. But exactly this would be needed for constant-time navigation in RMQ-based compressed suffix trees [23, 44], where for the occasional retrieval of the string-depth of nodes the LCP-array is still needed (so this is not the same as the above point).

In the following, we briefly sketch previous solutions for RMQ schemes. For a summary, see Tbl. 1.1, where, besides the final space consumption, in the third column we list the additional space needed for constructing the scheme.

1.1.1. Systematic Schemes. Most systematic schemes are based on the Cartesian Tree [60], the only exception being the schemes due to Alstrup et al. [1] and Yuan and Atallah [61]. All schemes are based on the idea of splitting the query range into several sub-queries, all of which have been precomputed, and then returning the overall minimum as the final result. The schemes from the first four rows of Tbl. 1.1 have the same final space guarantees (namely $O(n \lg n)$ bits), with Bender et al.'s scheme [4] being less complex than the previous ones, and Alstrup et al.'s [1] being even simpler (and most practical due to its small construction space). Yuan and Atallah's scheme is noteworthy because it does not involve overlapping queries, a fact that may be useful for related kinds of range queries. Recently, Brodal et al. [8] proved a *lower bound* for systematic schemes: any scheme that uses $O(n/C)$ bits in addition to A must have $\Omega(C)$ query time.[3]

An important special case are schemes for ± 1RMQ [52, 55], where it is assumed that subsequent array-elements differ by exactly 1; we will describe them in greater detail in § 2.5.

1.1.2. Non-Systematic Schemes. The only existing scheme is due to Sadakane [53] and uses $4n + o(n)$ bits. It is based on the balanced parentheses sequence (BPS) [41] of the Cartesian Tree of the input array A, and a $o(n)$-bit scheme for $O(1)$-LCA computation therein [52]. The difficulty that Sadakane overcomes is that in the "original" Cartesian Tree, there is no natural mapping between array-indices in A and positions of parentheses (because there is no way to distinguish between left and right nodes in the BPS of a tree); this is why building the BPS directly on the Cartesian Tree does not achieve $O(1)$-RMQs. Therefore, Sadakane introduces n "fake" leaves to get such a mapping. There are two main drawbacks of this solution.

1. Due to the introduction of the "fake" leaves, it does not achieve the *information-theoretic lower bound* (for non-systematic schemes) of $2n - \Theta(\lg n)$ bits. This lower bound is easy to see because any scheme for RMQs allows to reconstruct the Cartesian Tree by iteratively querying the scheme for the minimum (in analogy to the definition of the Cartesian Tree; see § 2.2). And because the Cartesian Tree is binary and each binary tree is a Cartesian Tree for some input array, any scheme must use at least $\lg\big(\binom{2n}{n}/(n+1)\big) = 2n - \Theta(\lg n)$ bits [35].
2. For getting an $O(n)$-time construction algorithm, the (modified) Cartesian Tree needs to be first constructed in a pointer-based implementation, and then converted to the space-saving BPS. This leads to a *construction space requirement* of $O(n \lg n)$ bits, as each node occupies $O(\lg n)$ bits in memory. The problem why the BPS cannot be constructed directly in $O(n)$ time (at least we are not aware of such an algorithm) is that a "local" change in A (be it only appending a new element at the end) does not necessarily lead to a "local" change in the tree; this is also the intuitive reason why maintaining dynamic Cartesian Trees is difficult [7].

1.2. Our Contributions.

1.2.1. Our Results. We present preprocessing schemes for range minimum queries of yet unseen small size (in fact optimal for their respective model); see again Tbl. 1.1 for a summary and comparison.

In the systematic setting, we first give a simple scheme that uses only $\frac{2n}{c(n)} - \Theta\big(\frac{n \lg\lg n}{c(n)\lg n}\big)$ bits on top of A and has $O(c(n))$ query time, assuming that the elements in A can be read in constant time (Thm. 3.7). Here, $c(n)$ can be any positive integer function bounded by $O\big(n^{\varepsilon}\big)$ for an arbitrary constant $0 < \varepsilon < 1$. If $c(n) = O(1)$, then Thm. 3.7 gives optimal constant query time with $O(n)$ space, where the big-Oh constant can be made arbitrarily small. This is the first systematic scheme with linear bit-complexity. Moreover, this space-time tradeoff is *optimal* [8].

We then show in Thm. 4.1 how to compress the scheme from Thm. 3.7 into a data structure of size $nH_k + O\big(\frac{n}{\lg n}(k \lg\sigma + \lg\lg n)\big) + |A|$ bits, simultaneously over all k. Here, H_k denotes the empirical entropy of order k [40] of the input array A, and σ

[3]The claimed lower bound of $2n + o(n) + |A|$ bits under the "min-probe-model" [20] turned out to be wrong, as was kindly pointed out to the authors by S. Srinivasa Rao (personal communication, November 2007).

denotes the number of *distinct* elements in A. The value nH_k is a common measure for the compressibility of data structures [43], as it provides a lower bound on the size of the output of any compressor that encodes a symbol based on the k preceding characters.

We also give a scheme for ± 1RMQ that needs only $O\big(\frac{n \lg\lg n}{\lg n}\big)$ bits on top of the n bits for storing the input array (Thm. 5.7), as opposed to $O\big(\frac{n \lg^2 \lg n}{\lg n}\big)$ bits needed by Sadakane's solution [52]. Although this result has already been superseded [55] after the initial submission of the present material, we chose to include it in this article due to its low construction space, as as opposed to $O(n)$ construction space (with a possibly large big-Oh constant) for Sadakane and Navarro [55].

In fact, we put a particular emphasis on construction *space*, as it is an important issue and often limits the practicality of a data structure, especially for large inputs (as they arise nowadays in web-page-analysis or computational biology). The schemes from Thm. 3.7, 4.1, and 5.7 can be constructed in-place (apart from negligibly small terms).

We finally focus on the non-systematic setting, where we show a preprocessing scheme of asymptotically *optimal* size $2n + O\big(\frac{n \lg\lg n}{\lg n}\big)$ bits and $O(1)$ query time (Thm. 5.8). To construct this scheme, we need only one additional bit-vector of length n, plus some sub-linear structures. This should be directly compared to the only existing non-systematic solution [53] with $4n + O\big(\frac{n \lg^2 \lg n}{\lg n}\big)$ bits, which needs $O(n \lg n)$ bits of construction space. Hence, Thm. 5.8 not only lowers the final space to optimal, but also the construction space to $O(n)$. This is a significant improvement over the $O(n \lg n)$-bit construction algorithm for Sadakane's non-systematic scheme [53]. Note again that the space for storing A is not necessarily $\Theta(n \lg n)$; for example, if the numbers in A are integers in the range $\big[1, \lg^{O(1)} n\big]$, A can be stored as an array of packed words using only $O(n \lg\lg n)$ bits of space. In such a case, a construction space of $O(n \lg n)$ bits would *dominate* the space for the input array A and thus constitute a severe memory bottleneck — a situation that is avoided only with our new $O(n)$-bit construction algorithm.

If construction space is not an issue, we can state an improved version of Thm. 5.8 by using recent results on succinct data structures *out of the box* and achieve $2n + O(n/\operatorname{polylg} n)$ bits (Cor. 5.9).

Construction *time* is linear for all our methods.

1.2.2. Technical Contributions. Our schemes from Thm. 3.7 and Thm. 4.1 are based on a novel variant of the Four-Russians-Trick [3]. At a high level, this trick decomposes a problem into smaller sub-problems, precomputes in a table all possible answers to these sub-problems, and then solves the larger problem by looking up (possibly many) answers to its sub-problems. This table lookup is usually done with a small portion of the input, where it is assumed that this portion fits into a computer word and can hence be treated as an integer. This makes implicit assumptions on the representation of the input data (among others, they need to be stored contiguously, for otherwise they cannot be read in constant time). We, on the other hand, proceed by *attaching* new information to the input, regardless of how it is stored. This new information is constructed from Cartesian Trees of small blocks of the input array, but unlike in previous schemes we do not use these Cartesian Trees *directly* for computing RMQs, but only *implicitly* when doing table lookups. In fact, we do not store the Cartesian Trees themselves, but only their indices in an enumeration of all Cartesian Trees. Most of the work in § 3 will therefore be devoted to develop such a fast and space-efficient enumeration. We remark here that only the analysis of that enumeration is complex, while the resulting algorithm is notably simple!

For Thm. 5.8 and Cor. 5.9, we introduce our second technical novelty, the *2-dimensional Min-Heap*. This is a tree which is, like the Cartesian Tree, inherently connected to RMQs. While it can be viewed as a "twisted" variant of the Cartesian Tree, we prove that it has better properties than the latter in many regards, e.g., a better mapping of array indices to nodes, more space-efficient constructability, etc. We mention at this point that the 2-dimensional Min-Heap has also applications in Sadakane and Navarro's succinct tree representation [55], who call it lrm-tree (for *left-to-right minima*).

2. Preliminaries.

2.1. Basic Conventions.

We use the notation $A[1, n]$ to indicate that A is an array of n objects, indexed from 1 through n. $A[i, j]$ denotes A's sub-array ranging from i to j for $1 \leq i \leq j \leq n$. For integers $\ell \leq r$, $[\ell : r]$ denotes the set $\{\ell, \ell+1, \ldots, r\}$.

Following Bender et al.'s notation [4], we say that a scheme with preprocessing time $p(n)$ and query time $q(n)$ has *time*-complexity $\langle p(n), q(n)\rangle$. We extend this notation to cover space by writing $[\![s(n), t(n)]\!]$ if $t(n)$ is the final space of the data structure, and $s(n)$ is the additional space at construction time.

When analyzing space, for the sake of clarity we write $O(m \cdot \lg(g(m)))$ for the number of bits needed to store a table of m positive integers from a range of size $(g(m))^{O(1)}$.

2.2. Cartesian Trees.

The following definition [60] is central for all RMQ-algorithms (here and in the following, "binary" refers to trees with nodes having *at most* two children, and not *exactly* two).

DEFINITION 2.1. *A* Cartesian Tree *of an array $A[\ell, r]$ is a rooted binary tree $\mathcal{C}(A[\ell, r])$, consisting of a root v that is labeled with the position i of a minimum in $A[\ell, r]$, and at most two subtrees connected to v. The left child of v is the root of the Cartesian Tree of $A[\ell, i-1]$ if $i > \ell$, otherwise v has no left child. The right child of v is defined analogously for $A[i+1, r]$.*

The tree $\mathcal{C}(A)$ is not necessarily unique if A contains equal elements. To overcome this problem, we impose a *strong* total order "$\prec$" on A by defining $A[i] \prec A[j]$ iff $A[i] < A[j]$, or $A[i] = A[j]$ and $i < j$. The effect of this definition is just to consider the "first" occurrence of equal elements in A as being the "smallest." Defining a Cartesian Tree over A using the $\prec$-order gives a *unique* tree that we call the *Canonical Cartesian Tree*. It is denoted by $\mathcal{C}^{\mathrm{can}}(A)$. Note also that this order results in unique answers to RMQs, because the minimum is unique.

Gabow et al. [25] give an algorithm for constructing $\mathcal{C}^{\mathrm{can}}(A)$ incrementally, which we summarize as follows. Let $\mathcal{C}^{\mathrm{can}}_i(A)$ be the Canonical Cartesian Tree for $A[1, i]$. Then $\mathcal{C}^{\mathrm{can}}_{i+1}(A)$ is obtained by climbing up from the rightmost node of $\mathcal{C}^{\mathrm{can}}_i(A)$ to the root, thereby finding the position where $A[i+1]$ belongs. To be precise, let $v_1, \ldots, v_k$ be the nodes on the rightmost path in $\mathcal{C}^{\mathrm{can}}_i(A)$ with labels $\lambda_1, \ldots, \lambda_k$, respectively, where v_1 is the root, and v_k is the rightmost node. Let m be defined such that $A[\lambda_m] \leq A[i+1]$ and $A[\lambda_{m+1}] > A[i+1]$ (hence $A[\lambda_{m'}] > A[i+1]$ for all $m < m' \leq k$). To build $\mathcal{C}^{\mathrm{can}}_{i+1}(A)$, create a new node w with label $i+1$ that becomes the right child of v_m, and the subtree rooted at v_{m+1} becomes the left child of w. This process inserts each element to the rightmost path exactly once, and each comparison removes one element from the rightmost path, resulting in an amortized $O(n)$ construction time to build $\mathcal{C}^{\mathrm{can}}(A)$.

The labels in a Cartesian Tree are often omitted, as they correspond to the inorder-numbers of the nodes and can hence be deduced from the tree topology.

2.3. Rank and Select on Binary Strings. Consider a *bit-string* $S[1,n]$ of length n. We define the fundamental *rank*- and *select*-operations on S as follows: $\mathit{rank}_1(S,i)$ gives the number of 1's in the prefix $S[1,i]$, and $\mathit{select}_1(S,i)$ gives the position of the i'th 1 in S, reading S from left to right ($1 \leq i \leq n$). Operations $\mathit{rank}_0(S,i)$ and $\mathit{select}_0(S,i)$ are defined analogously for 0-bits. There are data structures of size $O\big(\frac{n \lg \lg n}{\lg n}\big)$ bits in addition to S that support rank- and select-operations in $O(1)$ time [29]. These data structures can be constructed in-place and in linear time.

There are also smaller data structures for rank- and select: Pătraşcu [48] showed that $O(n/\lg^\gamma n)$ bits on top of S suffice to achieve $O(\gamma)$ query time, for an arbitrary $\gamma > 0$. However, this comes at the price of an increased construction space of $O(n)$ bits [55].

2.4. Sequences of Balanced Parentheses. A string $B[1,2n]$ of n opening parentheses '(' and n closing parentheses ')' is called *balanced* if in each prefix $B[1,i]$, $1 \leq i \leq 2n$, the number of ')'s is no more than the number of '('s. Operation *findopen*(B,i) returns the position j of the "matching" opening parenthesis for the closing parenthesis at position i in B. This position j is defined as the largest $j < i$ for which $\mathit{rank}_((B,i) - \mathit{rank}_)(B,i) = \mathit{rank}_((B,j) - \mathit{rank}_)(B,j)$. The *findopen*-operation can be computed in constant time [41]; the most space-efficient data structure for this needs $O\big(\frac{n \lg \lg n}{\lg n}\big)$ bits on top of B [27] and can be constructed in linear time, using $O\big(\frac{n \lg \lg n}{\lg n}\big)$ bits of working space [27, Remark 9].

Very recently, Sadakane and Navarro [55] showed that the findopen-operation can be implemented to run in $O(\gamma)$ time, by building a *range min-max tree* on top of B, using $O(n/\lg^\gamma n)$ bits. The disadvantage is again an increased construction space of $O(n)$ bits.

2.5. Data Structures for ±1RMQ. Consider an array $E[1,n]$ of natural numbers, where the difference between consecutive elements in E is either $+1$ or -1 (i.e., $E[i]-E[i-1] = \pm 1$ for all $1 < i \leq n$). Such an array E can be encoded as a bit-vector $S[1,n]$, where $S[1] = 0$, and for $i > 1$, $S[i] = 1$ iff $E[i] - E[i-1] = +1$. Then $E[i]$ can be obtained by $E[1] + \mathit{rank}_1(S,i) - \mathit{rank}_0(S,i) + 1 = E[1] + i - 2\mathit{rank}_0(S,i) + 1$. Under this setting, Sadakane [52] shows how to support RMQs on E in $O(1)$ time, using S and additional structures of size $O\big(\frac{n \lg^2 \lg n}{\lg n}\big)$ bits. We denote this restricted version of RMQ by ±1RMQ.

The range min-max tree from § 2.4, using $O(n/\lg^\gamma n)$ bits of final space and $O(n)$ bits of construction space, can also answer ±1-RMQs in $O(\gamma)$ time.

2.6. Depth-First Unary Degree Encoding of Ordered Trees. The Depth-First Unary Degree Sequence (DFUDS) U of an ordered tree T is defined as follows [5]. If T is a leaf, U is given by '()'. Otherwise, if the root of T has w subtrees $T_1,\ldots,T_w$ in this order, U is given by the juxtaposition of $w+1$ '('s, a ')', and the DFUDS's of $T_1,\ldots,T_w$ in this order, with the first '(' of each T_i being omitted. It is easy to see that the resulting sequence is balanced, and that it can be interpreted as a preorder-listing of T's nodes, where, ignoring the very first '(', a node with w children is encoded in *unary* as '$(^w)$' (hence the name DFUDS). Most navigational operations on trees can be simulated by *rank*, *select*, *findopen* and ±1RMQ-operations, in particular moving to the parent node [5], and finding the *lowest common ancestor* LCA(u,v) of two nodes

u and v [36], which is defined as the deepest node in T that is an ancestor of both u and v.

3. Preprocessing in the Systematic Setting. We now come to the description of the first contribution of this article: a direct and practicable representation of RMQ-information in the systematic setting.

3.1. Overview. The array $A[1,n]$ to be preprocessed is (conceptually) divided into blocks $B_1,\ldots,B_{\lceil n/s\rceil}$ of size $s=\left\lceil\frac{\lg n}{4}\right\rceil$, where $B_i=A\big[(i-1)s+1,is\big]$.[4] The idea is that a general query from ℓ to r can be divided into at most three sub-queries: one *out-of-block-query* that spans several blocks, and two *in-block-queries* (queries completely contained within a block) to the left and right of the out-of-block-query. The overall answer to the range minimum query is obtained by taking the minimum of these three sub-queries. See also the top half of Fig. 3.4 on p. 14, where the in-block-queries are labeled by ① and ③, and the out-of-block-query by ②. Note that if ℓ and r are in the same block, then there is only one in-block-query to be answered.

The overall appearance of our solution is similar to previous systematic schemes (dividing the array into several blocks); the main novelty lies in answering the in-block-queries, which we handle in § 3.2 with a novel variant of the Four-Russians-Trick [3] (precomputation of all answers for sufficiently small instances). However, also our solution to the long queries (§ 3.3) differs from earlier approaches, resulting in a smaller lower order term.

3.2. Preprocessing for In-Block-Queries. We first show how to store all necessary information for answering in-block-queries. The key to our solution is the following lemma, which has implicitly been used already in all previous schemes.

LEMMA 3.1. *Let B_x and B_y be two blocks of size s. Then $\textsc{rmq}_{B_x}(i,j)=\textsc{rmq}_{B_y}(i,j)$ for all $1\le i\le j\le s$ if and only if $\mathcal{C}^{\mathrm{can}}(B_x)=\mathcal{C}^{\mathrm{can}}(B_y)$.*

Proof. It is easy to see that $\textsc{rmq}_{B_x}(i,j)=\textsc{rmq}_{B_y}(i,j)$ for all $1\le i\le j\le s$ if and only if the following three conditions are satisfied:

1. The minimum under "$\prec$" occurs at the same position m in both arrays, i.e., $\operatorname{argmin} B_x=\operatorname{argmin} B_y=m$.
2. For all i',j' with $1\le i'\le j'<m$: $\textsc{rmq}_{B_x[1,m-1]}(i',j')=\textsc{rmq}_{B_y[1,m-1]}(i',j')$.
3. For all i',j' with $m<i'\le j'\le s$: $\textsc{rmq}_{B_x[m+1,s]}(i',j')=\textsc{rmq}_{B_y[m+1,s]}(i',j')$.

Due to the definition of the Canonical Cartesian Tree, points (1)–(3) are true if and only if the root of $\mathcal{C}^{\mathrm{can}}(B_x)$ equals the root of $\mathcal{C}^{\mathrm{can}}(B_y)$, and $\mathcal{C}^{\mathrm{can}}\big(B_x[1,m-1]\big)=\mathcal{C}^{\mathrm{can}}\big(B_y[1,m-1]\big)$, and $\mathcal{C}^{\mathrm{can}}\big(B_x[m+1,s]\big)=\mathcal{C}^{\mathrm{can}}\big(B_y[m+1,s]\big)$. As this is the definition of Cartesian Trees, this is true iff $\mathcal{C}^{\mathrm{can}}(B_x)=\mathcal{C}^{\mathrm{can}}(B_y)$. $\square$

The advantage of this is that we do not have to store the answers to in-block-queries for all $\lceil n/s\rceil$ *occurring* blocks, but only for C_s *possible* blocks, where $C_s=\frac{1}{s+1}\binom{2s}{s}$ is the s'th Catalan Number (number of rooted trees on s nodes). So if we have a table $P[1,C_s][1,s][1,s]$ that stores the answers to all RMQs inside of all C_s possible size-s blocks, knowing the type $t(B_x)$ of block B_x will allow us to look-up the answer to $\textsc{rmq}_{B_x}(i,j)$ as $P\big[t(B_x)\big][i][j]$. Here, by the *type* of B_x we mean a description of B_x's Canonical Cartesian Tree $\mathcal{C}^{\mathrm{can}}(B_x)$ that identifies it among all Cartesian Trees on s elements.

It thus remains to show how to compute the types of the $\lceil n/s\rceil$ blocks in A in linear time; i.e., how to fill an array $T\big[1,\lceil n/s\rceil\big]$ such that $T[x]=t(B_x)$ is the type of

[4]In fact, any block size $s=\left\lceil\frac{\lg n}{2+\delta}\right\rceil$ for an arbitrary constant $\delta>0$ would suffice, but we use $\delta=2$ for simplicity.

Algorithm 1: An algorithm to compute the type of a block B_x

```
Input: a block $B_x$ of size $s$
Output: the type of $B_x$, as defined by Eq. (3.1)
1  Let $R$ be an array of size $s+1$        {$R$ stores elements on the rightmost path}
2  $R[1] \leftarrow -\infty$
3  $q \leftarrow s, N \leftarrow 0$
4  for $i \leftarrow 1, \dots, s$ do
5      while $R[q+i-s] > B_x[i]$ do
6          $N \leftarrow N + C_{(s-i)q}$                  {add number of skipped paths}
7          $q \leftarrow q-1$                       {remove node from rightmost path}
8      end
9      $R[q+i+1-s] \leftarrow B_x[i]$                   {$B_x[i]$ is new rightmost node}
10 end
11 return $N$
```

block B_x. Lemma 3.1 implies that there are only C_s different types of blocks, so we are looking for a surjection

$$t : \mathcal{A}_s \to [0 : C_s - 1]\,,\ \text{and}\ t(B_x) = t(B_y)\ \text{iff}\ \mathcal{C}^{\text{can}}(B_x) = \mathcal{C}^{\text{can}}(B_y)\ , \tag{3.1}$$

where $\mathcal{A}_s$ is the set of arrays of size s.[5] We claim that Alg. 1 computes a function satisfying (3.1) in $O(s)$ time. It makes use of the so-called *Ballot Numbers* C_{pq} [37], defined by

$$C_{00} = 1, C_{pq} = C_{p(q-1)} + C_{(p-1)q},\ \text{if}\ 0 \le p \le q \ne 0,\ \text{and}\ C_{pq} = 0\ \text{otherwise.} \tag{3.2}$$

(Refer to the second paragraph of § 3.2.1 for a graph-theoretic interpretation of C_{pq}.) It can be proved that a closed formula for C_{pq} is given by $\frac{q-p+1}{q+1}\binom{p+q}{p}$ [37], which immediately implies that C_{ss} equals the s'th Catalan number C_s.

LEMMA 3.2. *Algorithm 1 correctly computes the type of a block of size s in $O(s)$ time, i.e., it computes a function satisfying the conditions given in* (3.1).

Proof. Intuitively, Alg. 1 simulates the algorithm for constructing $\mathcal{C}^{\text{can}}(B_x)$ given in § 2.2. Array $R[1, s+1]$ simulates the stack containing the labels of the nodes on the rightmost path of the partial Canonical Cartesian Tree $\mathcal{C}^{\text{can}}_i(B_x)$, with $q+i-s$ pointing to the top of the stack (i.e., the rightmost node), and $R[1]$ acting as a "stopper." If ℓ_i denotes the number of times the while-loop (lines 5–8) is executed during the ith iteration of the outer for-loop, then ℓ_i equals the number of elements that are removed from the rightmost path when going from $\mathcal{C}^{\text{can}}_{i-1}(B_x)$ to $\mathcal{C}^{\text{can}}_i(B_x)$. Because one cannot remove more elements from this rightmost path than one has inserted before, the sequence $\ell_1\ell_2\dots\ell_s$ satisfies

$$0 \le \sum_{k=1}^{i} \ell_k < i\ \text{for all}\ 1 \le i \le s\ . \tag{3.3}$$

[5]Note that if we did not require the function be surjective then a simple $2s$-bit encoding of $\mathcal{C}^{\text{can}}(B_x)$ would suffice as the type of B_x (e.g., list the tree nodes level by level, writing ij for a node with i (or j) left (or right) children for $i, j \in \{0, 1\}$); surjectivity, however, ensures the best possible space for array T.

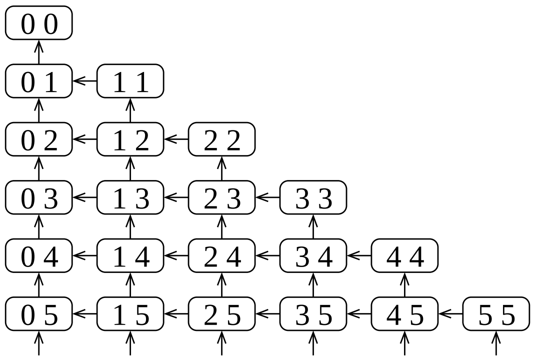


FIG. 3.1. *The infinite graph arising from the definition of the Ballot Numbers. Its vertices are* $\boxed{p\ q}$ *for all* $0 \le p \le q$. *There is an edge from* $\boxed{p\ q}$ *to* $\boxed{(p-1)\ q}$ *if* $p > 0$ *and to* $\boxed{p\ (q-1)}$ *if* $q > p$.

Let $\mathcal{L}_s$ denote the set of sequences of positive integers $\ell_1, \ldots, \ell_s$ satisfying (3.3). Because for every $l \in \mathcal{L}_s$ there is an array $B \in \mathcal{A}_s$ such that Alg. 1, run on B, produces l, we established a surjection from $\mathcal{A}_s$ to $\mathcal{L}_s$. It thus remains to prove that the additions performed in line 6 of Alg. 1 yield a unique index in an enumeration of $\mathcal{L}_s$. These additions are captured by function f

$$f : \mathcal{L}_s \to [0 : C_s - 1] : \ell_1\ell_2 \ldots \ell_s \mapsto \sum_{i=1}^{s} \sum_{0 \le j < \ell_i} C_{(s-i)\left(s-j-\sum_{k<i} \ell_k\right)} , \tag{3.4}$$

and we shall prove in the following section that f is bijective. Hence f, the function computed by Alg. 1, satisfies Eq. (3.1). Because all numbers arising in Alg. 1 are at most n for $s = \lceil \frac{\lg n}{4} \rceil$, they can be added in $O(1)$ time; the claim on linear running time follows. $\square$

3.2.1. Bijectivity of *f*. Throughout this section, the reader is encouraged to peek at Tbl. 3.1, where most of the concepts are illustrated. Remember that $\mathcal{L}_s$ denotes the set of sequences $\ell_1\ell_2 \ldots \ell_s$ satisfying (3.3).

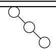
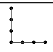
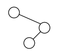
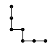
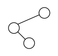
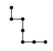
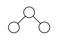
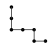
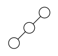
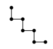
Recall the definition of the Ballot Numbers (3.2) and look at the infinite directed graph shown in Fig. 3.1: C_{pq} equals the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$, because of (3.2): if the current vertex is $\boxed{p\ q}$, one can either first go "up" and then take any of the $C_{p(q-1)}$ paths from $\boxed{p\ (q-1)}$ to $\boxed{0\ 0}$; or one first goes "left" to $\boxed{(p-1)\ q}$ and afterwards takes any of the $C_{(p-1)q}$ paths to $\boxed{0\ 0}$.

Any sequence $\ell_1 \ldots \ell_s$ corresponds to a path from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in Fig. 3.1 (and vice versa). This is because the graph is constructed in a way such that one cannot move more cells upwards than one has already gone to the left if one starts at $\boxed{s\ s}$. So the path corresponding to $\ell_1 \ldots \ell_s$ is obtained as follows: in step i, go ℓ_i steps upwards and one step to the left, and after step s go upwards until reaching $\boxed{0\ 0}$.

From the discussion above we already know how we can bijectively map the sequences in $\mathcal{L}_s$ to paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in the graph in Fig. 3.1. Calling the set of such paths $\mathcal{P}_s$, we thus have to show that f is a bijection from $\mathcal{P}_s$ to $\left[0 : C_s - 1\right]$,

TABLE 3.1

Example-arrays of length 3, their Cartesian Trees, and their corresponding paths in the graph in Fig. 3.1. The last column shows how to calculate the index of $\mathcal{C}^{can}(A)$ in an enumeration of all Cartesian Trees.

array A	$\mathcal{C}^{\mathrm{can}}(A)$	path	$\ell_1\ell_2\ell_3$	number in enumeration
123			000	0
132			001	$C_{03} = 1$
231			002	$C_{03} + C_{02} = 2$
213			010	$C_{13} = 3$
321			011	$C_{13} + C_{02} = 4$

with the intended meaning that the paths in $\mathcal{P}_s$ should actually be first mapped bijectively to a sequence in $\mathcal{L}_s$.

We need the following identities on the Ballot Numbers:

$$C_{pq} = \sum_{p \le q' \le q} C_{(p-1)q'} \text{ for } 1 \le p \le q \tag{3.5}$$

$$C_{(p-1)p} = 1 + \sum_{0 \le i < p-1} C_{(p-i-2)(p-i)} \text{ for } p > 0 \tag{3.6}$$

Eq. (3.5) follows easily by "unfolding" the definition of the Ballot Numbers, but before proving it formally, let us first see how this formula can be interpreted in terms of *paths*. It actually says that the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$ can be obtained by summing over the number of paths to $\boxed{0\ 0}$ from $\boxed{(p-1)\ q}$, $\boxed{(p-1)\ (q-1)}$, ..., $\boxed{(p-1)\ p}$, as all paths starting at $\boxed{p\ q}$ can be expressed as the disjoint union over those paths. The formal proof of (3.5) is by induction on q: for $q = 1$, $C_{11} = C_{10} + C_{01} = 0 + 1 = 1$ by (3.2), and (3.5) gives $C_{11} = \sum_{1 \le q' \le 1} C_{0q'} = C_{01} = 1$. For the induction step, let the inductive hypothesis (IH) be $C_{p(q-1)} = \sum_{p \le q' \le q-1} C_{(p-1)q'}$ for all $1 \le p \le q-1$. Then

$$C_{pq} \overset{(3.2)}{=} C_{p(q-1)} + C_{(p-1)q} \overset{(\mathrm{IH})}{=} \sum_{p \le q' \le q-1} C_{(p-1)q'} + C_{(p-1)q} = \sum_{p \le q' \le q} C_{(p-1)q'} \ .$$

Eq. (3.6) is only slightly more complicated and can be proved by induction on p: for $p = 1$, $C_{01} = C_{00} + C_{(-1)1} = 1 + 0$ by (3.2), and (3.6) yields $C_{01} = 1 + \sum_{0 \le i < 0} C_{(-i-2)(-i)} = 1$, as the sum is empty. For the induction step, let the inductive hypothesis be $C_{(p-2)(p-1)} = 1 + \sum_{0 \le i < p-2} C_{(p-1-i-2)(p-1-i)}$. Then

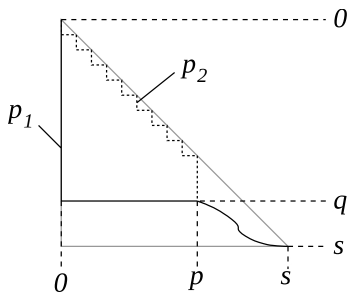


FIG. 3.2. *Smallest (p_1) and largest (p_2) paths (under f) among the paths that are equal up to* $\boxed{p\ q}$.

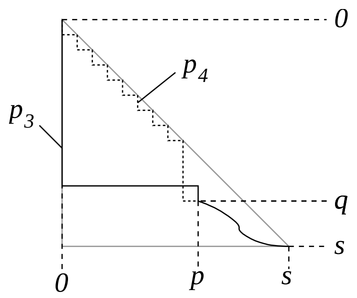


FIG. 3.3. *p_3 is the next-largest path (under f) after p_4 among those paths that are equal up to* $\boxed{p\ q}$.

$$\begin{aligned}
C_{(p-1)p} &= C_{(p-1)(p-1)} + C_{(p-2)p} && \text{(by (3.2))}\\
&= C_{(p-1)(p-2)} + C_{(p-2)(p-1)} + C_{(p-2)p} && \text{(again by (3.2))}\\
&= C_{(p-2)(p-1)} + C_{(p-2)p} && \text{(because } C_{(p-1)(p-2)} = 0)\\
&= 1 + \sum_{0\le i<p-2} C_{(p-1-i-2)(p-1-i)} + C_{(p-2)p} && \text{(inductive hypothesis)}\\
&= 1 + \sum_{1\le i<p-1} C_{(p-i-2)(p-i)} + C_{(p-2)p} && \text{(shifting indices)}\\
&= 1 + \sum_{0\le i<p-1} C_{(p-i-2)(p-i)} - C_{(p-2)p} + C_{(p-2)p}\\
&= 1 + \sum_{0\le i<p-1} C_{(p-i-2)(p-i)}\ .
\end{aligned}$$

We also need the following two lemmas for proving our claim.

LEMMA 3.3. *For $0 \le p \le q \le s$ and an arbitrary (but fixed) path $\overline{pq}$ from* $\boxed{s\ s}$ *to* $\boxed{p\ q}$, *let $\mathcal{P}_s^{\overline{pq}} \subseteq \mathcal{P}_s$ be the set of paths from* $\boxed{s\ s}$ *to* $\boxed{0\ 0}$ *that move along $\overline{pq}$ up to* $\boxed{p\ q}$. *Then f assigns the smallest number to the path $p_1 \in \mathcal{P}_s^{\overline{pq}}$ that first goes horizontally from* $\boxed{p\ q}$ *to* $\boxed{0\ q}$ *and then vertically to* $\boxed{0\ 0}$, *and the largest value to the path $p_2 \in \mathcal{P}_s^{\overline{pq}}$ that first goes vertically from* $\boxed{p\ q}$ *to* $\boxed{p\ p}$, *and then "crawls" along the main diagonal to* $\boxed{0\ 0}$ *(see also Fig. 3.2).*

Proof. The claim for p_1 is true because there are no more values added to sum when going only leftwards to the first column. The claim for p_2 follows from the "left-to-right" monotonicity of the Ballot Numbers: $C_{ij} < C_{(i+1)j}$ for all $0 \le i+1 \le j-1$ (this follows directly from $C_{(i+1)j} = C_{ij} + C_{(i+1)(j-1)}$ and the fact that for $0 \le i+1 \le j-1$, $C_{(i+1)(j-1)} > 0$). So taking the rightmost (i.e. highest) possible value from each row $q' \le q$ must yield the highest sum (note that f can add at most one Ballot Number from each row $q' \le q$). ◻

LEMMA 3.4. *Let p, q, and $\mathcal{P}_s^{\overline{pq}}$ be as in Lemma 3.3. Let $p_3 \in \mathcal{P}_s^{\overline{pq}}$ be the path that first moves one step upwards to* $\boxed{p\ (q-1)}$, *then horizontally until reaching* $\boxed{0\ (q-1)}$, *and then vertically to* $\boxed{0\ 0}$. *Let $p_4 \in \mathcal{P}_s^{\overline{pq}}$ be the path that first moves* *one step leftwards to* $\boxed{(p-1)\ q}$*, then vertically until reaching* $\boxed{(p-1)\ (p-1)}$*, and then "crawls" along the main diagonal to* $\boxed{0\ 0}$ *(see also Fig. 3.3). Then* $f(p_3) = f(p_4) + 1$*.*

Proof. Let S be the sum of the Ballot Numbers that have already been added to the sum of both p_3 and p_4 when reaching $\boxed{p\ q}$. Then $f(p_3) = S + C_{(p-1)q}$, and

$$f(p_4) = S + \sum_{p \le q' \le q} C_{(p-2)q'} + \sum_{0 \le i < p-2} C_{(p-i-3)(p-i-1)} \ ,$$

by simply summing over the Ballot Numbers that are added to S when making upwards moves.

Then

$$\begin{aligned} f(p_3) &= S + C_{(p-1)q} \\ &= S + \sum_{p-1 \le q' \le q} C_{(p-2)q'} && \text{(by (3.5))} \\ &= S + \sum_{p \le q' \le q} C_{(p-2)q'} + C_{(p-2)(p-1)} \\ &= S + \sum_{p \le q' \le q} C_{(p-2)q'} + 1 + \sum_{0 \le i < p-2} C_{(p-i-3)(p-i-1)} && \text{(by (3.6))} \\ &= f(p_4) + 1 \ , \end{aligned}$$

which proves the claim. ◻

This gives us all the tools for

LEMMA 3.5. *Function* f *defined by* (3.4) *is a bijective mapping from* $\mathcal{L}_s$ *to the interval* $\left[0 : C_s - 1\right]$.

Proof. The smallest path (under f) receives number 0, and the largest path (crawling along the main diagonal) receives number $\sum_{0 \le i < s-1} C_{(s-i-2)(s-i)} = C_{(s-1)s} - 1 = C_s - 1$ (by (3.6)). So $f(p) \in \left[0 : C_s - 1\right]$ for all paths $p \in \mathcal{P}_s$.

Injectivity can be seen as follows: different paths p_5 and p_6 must have one point $\boxed{p\ q}$ where they diverge; w.l.o.g. assume that p_5 continues with an upwards step to $\boxed{p\ (q-1)}$, and p_6 with a leftwards step to $\boxed{(p-1)\ q}$. Let p_5' be the path that equals p_5 up to $\boxed{p\ (q-1)}$, then goes horizontally to $\boxed{0\ (q-1)}$, and finally moves vertically to $\boxed{0\ 0}$. Also, let p_6' be the path that equals p_6 up to $\boxed{(p-1)\ q}$, then goes vertically to $\boxed{(p-1)\ (p-1)}$, and finally crawls along the main diagonal to $\boxed{0\ 0}$. Hence,

$$\begin{aligned} f(p_6) &\le f(p_6') && \text{(by Lemma 3.3)} \\ &= f(p_5') - 1 && \text{(by Lemma 3.4)} \\ &< f(p_5') \\ &\le f(p_5) && \text{(again by Lemma 3.3)} \ . \end{aligned}$$

The claim on surjectivity follows directly from the fact that $\left|\mathcal{L}_s\right| = \left|\mathcal{P}_s\right| = C_s$. ◻

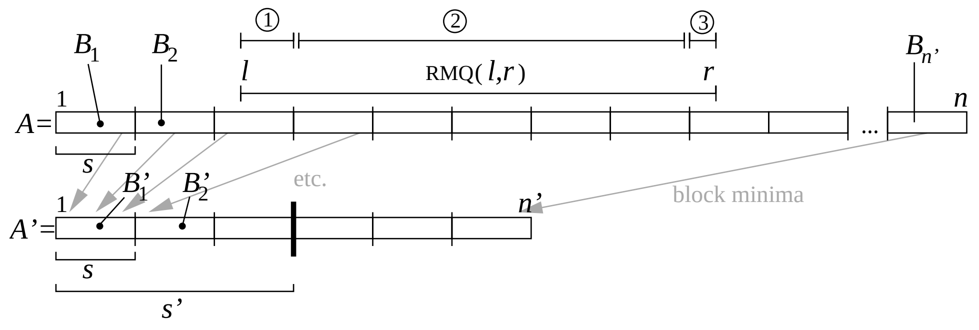


FIG. 3.4. *The input array A is divided into blocks $B_1, \dots, B_{n'}$ of size s, and a query* $\text{RMQ}_A(i,j)$ *is divided into three sub-queries ①–③. Array A' stores the block-minima, and is again divided into blocks $B'_1, \dots, B'_{\lceil n'/s \rceil}$ of size s. Further, s of these blocks are grouped into super-blocks of size s'.*

3.3. Preprocessing for Out-of-Block-Queries. It remains to show how the out-of-block-queries (those perfectly aligned with block boundaries at both ends) are answered. Proceeding as in previous schemes [1, 4] would result in a super-linear bit space $O(n \lg n)$, so we need a different approach. In principle, we could directly adapt the solution of the non-systematic scheme due to Sadakane [53] to our setting, which would result in $O\big(\frac{n \lg^2 \lg n}{\lg n}\big)$ bits of space. A further blocking level, however, can reduce this to $O\big(\frac{n \lg \lg n}{\lg n}\big)$ bits, as explained next. See also Fig. 3.4 for what follows.

For each of the $n' = \lceil \frac{n}{s} \rceil$ blocks B_i, we store the minimum of B_i in a new array $A'[1, n']$ at $A'[i]$, such that answering out-of-block-queries now corresponds to answering RMQs on A'. To this end, array A' is *again* divided into blocks of size s, say $B'_1, \dots, B'_{\lceil n'/s \rceil}$. A query $\text{RMQ}_{A'}(i,j)$ is again decomposed into three non-overlapping sub-queries: one out-of-block query, and two in-block-queries. The in-block-queries are handled with the same mechanism as in § 3.2, i.e., by calculating a *type* for each block in A', storing these types in an array T', and using a lookup-table to answer the queries. In fact, since the block size remains untouched, we can use the same universal lookup-table P as in § 3.2; only the type-array T' needs to be stored.

For answering the out-of-block-queries on A', we could keep recursing in the same manner, but this would not result in constant query time. So we need a different strategy, as explained next. In essence, we do this with a two-level storage scheme due to Sadakane [52, 53]. We group s contiguous blocks $B'_{is+1}, \dots, B'_{(i+1)s}$ into *super-blocks* consisting of $s' = s^2$ elements. Call the resulting super-blocks $B''_1, \dots, B''_{\lceil n'/s' \rceil}$. We first wish to precompute the answers to all RMQs in A' that span over at least one such super-block. To do so, define a table $M''\big[1, \lceil n'/s' \rceil\big]\big[0, \big\lfloor \lg\big(\lceil n'/s' \rceil\big)\big\rfloor\big]$, where $M''[i][j]$ stores the position of the minimum of super-blocks $B''_i \dots, B''_{i+2^j-1}$ (minimum in $A'[(i-1)s'+1, (i+2^j-1)s']$). The first row $M''[i][0]$ can be filled by a linear pass over A', and for $j > 0$ we use a dynamic programming approach by setting $M''[i][j] = \text{argmin}_{k \in \{M''[i][j-1], M''[i+2^{j-1}][j-1]\}}\big\{A'[k]\big\}$.

To find the minimum in super-blocks $B''_i, \dots, B''_j$, we decompose the range $[i, j]$ into two (possibly overlapping) sub-ranges whose length is a power of two: letting $p = \big\lfloor \lg(j-i+1) \big\rfloor$, the corresponding sub-ranges are $\big[i, i+2^p-1\big]$ and $\big[j-2^p+1, j\big]$. Hence, the minimum of $B''_i, \dots, B''_j$ can be found by $\text{argmin}_{k \in \{M''[i][p], M''[j-i+1][p]\}}\big\{A'[k]\big\}$.

In a similar manner we precompute the answers to all RMQs in A' that span over at least one block, but *not* over a super-block. These answers are stored in a table

$M'\big[1, \lceil n'/s\rceil\big]\big[0, \big\lfloor \lg\big(\lceil s'/s\rceil\big)\big\rfloor\big]$, where $M'[i][j]$ stores the position of the minimum of blocks $B'_i \ldots, B'_{i+2^j-1}$ (minimum in $A'\big[(i-1)s+1, (i+2^j-1)s\big]$). Again, dynamic programming can be used to fill table M' in optimal time.

Summarizing this section, an out-of-block-query in A is decomposed into at most two in-block-queries in A' (answered with T' and P), two out-of-block-queries in A' (answered by consulting M'), and one out-of-super-block-query in A' (answered with M'').

3.4. Space Analysis. Let us now analyze the space occupied by the scheme given in § 3.2–3.3. We start with the structures from § 3.2. Recall that the block size is $s = \big\lceil \frac{\lg n}{4} \big\rceil$. To store the type of each block, array T has length $\lceil n/s\rceil$, and because of Lemma 3.1, the numbers are in the range $\big[0 : C_s - 1\big]$. Note $C_s = 4^s/\big(\sqrt{\pi}s^{3/2}\big)\big(1+O(s^{-1})\big)$ by Stirling, in particular $C_s \le \frac{4^s}{s^{3/2}}$. This implies that the number of bits to encode T is

$$\begin{aligned}
|T| &= \left\lceil \frac{n}{s} \right\rceil \cdot \lceil \lg C_s \rceil \\
&\le \frac{n}{s} \cdot \lg C_s + \frac{n}{s} + \lg C_s + 1 \\
&\le \frac{n}{s} \cdot \lg \frac{4^s}{s^{3/2}} + \frac{n}{s} + \lg \frac{4^s}{s^{3/2}} + 1 \\
&= \frac{n}{s} \cdot \left(2s - \frac{3}{2}(\lg\lg n - 2)\right) + \frac{n}{s} + O(\lg n) \\
&= 2n - 6\frac{n \lg\lg n}{\lg n} + O\left(\frac{n}{\lg n}\right) \ .
\end{aligned}$$

To analyze the space of the lookup-table P, by Lemma 3.1 we know that P has only $C_s \le \frac{4^s}{s^{3/2}}$ rows, one for each possible block-type. For each type t and a block B of type t we need to precompute $P[t][i][j] = \text{RMQ}_B(i,j)$ for all $1 \le i \le j \le s$; this would take $O\big(\frac{4^s}{s^{3/2}} s^2 \cdot \lg s\big)$ bits of space. Some further space can be saved if we use the method described by Alstrup et al. [1], which uses a single bit vector of length s to represent the answers to all RMQs that *start* at a given point inside the block. The total space is thus

$$\begin{aligned}
|P| &= O\left(\frac{4^s}{s^{3/2}} s \cdot s\right) \\
&= O\left(n^{1/2}\sqrt{\lg n}\right) \\
&= o(n/\lg n)
\end{aligned}$$

bits.

We come to the structures from § 3.3. First note that array A' need not be stored at all, as

$$A'[i] = A\left[(i-1)s + P[T[i]][1][s]\right] \ .$$

Array T' is of length $\lceil n'/s\rceil$ and stores values of the same range as T; the space is thus

$$\begin{aligned} |T'| &= \left\lceil \frac{n'}{s} \right\rceil \cdot \lceil \lg C_s \rceil \\ &= O\left(\frac{n}{s^2} \cdot 2s\right) \\ &= O\left(\frac{n}{\lg n}\right) . \end{aligned}$$

Table M'' has dimensions $\lceil n'/s' \rceil \times \lfloor \lg \lceil n'/s' \rceil \rfloor$ and stores values up to $n' \leq n$; the total number of bits needed is therefore

$$\begin{aligned} |M''| &= O\left(\frac{n'}{s'} \lg\left(\frac{n'}{s'}\right) \cdot \lg n\right) \\ &= O\left(\frac{n}{\lg^3 n} \lg n \cdot \lg n\right) \\ &= O\left(\frac{n}{\lg n}\right) . \end{aligned}$$

Table M' has dimensions $\lceil n'/s \rceil \times \lfloor \lg \lceil s'/s \rceil \rfloor$. If we just store the offsets of the minima then the values do not become greater than s'; the total number of bits needed for M is therefore

$$\begin{aligned} |M'| &= O\left(\frac{n'}{s} \lg\left(\frac{s'}{s}\right) \cdot \lg s'\right) \\ &= O\left(\frac{n \lg^2 \lg n}{\lg^2 n}\right) \\ &= o(n/\lg n) . \end{aligned}$$

For constructing the scheme, we first note that tables P, M', and M'' can be filled directly with no extra space. When filling T and T' with Alg. 1, we need $O(\lg n \lg \lg n)$ bits for array R (R stores $s = O(\lg n)$ integers in the range $[1 : s]$), plus $O(\lg^3 n)$ bits for the Ballot Numbers C_{pq} (an array of $\leq s^2$ integers in the range $[1 : C_s] = [1 : O(\lg n)]$ for $s = O(\lg n)$).

Because the query algorithm needs to compare several values in A for obtaining the final answer, this scheme is systematic. Letting t_A denote the time to access an element from the input array A ($t_A = O(1)$ for "normal," uncompressed arrays), we can thus state:

LEMMA 3.6. *For a static array A with n elements from a totally ordered set and access time t_A, there exists a preprocessing scheme for RMQs with time complexity $\langle O(n \cdot t_A), O(t_A) \rangle$ and bit-space complexity $\left[\!\left[O(\lg^3 n), |A| + 2n - 6\frac{n \lg\lg n}{\lg n} + O\left(\frac{n}{\lg n}\right) \right]\!\right]$.*

Note that for sufficiently large n the negative $\frac{n \lg \lg n}{\lg n}$-term becomes larger than the $O\left(\frac{n}{\lg n}\right)$-term. Hence, the final space on top of A is asymptotically less than $2n$ bits.

3.5. The Final Result. Finally, we show how to lower the $2n$-bit term from Lemma 3.6 to $2n/c(n)$ for an arbitrary integer function $c(n) \in O(n^{\varepsilon})$ for a constant $0 < \varepsilon < 1$. The idea is to build groups of $c(n)$ consecutive elements from the input array A, construct a (conceptual) new array B consisting of the minima in these groups, and construct the scheme from Lemma 3.6 on B. A query in A is then translated into a query in B, consisting of exactly the groups that are *strictly* contained in the query in A. Because objects in B correspond to groups of $c(n)$ consecutive objects in A, every access to B now results in a *scan* of $c(n)$ entries in A, as B is not actually present. Further, we also scan at most $c(n)$ entries in A at both ends of the query, and compare these values to the minimum obtained by querying B. Note that we do not even have to store B at construction time, as every of the $O\big(n/c(n)\big)$ accesses to B can be simulated by $O\big(c(n)\big)$ accesses to A, resulting in $O(n)$ preprocessing time in total.

This leads us to the following theorem (the restriction on $c(n)$ is only there to keep the second-order term simple).

Theorem 3.7. *For a static array A with n elements from a totally ordered set and access time t_A, there is a preprocessing scheme for RMQs with time complexity $\langle O(n \cdot t_A), O(c(n) \cdot t_A)\rangle$ and space complexity $\Big[\!\!\Big[O\big(\lg^3 n\big), |A| + \frac{2n}{c(n)} - \Theta\big(\frac{n \lg\lg n}{c(n)\lg n}\big)\Big]\!\!\Big]$, for an arbitrary positive integer function $c(n) \in O(n^{\varepsilon})$ (constant $0 < \varepsilon < 1$).*

Function $c(n)$ can be constant, in which case Thm. 3.7 gives optimal $O(1)$ query time (assuming constant access time t_A). Notwithstanding, there are also applications where $\omega(1)$ query time suffices [23]. In this case, Thm. 3.7 is stronger than the currently best solution for RMQs with sublinear space [23, Lemma 2]. For example, we can achieve $O(\lg\lg n)$ query time with $O\big(\frac{n}{\lg\lg n}\big)$ space on top of A (setting $c(n) = \lceil \lg\lg n\rceil$ in Thm. 3.7), whereas [23] would give $O\big(\lg\lg n \cdot \lg^2 \lg\lg n\big)$ query time within that space.

We finally stress that our algorithm is easy to implement on PRAMs (or real-world shared-memory machines), where with n/t processors the preprocessing runs in time $\Theta(t)$ if $t = \Omega(\lg n)$, which is work-optimal. This is simply because the minimum-operation is associative and can hence be parallelized after a $\Theta(t)$ sequential initialization [57].

4. Compressed Preprocessing Scheme.

Let us now consider input arrays A of length n that are compressible. As already mentioned in the introduction, compressibility is usually measured by the order-k entropy $H_k(A)$, as $nH_k(A)$ provides a lower bound on the number of bits needed to encode A by any compressor that considers a *context* of length k when it encodes a symbol in A. We now show that the simple *text*-encoding by Ferragina and Venturini [16] is also effective for our type-array T. In this section, σ denotes the size of the "alphabet" Σ, i.e., the number of *different* objects in A.

4.1. Adapting the Ferragina-Venturini-Scheme to RMQs.

We explain how to adapt the encoding due to Ferragina and Venturini [16] to yield the first entropy-bounded preprocessing scheme for RMQs. The basis is the RMQ-algorithm from Lemma 3.6, and the block size is set again to $s = \lceil \frac{\lg n}{4} \rceil$. Again, B_j denotes the j'th block in A. The idea for compression is to reduce the size of the type-array T (see the first two paragraphs after the proof of Lemma 3.1 for the meaning of T), as all other structures are already of size $o(n)$. Compressing T works as follows.

- Let $\mathcal{T}$ be the set of occurring block types in A: $\mathcal{T} = \big\{T[i] \mid i \in [1 : \lceil n/s \rceil]\big\}$.
- For $T \in \mathcal{T}$ let $n(T)$ be the number of occurrences of block type T in $\mathcal{T}$, $n(T) = \big|\big\{i \in \big[1 : \lceil n/s \rceil\big] \mid T[i] = T\big\}\big|$. Sort the elements of $\mathcal{T}$ by $n(T)$ in decreasing order, and let $r\big(B_j\big)$ the rank of B_j's Cartesian Tree in this sorted list, $r\big(B_j\big) = \big|\big\{T \in \mathcal{T} \mid n(T) \le n\big(T[j]\big)\big\}\big|$.

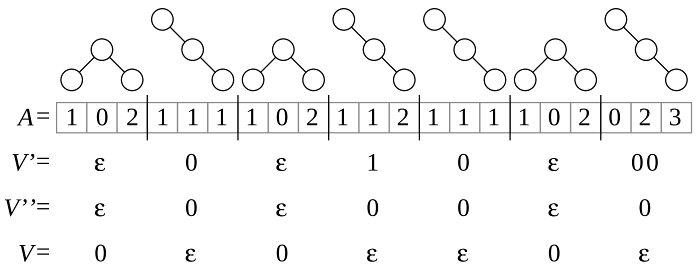


FIG. 4.1. *Illustration to the compressed representation of RMQ-information. On top of each block of size $s = 3$ we display its Canonical Cartesian Tree. The final encoding can be found in the row labeled V; the rows labeled V' and V'' are solely for the proof of Thm. 4.1.*

- Assign to each block B_j a codeword $c(B_j)$ that is the binary string of rank $r(B_j)$ in $\mathcal{B}$, the canonical enumeration of all binary strings: $\mathcal{B} = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$. The codeword $c(B_j)$ will be used as the type of block B_j; there is *no need to recover the original block types*.
- Build a sequence $V = c(B_1)c(B_2)\dots c(B_{\lceil n/s \rceil})$. In other words, V is obtained by concatenating the codewords for each block. See Fig. 4.1 for an example (ignore for now the rows labeled V' and V'').
- In order to find the beginning and ending of B_j's codeword in V, we use again a two-level scheme for storing the starting position of B_j's encoding in V: we group every s contiguous blocks in A into a super-block. Table D' stores the beginning (in V) of the encodings of these super-blocks. Table D does the same for the blocks, but storing the positions only relative to the beginning of the super-block encoding. These tables can be filled "on the fly" when writing the compressed string V.

D and D' can be used to reconstruct the codeword (and hence the type) of block B_j: simply extract the beginning of block j and that of $j + 1$ (if existent); thus, the above structures *substitute* the type-array T. The result of this section can now be stated as follows:

THEOREM 4.1. *For a static array A with n elements from a totally ordered set of size σ and access time t_A, there exists a preprocessing scheme for RMQs with time complexity $\langle O(n \cdot t_A), O(t_A)\rangle$ and bit-space complexity*

$$\left[\!\left[O(\sqrt{n/\lg n}), \min\{nH_k(A) + O(\frac{n(k\lg\sigma + \lg\lg n)}{\lg n}), 2n\} + |A| \right]\!\right] ,$$

simultaneously over all k.

Proof. We start by bounding the size of V. Assume first that instead of compressing the block types (i.e., array T), we run the above compression algorithm directly on the *contents* of A, with the same block size. In other words, we assign the same codeword c' to two size-s-blocks iff their contents is equal, and these codewords are derived from the frequencies of the blocks in A. See also Fig. 4.1, where the sequence thus obtained is called V', and $c'(102) = \epsilon$, $c'(111) = 0$, $c'(112) = 1$, and $c'(023) = 00$. It can be shown [16, Thm. 1] that the resulting codeword $c'(B_j)$ produced for block B_j is always smaller than if one were to compress the contents of that block with a k-th order Arithmetic Encoder. In turn, González and Navarro [30] proved that the total output of such an Arithmetic Encoder is bounded by $nH_k(A) + O(\frac{nk\lg\sigma}{b})$, where b is the block-size (in our case $b = O(\lg n)$). Here, the $O(\frac{nk\lg\sigma}{b})$-term accounts for encoding the first k symbols of each of the n/b blocks with $\lg \sigma$ bits; hence, for this we need to assume that the elements in Σ are integers in a range of size $\sigma^{O(1)}$.

Now, observe that if two blocks in the original array A are equal, then they also have the same Cartesian Tree and thus the same block type; so if we encode each block-type with the *shortest* codeword $c'(B_j)$ among all the codewords for blocks that have the same Cartesian Tree, the resulting sequence V'' will always be shorter than V'. See Fig. 4.1 for an example. Now our encoding V cannot be longer than V'', as it assigns codewords optimally (shorter codewords to more frequent types). Finally, by noting that the compressed V is never larger than the uncompressed T, we can conclude that $|V| = \min\{nH_k(A) + O\big(\frac{nk\lg\sigma}{\lg n}\big), 2n\}$. Further, we can drop the assumption that the values are in a range of size $\sigma^{O(1)}$, as we can think of having scaled down the alphabet down to $[1, \sigma]$, and then running the algorithm on the transformed array (which has the same answers to all RMQs).

We continue with tables D and D'. Because the block types are in the range $[0 : C_s - 1]$, a super-block spans at most $s' = s\lg C_s = O(\lg^2 n)$ bits in V. Consequently, the size of table D is $|D| = O(n/s \cdot \lg s') = O\big(\frac{n\lg\lg n}{\lg n}\big)$ bits. The size of table D' is simply $|D'| = O\big(n/s' \cdot \lg|V|\big) = O\big(\frac{n}{\lg n}\big)$.

Arrays P, T', M' and M'' of the RMQ-scheme from § 3 are still needed for answering the out-of-block-queries. Their space can be bounded by $o\big(\frac{n\lg\lg n}{\lg n}\big)$ (see § 3.4). The claim on the final space follows.

We finally show that the scheme can be constructed in $O(n \cdot t_A)$ time with little extra space. In a first scan of A, we only *count* the number of occurrences of each block type *without* actually storing the types; this needs an array of size $O(C_s \cdot \lg n) = O\big(\sqrt{n/\lg n}\big)$ bits. We then sort this array in-place [24] and assign the codes according to the frequency, needing at most additional $O\big(\sqrt{n/\lg n}\big)$ bits. A second scan over A constructs the block types *again*, and directly writes the output stream V, along with D and D'. □

Thm. 4.1 is independent of the representation and the size of the keys in A; we only require $O(t_A)$-read-only-access to the elements from A. For example, A could contain large uncompressed values and reside on disk (but still have a small alphabet size and a small entropy), or A could be compressed to something even smaller than $nH_k(A)$. However, if the elements in A have size $O(\lg\sigma)$, then A itself could be compressed to $nH_k(A) + O\big(\frac{nk\lg\sigma}{\lg n}\big)$ bits with Ferragina and Venturini's scheme [16]. Then Thm. 4.1 would give $2nH_k(A) + o(n)$ total space. In this case we can do better by incorporating the RMQ-information directly into the compression of A:

COROLLARY 4.2. *For a static array $A[1, n]$ consisting of $O(\lg\sigma)$-bit numbers, there exists a preprocessing scheme for RMQs with time complexity $\langle O(n), O(1)\rangle$ and bit-space complexity*

$$\left[\!\!\left[|A| + O\big(\sqrt{n}\lg n\big), nH_k(A) + O\left(\frac{n(k\lg\sigma + \lg\lg n)}{\lg_\sigma n}\right) \right]\!\!\right] ,$$

simultaneously over all k.

Proof. We use Ferragina and Venturini's scheme [16, Thm. 1] to compress A, which divides A into blocks of size $b = \lfloor \frac{1}{2}\lg_\sigma n \rfloor$. To their table r^{-1}, which gives the contents of a block with a given rank, we attach the type (a.k.a. Cartesian Tree) of the corresponding block, needing additional $O\big(\sigma^b \cdot \lg C_b\big) = O\big(\sqrt{n} \cdot \lg_\sigma n\big)$ bits. This gives $O(1)$-access to both the contents of the block and its Cartesian Tree, so we can proceed as in the scheme from Lemma 3.6, without having to store T. If the resulting array A' is divided into blocks/super-blocks with the original block size s instead of b, then this increases the space of for T', M', and M'' only by a factor of $\lg \sigma$, but the resulting terms are all within the space for compressing A. The scheme can be constructed in linear time using additional tables of size $O(\sigma^b \cdot \lg n) = O(\sqrt{n} \lg n)$ bits, using the ideas from the last paragraph in the proof of Thm. 4.1. The claim follows. $\square$

5. Optimal Preprocessing in the Non-Systematic Setting. We now turn our attention to the non-systematic setting. In this section, we show a preprocessing scheme of asymptotically optimal final size. For ease of presentation, our new scheme always returns the *rightmost* minimum in case of draws — though it can be easily arranged to return the usual *leftmost* minimum if this is desired (e.g., by conceptually reversing both the input array and the queries).

5.1. 2-dimensional Min-Heaps. Recall that $A[1, n]$ is the array to be preprocessed for RMQs. For technical reasons, we define $A[0] = -\infty$ as the "artificial" overall minimum. We first need an auxiliary definition:

DEFINITION 5.1. *For* $1 \le i \le n$, *let* $\text{PSV}_A(i) = \max\{j < i \mid A[j] < A[i]\}$ *denote the* previous smaller value *of position* i.

The previous smaller value has a "crossing-free" property, stated formally in the following

LEMMA 5.2. *For* $1 \le i < j \le n$, $\text{PSV}_A(j) \notin [\text{PSV}_A(i) + 1 : i - 1]$.

Proof. Assume $\text{PSV}_A(j) = k \in [\text{PSV}_A(i) + 1 : i - 1]$. Then $A[k] < A[j]$ and $A[m] \ge A[j]$ for all $m \in [k + 1 : j]$, in particular $A[i] \ge A[j]$ because $k < i < j$. So $A[i] > A[k]$, a contradiction to the fact that $\text{PSV}_A(i) < k$. $\square$

The basis for our non-systematic scheme will be a new tree, the *2d-Min-Heap*, defined as follows.

DEFINITION 5.3. *The* 2d-Min-Heap $\mathcal{M}^A$ *of* A *is an ordered labeled tree with vertices* $v_0, \ldots, v_n$, *where* v_i *is labeled with* i *for all* $0 \le i \le n$. *For* $1 \le i \le n$, *the parent node of* v_i *is the node labeled* $\text{PSV}_A(i)$. *The order of the children is chosen such that their labels are increasing from left to right.*

By Lemma 5.2 $\mathcal{M}^A$ is a well-defined tree with the root being always labeled 0. Observe that a node v_i can be uniquely identified by its label i, which we will do henceforth. See Fig. 5.1 for an example.

We note the following useful properties of $\mathcal{M}^A$.

LEMMA 5.4. *Let* $\mathcal{M}^A$ *be the 2d-Min-Heap of* A.

(i) *Let* i *be a node in* $\mathcal{M}^A$ *with children* $x_1, \ldots, x_k$. *Then* $A[i] < A[x_j]$ *for all* $1 \le j \le k$.

(ii) *Let* i *be a node in* $\mathcal{M}^A$. *Then* $\mathcal{M}^A_i$, *the subtree of* $\mathcal{M}^A$ *rooted at* i, *consists of nodes* $[i : h - 1]$ *for some* $i < h \le n + 1$.

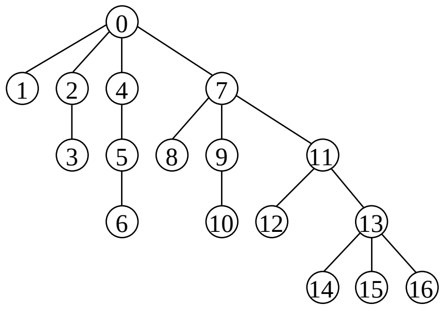
(iii) *The node labels correspond to the preorder-numbers of* $\mathcal{M}^A$ *(counting starts at 0).*

(iv) *Again, let* i *be a node in* $\mathcal{M}^A$ *with children* $x_1, \ldots, x_k$. *Then* $A[x_j] \le A[x_{j-1}]$ *for all* $1 < j \le k$.

Proof. We prove each property in turn.

(i) Follows immediately from Def. 5.3.

(ii) Let $h > i$ be the smallest index with $\text{PSV}_A(h) < i$, or $h = n + 1$ if no such value exists. Because $i \le \text{PSV}_A(k) < h$ for all $k \in [i + 1 : h - 1]$ and the transitivity of "$\le$", $k \in \mathcal{M}^A_i$ for all such k. It follows immediately from Lemma 5.2 that these are the only values in $\mathcal{M}^A_i$ apart from i itself.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$A=$	$-\infty$	5	3	4	3	4	5	$\underset{i}{1}$	3	$\underset{j}{2}$	$\underset{i}{4}$	2	5	3	5	$\underset{j}{5}$	4
$U=$	((((()	)	()	)	()	()	)	((()	)	()	)	(()	)	((()	)	)	)
$E=$	123454	3	43	2	32	32	1	234$\underset{x}{3}$	$\underset{y}{2}$	32	$\underset{x}{1}$	232	1	2343	$\underset{y}{2}$	1	0

FIG. 5.1. *Top: The 2d-Min-Heap $\mathcal{M}^A$ of the input array A. Bottom: $\mathcal{M}^A$'s DFUDS U and U's excess sequence E. Two example queries* $\text{RMQ}_A(i,j)$ *are underlined, including their corresponding queries* $\pm 1\text{RMQ}_E(x,y)$.

(iii) Let $\pi(i)$ denote the preorder-number of i in $\mathcal{M}^A$. We proceed by induction on i. The claim is obviously true for the root 0. Now let $i \geq 1$, and assume the claim is true for all $i' < i$. To prove the claim for i, let $j = \text{PSV}_A(i) < i$ be the parent of i, and $i_1, \ldots, i_k$ be j's children with $i_x = i$ for some $1 \leq x \leq k$. By the definition of preorder,

$$\pi(i) = \begin{cases} \pi(j) + 1 & \text{if } x = 1\ , \\ \pi(i_{x-1}) + \left|\mathcal{M}^A_{i_{x-1}}\right| & \text{otherwise.} \end{cases}$$

In the first case, by the inductive hypothesis we know $\pi(i) = j + 1$, so it remains to prove $i = j+1$. For the sake of contradiction, assume $i > j+1$ and let $m = \text{RMQ}_A(j+1, i-1)$. Then $\text{PSV}_A(m) = j$, as from Lemma 5.2 we cannot have $\text{PSV}_A(m) < j$, and $A[m'] \geq A[m]$ for all $m' \in [j+1 : m]$ because m is a range minimum. Hence $m < i$ is a child i_y of j. Due to the order of the children in $\mathcal{M}^A$ we must have $y < x = 1$, contradicting the fact that $i = i_x$ is the first child of j.
In the second case, by the inductive hypothesis we know $\pi(i) = i_{x-1} + \left|\mathcal{M}^A_{i_{x-1}}\right|$. From property (ii), we have $\mathcal{M}^A_{i_{x-1}} = \left[i_{x-1} : h-1\right]$ for some $h > i_{x-1}$, hence $|\mathcal{M}^A_{i_{x-1}}| = h - i_{x-1}$. This implies $\pi(i) = h$, so it remains to show that $h = i$. For the sake of contradiction assume $h \neq i$, implying $h < i$ because $\mathcal{M}^A_{i_{x-1}}$ is consecutive (note $i \notin \mathcal{M}^A_{i_{x-1}}$). Let $m = \text{RMQ}_A(h, i-1)$. We now proceed as in the first case. From Lemma 5.2, $\text{PSV}_A(m) \geq j$ and $\text{PSV}_A(m) \notin \left[j+1 : i_{x-1}-1\right]$. From the fact that m is a range minimum, $\text{PSV}_A(m) < h$. But we cannot have $\text{PSV}_A(m) \in \left[i_{x-1} : h-1\right]$, for otherwise $m \in \mathcal{M}^A_{i_{x-1}}$, contradicting the size of $\mathcal{M}^A_{i_{x-1}}$. In total, $\text{PSV}_A(m) = j$, so $i_{x-1} < m < i_x$ is a child of j. This contradicts again the order of the children of $\mathcal{M}^A$, which requires that m should appear between i_{x-1} and i_x.

(iv) Assume for the sake of contradiction that $A\left[x_j\right] > A[x_{j-1}]$ for two children x_j and x_{j-1} of i. From property (iii), we know that $i < x_{j-1} < x_j$, implying $\text{PSV}_A(x_j) \geq x_{j-1} > i$, in contradiction to $\text{PSV}_A(x_j) = i$ by Def. 5.3. $\square$

Properties (i) and (iv) of the above lemma explain the choice of the name "2d-

Min-Heap," because $\mathcal{M}^A$ exhibits a minimum-property on both the parent-child- and the sibling-sibling-relationship, i.e., in two dimensions.

The following lemma will be central for our scheme, as it establishes the desired connection of 2d-Min-Heaps and RMQs.

LEMMA 5.5. *Let $\mathcal{M}^A$ be the 2d-Min-Heap of A. For arbitrary nodes i and j, $1 \le i < j \le n$, let ℓ denote the LCA of i and j in $\mathcal{M}^A$ (recall that we identify nodes with their labels). Then if $\ell = i$, $\text{RMQ}_A(i,j)$ is given by i, and otherwise, $\text{RMQ}_A(i,j)$ is given by the child of ℓ that is on the path from ℓ to j.*

Proof. Recall that $\mathcal{M}^A_x$ denotes the subtree of $\mathcal{M}^A$ that is rooted at x. There are two cases to prove.

1. If $\ell = i$, this means that j is a descendant of i. Property (ii) of Lemma 5.4 implies that all nodes $[i:j]$ are in $\mathcal{M}^A_i$, and the recursive application of property (i) implies that $A[i]$ is the minimum in $A[i,j]$.

2. Now assume $\ell \ne i$. Let $x_1, \dots, x_k$ be the children of ℓ. Further, let α and β ($1 \le \alpha \le \beta \le k$) be defined such that $\mathcal{M}^A_{x_\alpha}$ contains i, and $\mathcal{M}^A_{x_\beta}$ contains j. Because $\ell \ne i$ and property (iii) of Lemma 5.4, we must have $\ell < i$; in other words, the LCA is not in the query range. But also due to property (iii), every node in $[i:j]$ is in $\mathcal{M}^A_{x_\gamma}$ for some $\alpha \le \gamma \le \beta$, and $x_\gamma \in [i:j]$ for all $\alpha < \gamma \le \beta$. Taking this together with property (i), we see that $\{x_\gamma \mid \alpha < \gamma \le \beta\}$ are the only candidate positions for the minimum in $A[i,j]$. Due to property (iv), we see that x_β (the child of ℓ on the path to j) is the position where the overall minimum in $A[i,j]$ occurs. □

To achieve the optimal $2n + o(n)$ bits for our scheme, we represent the 2d-Min-Heap $\mathcal{M}^A$ by its DFUDS U (occupying $2n$ bits), plus $o(n)$-bit structures for *rank*$_)$-, *select*$_)$-, and *findopen*-operations on U (see § 2). We further need structures for $\pm 1\text{RMQ}$ on the *excess-sequence* $E[1,2n]$ of U, defined as

$$E[i] = \mathit{rank}_{(}(U,i) - \mathit{rank}_{)}(U,i)\ . \tag{5.1}$$

This sequence clearly satisfies the property that subsequent elements differ by exactly 1, and is already encoded in the right form (by means of the DFUDS U) for applying the $\pm 1\text{RMQ}$-scheme from § 2.5.

The reasons for preferring the DFUDS over the BPS-representation [41] of $\mathcal{M}^A$ are (1) the operations needed to perform on $\mathcal{M}^A$ are particularly easy on DFUDS (see the next corollary), and (2) we have found a fast and space-efficient algorithm for constructing the DFUDS directly (see the next section).

COROLLARY 5.6. *Given the DFUDS U of $\mathcal{M}^A$, $\text{RMQ}_A(i,j)$ can be answered in $O(1)$ time by the following sequence of operations ($1 \le i < j \le n$).*

```
1. $x \leftarrow \mathit{select}_{)}(U, i+1)$
2. $y \leftarrow \mathit{select}_{)}(U, j)$
3. $w \leftarrow \pm 1\text{RMQ}_E(x, y)$
4. if $\mathit{rank}_{)}\big(U, \mathit{findopen}(U, w)\big) = i$ then return $i$
5. else return $\mathit{rank}_{)}(U, w)$
```

Proof. Let ℓ be the true LCA of i and j in $\mathcal{M}^A$. Inspecting the details of how LCA-computation in DFUDS is done [36, Lemma 3.2], we see that after the $\pm 1\text{RMQ}$-call in line 3 of the above algorithm, $w + 1$ contains the starting position in U of the encoding of ℓ's child that is on the path to j.[6] Line 4 checks if $\ell = i$ by comparing their preorder-numbers and returns i in that case (case 1 of Lemma 5.5) — it follows

[6]In line 1, we correct a minor error in the original article [36] by computing the starting position x slightly differently, which is necessary in the case that $i = \text{LCA}(i,j)$ (confirmed by K. Sadakane, personal communication, May 2008).

from the description of the parent-operation in the original article on DFUDS [5] that this is correct. Finally, in line 5, the preorder-number of ℓ's child that is on the path to j is computed correctly (case 2 of Lemma 5.5). □

We have shown these operations so explicitly in order to emphasize the simplicity of our approach. Note in particular that not all operations on DFUDS have to be "implemented" for our RMQ-scheme, and that we find the correct child of the LCA ℓ directly, without finding ℓ explicitly. We encourage the reader to work on the examples in Fig. 5.1, where the respective RMQs in both A and E are underlined and labeled with the variables from Cor. 5.6.

5.2. Construction of 2d-Min-Heaps. We show how to construct the DFUDS U of $\mathcal{M}^A$ in linear time and $n + o(n)$ bits of extra space. We first give a general $O(n)$-time algorithm that uses $O(n \lg n)$ bits (§ 5.2.1), and then show how to reduce its space to $n + o(n)$ bits, while still having linear running time (§ 5.2.2).

5.2.1. The General Linear-Time Algorithm. We show how to construct U (the DFUDS of $\mathcal{M}^A$) in linear time. The idea is to scan A from *right to left* and build U from right to left, too. Suppose we are currently in step i ($n \geq i \geq 0$), and $A[i+1, n]$ have already been scanned. We keep a stack $S[1, h]$ (where $S[h]$ is the top) with the properties that $A\big[S[h]\big] \geq \cdots \geq A\big[S[1]\big]$, and $i < S[h] < \cdots < S[1] \leq n$. S contains exactly those indices $j \in [i+1, n]$ for which $A[k] \geq A[j]$ for all $i < k < j$. Initially, both S and U are empty. When in step i, we first write a ')' to the current beginning of U, and then pop all w indices from S for which the corresponding entry in A is strictly greater than $A[i]$. To reflect this change in U, we write w opening parentheses '(' to the current beginning of U. Finally, we push i on S and move to the next (i.e. preceding) position $i - 1$. It is easy to see that these changes on S maintain the properties of the stack. If $i = 0$, we write an initial '(' to U and stop the algorithm.

The correctness of this algorithm follows from the fact that due to the definition of $\mathcal{M}^A$, the degree of node i is given by the number w of array-indices to the right of i that have $A[i]$ as their previous smaller value (Def. 5.3). Thus, in U node i is encoded as '$(^w)$', which is exactly what we do. Because each index is pushed and popped exactly once on/from S, the linear running time follows.

5.2.2. $O(n)$-bit Solution. The only drawback of the above algorithm is that stack S requires $O(n \lg n)$ bits in the worst case. We solve this problem by representing S as a *bit-vector* $S'[1, n]$. $S'[i]$ is 1 if i is on S, and 0 otherwise. In order to maintain constant time access to S, we use a standard blocking-technique as follows. We logically group $s = \lceil \frac{\lg n}{2} \rceil$ consecutive elements of S' into *blocks* $B_0, \ldots, B_{\lfloor \frac{n-1}{s} \rfloor}$. Further, $s' = s^2$ elements are grouped into *super-blocks* $B'_0, \ldots, B'_{\lfloor \frac{n-1}{s'} \rfloor}$.

For each super-block B'_z that contains at least one 1, in a new table M' at position z we store the number of the leftmost *block* to the right of B'_z that contains a 1: $M'[z] = \min\big\{y \geq (z+1)s \mid B_y \text{ contains a } 1\big\}$ if B'_z contains a 1, and $M'[z]$ is undefined otherwise. Here and in the following, the minimum of the empty set is defined as $+\infty$. Likewise, in a new table M at position x we store the number of the leftmost block to the right of B_x that contains a 1, this time only within x's super-block B'_z ($z = \lfloor \frac{x}{s} \rfloor$) and relative to the beginning of B_x in B'_z: $M[x] = \min\big\{x < y < (z+1)s \mid B_y \text{ contains a } 1\big\} - zs$ if B_x contains a 1, and $M[x]$ is undefined otherwise. Note that some values in M and M' are undefined, but they will never be accessed by the algorithm. These tables need $|M| = O\big(\frac{n}{s} \cdot \lg \frac{s'}{s}\big) = O\big(\frac{n \lg \lg n}{\lg n}\big)$ and $|M'| =$

$O\big(\frac{n}{s'} \cdot \lg \frac{n}{s}\big) = O\big(\frac{n}{\lg n}\big)$ bits of space. Further, for all possible bit-vectors of length s we maintain a table P that stores the position of the leftmost 1 in that vector. This table needs $|P| = O\big(2^s \cdot \lg s\big) = O\big(\sqrt{n} \lg \lg n\big)$ bits. Next, we show how to use these tables for constant-time access to S, and how to keep M and M' up-to-date.

When entering step i of the algorithm, we know that $S'[i{+}1] = 1$, because position $i{+}1$ has been pushed on S as the last operation of the previous step. Thus, the top of S is given by $i + 1$, and we can pop it from S (i.e., set $S'[i + 1]$ to 0). For finding the leftmost 1 in S' to the right of $j > i$ (position j has just been popped from S, meaning that $S'[j]$ has thus been set to 0), we find the leftmost 1 in j's block B_x, $x = \big\lfloor \frac{j-1}{s} \big\rfloor$, by consulting $p = P[B_x]$. If such a 1 exists (i.e., $p \neq 0$), then $xs + p - 1$ is the answer. If not ($p = 0$), we first check if $M[x] \neq +\infty$, and if so, compute $y = x + M[x]$ as the next block containing a 1; the answer is then $ys + P[y] - 1$. If not ($M[x] = +\infty$), we compute j's super-block number as $z = \big\lfloor \frac{x}{s} \big\rfloor$. Then if $M'[z] = +\infty$, the stack is empty, and otherwise, we can again use P to find the leftmost 1 in block $y = M'[z]$. In total, we can find the new top of S in constant time.

In order to keep M and M' up to date, we need to handle the operations where (1) elements are pushed on S (i.e., a 0 is changed to a 1 in S'), and (2) elements are popped from S (a 1 changed to a 0). For operation (1), let i be the element that has just been pushed on S, and let $x = \big\lfloor \frac{i-1}{s} \big\rfloor$ and $z = \big\lfloor \frac{x}{s} \big\rfloor$ be i's block and super-block, respectively. Further, let x' and z' be the block/super-block number of S's former top element (just before i is pushed), where we assume that $x' = z' = +\infty$ if the stack was empty. Then if $z \neq z'$ (the former top is in a different super-block), set $M'[z]$ to x' and $M[x]$ to $+\infty$. Otherwise ($z = z'$), if $x \neq x'$ (the former top is in a different block, but in the same super-block), just set $M[x]$ to $x' - zs$. For operation (2), nothing has to be done at all, because even if a popped index j was the last 1 in its (super-)block x, we know that all (super-)blocks before j's block do not contain a 1, so all future pointers in M and M' will jump over them. Note that this only works because elements to the right of a popped element will never be pushed again onto S. This completes the description of the $n + o(n)$-bit construction algorithm.

5.3. Lowering the Second-Order-Term. Until now, the second-order-term is dominated by the $O\big(\frac{n \lg^2 \lg n}{\lg n}\big)$ bits from Sadakane's preprocessing scheme for ± 1RMQ (§ 2.5), while all other terms (for *rank*, *select* and *findopen*) are $O\big(\frac{n \lg \lg n}{\lg n}\big)$. We show in this section a simple way to lower the space for ± 1RMQ to $O\big(\frac{n \lg \lg n}{\lg n}\big)$, thereby completing the proof of Thm. 5.8. The techniques are similar to the ones presented in § 3.3.

As in the original algorithm [52], we divide the input array E into $n' = \big\lfloor \frac{n-1}{s} \big\rfloor$ blocks of size $s = \big\lceil \frac{\lg n}{2} \big\rceil$. Queries are decomposed into at most three non-overlapping sub-queries, where the first and the last sub-queries are inside of the blocks of size s, and the middle one exactly spans over blocks. The two queries inside of the blocks are answered by table lookups using $O\big(\sqrt{n} \lg^2 n\big)$ bits, as in the original algorithm.

For the queries spanning exactly over blocks of size s, we proceed as follows. Define a new array $E'[0, n']$ such that $E'[i]$ holds the minimum of E's i'th block. E' is represented only *implicitly* by an array $E''[0, n']$, where $E''[i]$ holds the position of the minimum in the i'th block, relative to the beginning of that block. Then $E'[i] = E[is + E''[i]]$. Because E'' stores $n/\lg n$ numbers from the range $[1, s]$, the size for storing E' is thus $O\big(\frac{n \lg \lg n}{\lg n}\big)$ bits. Observe that unlike E, E' does not necessarily fulfill the ± 1-property. E' is now preprocessed for constant-time RMQs with the systematic scheme from Lemma 3.6 in § 3, using $2n' + o(n') = O\big(\frac{n}{\lg n}\big)$ bits of space.

Thus, by querying $\text{RMQ}_{E'}(i,j)$ for $1 \le i \le j \le n'$, we can also find the minima for the sub-queries spanning exactly over the blocks in E.

Two comments are in order at this place. First, the RMQ-scheme from § 3 does allow the input array to be represented implicitly, as in our case. And second, it does not use Sadakane's solution for ± 1RMQ, so there are no circular dependencies. Hence, we get:

THEOREM 5.7. *Let $E[1,n]$ be an array of numbers with the property $E[i] - E[i-1] = \pm 1$ for all $1 < i \le n$, encoded as a bit-vector $S[1,n]$ by $S[1] = 0$, and $S[i] = 1$ iff $E[i] - E[i-1] = +1$ for all $1 < i \le n$. Then there is a preprocessing scheme for RMQs on E with time complexity $\langle O(n), O(1)\rangle$ and bit-space complexity $\left[\!\left[O(\lg^3 n), |S| + O(\frac{n \lg\lg n}{\lg n}) \right]\!\right]$.*

5.4. The Final Result. We summarize this section in a final theorem.

THEOREM 5.8. *For a static array A with n elements from a totally ordered set and access time t_A, there is a preprocessing scheme for RMQs with time complexity $\langle O(n \cdot t_A), O(1)\rangle$ and bit-space complexity*

$$\left[\!\!\left[|A| + n + O\left(\frac{n \lg\lg n}{\lg n}\right), 2n + O\left(\frac{n \lg\lg n}{\lg n}\right) \right]\!\!\right] .$$

Proof. The scheme needs $2n$ bits for the DFUDS U of the 2d-Min-Heap of A, plus $O(\frac{n \lg\lg n}{\lg n})$ bits for doing rank, select, and findopen in $O(1)$ time. § 5.3 shows that the space for ± 1RMQ is also $O(\frac{n \lg\lg n}{\lg n})$ bits — note that E needs not be stored at all, as it is encoded implicitly by U (see Eq. (5.1)).

Construction space is $n + O(\frac{n \lg\lg n}{\lg n})$ for U (see § 5.2), and at most $O(\frac{n \lg\lg n}{\lg n})$ for rank, select, and findopen. The ± 1RMQ-scheme from Thm. 5.7 needs only a negligible amount of construction space. Construction time is linear for all structures. ◻

Instead of using [29] for rank and select, [27] for findopen, and Thm. 5.7 for ± 1RMQ, we could also use [48] for rank and select, and [55] for findopen and ± 1RMQ (see § 2.3–2.5), in exchange for an increased construction space:

COROLLARY 5.9. *For an array A of n objects from a totally ordered universe and access time t_A, there is a preprocessing scheme for RMQs with time complexity $\langle O(n \cdot t_A), O(1)\rangle$ and bit-space complexity*

$$\left[\!\!\left[|A| + O(n), 2n + O\left(\frac{n}{\lg^\gamma n}\right) \right]\!\!\right] ,$$

for any constant $\gamma > 0$.

6. Conclusions. We gave several space-optimal preprocessing schemes for one-dimensional range minimum queries on static arrays in constant time; the problem is thereby mostly settled in one dimension in the static case, at least in terms of worst-case complexity. There are some open questions we wish to mention: (1) dynamic problems: while linear schemes [11] for worst-case $O(1)$-LCAs in dynamic trees exist (under insertion and deletion of nodes), no such results are known for RMQs on dynamically changing arrays. The problem is here that a single modification of the array could change large portions of the underlying tree, be it the usual Cartesian Tree or our newly introduced 2d-Min-Heap. (2) better compression: Thm. 4.1 exploits repetitions in the input, but it would certainly be desirable to have schemes that are adaptive to other kinds of regularities. We are only aware of the work of Poon and

Yiu [47] that aims at this target. (3) higher dimensions: in two dimensions, there is a systematic data structure using $O(n/c)$ bits and has $O(c\lg^2 c)$ query time [8], while the lower bound states that one needs $O(n/c)$ bits for $O(c)$ query time. It would be interesting to close this gap, also in dimensions greater than 2.

Acknowledgments. We thank the anonymous referees for their valuable comments that helped to improve the presentation substantially. J.F. wishes to thank Gad M. Landau (Haifa University) for his continious reassurance that this work is going in the right direction.

REFERENCES

[1] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory Comput. Syst.*, 37:441–456, 2004.

[2] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *Proc. SODA*, pages 344–357. ACM/SIAM, 1990.

[3] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk. SSSR*, 194:487–488, 1970. English translation in *Soviet Math. Dokl.*, 11:1209–1210, 1975.

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

[5] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[6] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[7] I. Bialynicka-Birula and R. Grossi. Amortized rigidness in dynamic Cartesian Trees. In *Proc. STACS*, volume 3884 of *LNCS*, pages 80–91. Springer, 2006.

[8] G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. ESA (Part II)*, volume 6347 of *LNCS*, pages 171–182. Springer, 2010.

[9] G. Chen, S. J. Puglisi, and W. F. Smyth. LZ factorization using less time and space. *Math. Comput. Sci.*, 1(4):605–623, 2007.

[10] K.-Y. Chen and K.-M. Chao. On the range maximum-sum segment query problem. In *Proc. ISAAC*, volume 3341 of *LNCS*, pages 294–305. Springer, 2004.

[11] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.

[12] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, and T. Walen. Improved algorithms for the range next value problem and applications. In *Proc. STACS*, pages 205–216. IBFI Schloss Dagstuhl, 2008.

[13] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS*, pages 184–196. IEEE Computer Society, 2005.

[14] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):Article No. 20, 2007.

[16] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007.

[17] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, volume 6034 of *LNCS*, pages 158–169. Springer, 2010.

[18] J. Fischer. Wee LCP. *Inform. Process. Lett.*, 110(8–9):317–320, 2010.

[19] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. CPM*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006.

[20] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.

[21] J. Fischer, V. Heun, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 4213 of *LNCS*, pages 139–150. Springer, 2006.

[22] J. Fischer, V. Heun, and H. M. Stühler. Practical entropy bounded schemes for $O(1)$-range minimum queries. In *Proc. DCC*, pages 272–281. IEEE Press, 2008.

[23] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[24] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu. Radix sorting with no extra space. In *Proc. ESA*, volume 4698 of *LNCS*, pages 194–205. Springer, 2007.

[25] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM Press, 1984.

[26] A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007.

[27] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[28] L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *Proc. SODA*, pages 869–878. ACM/SIAM, 2004.

[29] A. Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.

[30] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. CPM*, volume 4009 of *LNCS*, pages 294–305. Springer, 2006.

[31] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.

[32] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[33] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. See also FOCS'80.

[34] W. K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722. IEEE Computer Society, 2009.

[35] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[36] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.

[37] D. E. Knuth. *The Art of Computer Programming Volume 4 Fascicle 4: Generating All Trees; History of Combinatorial Generation.* Addison-Wesley, 2006.

[38] H.-F. Liu and K.-M. Chao. Algorithms for finding the weight-constrained k longest paths in a tree and the length-constrained k maximum-sum segments of a sequence. *Theor. Comput. Sci.*, 407(1–3):349–358, 2008.

[39] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[40] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[41] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[42] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666. ACM/SIAM, 2002.

[43] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.

[44] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

[45] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA*, volume 5193 of *LNCS*, pages 696–707. Springer, 2008.

[46] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[47] C. K. Poon and W. K. Yiu. Opportunistic data structures for range queries. *J. Comb. Optim.*, 11(2):145–154, 2006.

[48] M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE Computer Society, 2008.

[49] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time. In *Proc. AWOC*, volume 319 of *LNCS*, pages 33–42. Springer, 1988.

[50] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM Trans. Algorithms*, 3(4):Article No. 43, 2007.

[51] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[52] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[53] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*,

5(1):12–22, 2007.

[54] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.

[55] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149. ACM/SIAM, 2010.

[56] S. Saxena. Dominance made simple. *Inform. Process. Lett.*, 109(9):409–421, 2009.

[57] J. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, 1980.

[58] T. Shibuya and I. Kurochkin. Match chaining algorithms for cDNA mapping. In *Proc. WABI*, volume 2812 of *LNCS*, pages 462–475. Springer, 2003.

[59] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, volume 4580 of *LNCS*, pages 205–215. Springer, 2007.

[60] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.

[61] H. Yuan and M. J. Atallah. Data structures for range minimum queries in multidimensional arrays. In *Proc. SODA*, pages 150–160. ACM/SIAM, 2010.