

Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays*

Johannes Fischer and Volker Heun

December 10, 2009

Abstract

Given a static array of n totally ordered objects, the range minimum query problem is to build an additional 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 $\frac{n}{c(n)}(2 + o(1))$ bits and query time $O(c(n))$, 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 optimal size $2n + o(n)$ bits and query time $O(1)$. All data structures can be constructed in linear time and almost in-place.

1 Introduction

For an array $A[1, n]$ of n objects from a totally ordered universe and two indices i and j with $1 \leq i \leq j \leq n$, a *Range Minimum Query*¹ $\text{RMQ}_A(i, j)$ returns the *position* of a minimum element in the sub-array $A[i, j]$; $\text{RMQ}_A(i, j) = \text{argmin}_{i \leq k \leq j} \{A[k]\}$. 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 [1, 22, 47], pattern matching [3, 11], string mining [20, 32], text compression [9, 42], document retrieval [40, 48, 55], trees [5, 7, 36], graphs [26, 44], bioinformatics [54], and in other types of range queries [10, 51], 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, around which this article is centered.

*Parts of this work have already been presented at the 17th Annual Symposium on Combinatorial Pattern Matching [18], at the 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies [19], at the 2008 Data Compression Conference [21], and at the 9th Latin American Theoretical Informatics Symposium [to be published 2010].

¹Sometimes also called *Discrete Range Searching* [2] or, depending on the context, *Range Maximum Query*.

Problem 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 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 [23] 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* [56] of A (see Sect. 2.2 for a formal definition of this tree). For constant-time LCA-queries, linear preprocessing schemes had already been discovered earlier [31, 52].

The problem of the solution by Gabow et al. [23], and also that of subsequent simplifications [2, 5, 7], 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 [6, 13, 25, 34, 39, 50], dictionaries [43, 45], and strings [14, 15, 29, 30, 46, 49], being well aware of the fact that this list is far from complete.

Our results for RMQs are situated in the field of succinct and compressed data structures. But before detailing our contributions, we first classify and summarize existing schemes for $O(1)$ -RMQs.

1.1 Previous Solutions for RMQ

In accordance with common nomenclature [24], preprocessing schemes for $O(1)$ -RMQs can be classified into two different types: *systematic* and *non-systematic*. 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 [2, 5, 7, 23] are systematic. They are appropriate in the following situations:

- If $|A|$, the number of bits to store A , is small enough to be dominated by the space for the RMQ-scheme (e.g., $|A| = O(n)$),.
- If $\omega(1)$ query time suffices, and whole blocks of the input array are to be *scanned* when answering the queries.

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

- Perhaps most importantly, when the actual *values* of the minima matter, or if A is needed by the algorithm for different purposes, such that the input array has to be kept in memory anyway.

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:

1. In some applications, e.g., in algorithms for document retrieval [40, 48] or position restricted substring matching [11], 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, just for obtaining the final answer to the RMQs, as in the case of systematic schemes.
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. As a prominent example, in string processing RMQ is often used in conjunction with the array of *longest common prefixes* of lexicographically consecutive suffixes, the so-called *LCP-array* [37]. However, storing the LCP-array efficiently in $2n + o(n)$ bits [47] or even less [17, 22] increases the access-time to the time needed to retrieve an entry from the corresponding *suffix array* [37], which is $\Omega(\lg^\varepsilon n)$ (constant $\varepsilon > 0$) at the very best if the suffix array is also stored in compressed form [29, 46]. 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 [22] (where for different reasons 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, where, besides the final space consumption, in the second column we list the peak space consumption at construction time of each scheme, which sometimes differs from the former term.

1.1.1 Systematic Schemes

Most systematic schemes are based on the Cartesian Tree [56], the only exception being the scheme due to Alstrup et al. [2]. All direct schemes [2, 5, 47] 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 three rows of Tbl. 1 have the same theoretical guarantees (namely $O(n \lg n)$ bits of space), with Bender et al.’s scheme [5] being less complex than the previous ones, and Alstrup et al.’s [2] being even simpler (and most practical). For systematic schemes, no lower bound on space is known.³

An important special case is Sadakane’s $n + o(n)$ -bit solution [47] for ± 1 RMQ, where it is assumed that subsequent array-elements differ by only 1; we will describe it in greater detail in Sect. 2.4.

³The claimed lower bound of $2n + o(n) + |A|$ bits under the “min-probe-model” [19] turned out to be wrong, as was kindly pointed out to the authors by S. Srinivasa Rao (personal communication, November 2007).

Table 1: Preprocessing schemes for $O(1)$ -RMQs, where $|A|$ denotes the space of the (read-only) input array. (Set $c(n) = O(1)$ to obtain $O(1)$ query time in Thm. 9.) Space is measured in bits.

reference	construction space	final space	comments
[7, 31, 52]	$O(n \lg n) + A $	$O(n \lg n) + A $	via LCA in Cartesian Trees
[5]	$O(n \lg n) + A $	$O(n \lg n) + A $	simpler than previous schemes
[2]	$O(n \lg n) + A $	$O(n \lg n) + A $	not based on Cartesian Trees
Thm. 9	$\frac{2n}{c(n)} + o(n) + A $	$\frac{2n}{c(n)} + O(\frac{n \lg \lg n}{c(n) \lg n}) + A $	query time $O(c(n))$
Thm. 10	$nH_k + o(n) + A $	$nH_k + O(\frac{n}{\lg n}(k \lg \sigma + \lg \lg n)) + A $	H_k empirical entropy of A
[47]	$o(n) + n$	$O(n \lg^2 \lg n / \lg n) + n$	only for ± 1 RMQ
Thm. 15	$o(n) + n$	$O(n \lg \lg n / \lg n) + n$	
[48]	$O(n \lg n) + A $	$4n + O(n \lg^2 \lg n / \lg n)$	only non-systematic scheme
Thm. 17	$3n + o(n) + A$	$2n + O(n \lg \lg n / \lg n)$	final space optimal

1.1.2 Non-Systematic Schemes

The only existing scheme is due to Sadakane [48] and uses $4n + o(n)$ bits. It is based on the balanced-parentheses-encoding (BPS) [39] of the Cartesian Tree of the input array A , and a $o(n)$ -bit scheme for $O(1)$ -LCA computation therein [47]. 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 (basically because there is no way to distinguish between left and right nodes in the BPS of a tree); 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 Sect. 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 $\lg(\binom{2n}{n}/(n+1)) = 2n - \Theta(\lg n)$ bits [33].
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 [8].

1.2 Our Results

We present preprocessing schemes for range minimum queries of yet unseen small size; see again Tbl. 1 for a summary and comparison.

In the systematic setting, we first give a simple scheme that uses only $\frac{n}{c(n)} + O(\frac{n \lg \lg n}{c(n) \lg n})$ bits on top of A (Thm. 9). Here, $c(n)$ can be any positive integer function. If $c(n) = O(1)$, then Thm. 9 gives optimal constant query time with $O(n)$ space, where the big- O constant can be made arbitrarily small. This is the first systematic scheme with linear bit-complexity. The scheme from Thm. 9 builds on a sophisticated enumeration of binary trees (Sect. 3.3), which might be useful also for different purposes.

We then show in Thm. 10 how to compress the scheme from Thm. 9 into a data structure of size $nH_k + O(\frac{n}{\lg n}(k \lg \sigma + \lg \lg n)) + |A|$ bits, simultaneously over all $k = o(\lg_\sigma n)$. Here, H_k denotes the empirical entropy of order k [38] of the input array that consists of σ different elements. The value nH_k is a common measure for the compressibility of data structures [41], as it provides an upper bound on the size of the output of any compressor that encodes a symbol based on the k preceding characters. Note that the input array itself could also be compressed with recent schemes to provide $O(1)$ -access to its elements [16, 28, 49], such that $|A| = nH_k + o(n)$. This yields a truly compressed scheme for $O(1)$ -RMQs.

We also improve on the space for ± 1 RMQ by giving a scheme that needs only $O(\frac{n \lg \lg n}{\lg n})$ bits on top of the n bits for storing the input array (Thm. 15), as opposed to $O(\frac{n \lg^2 \lg n}{\lg n})$ bits needed by the only previous solution [47]. An interesting by-product of this is that this also lowers the space for LCA-computation in succinctly encoded trees (see Cor. 16), as these methods are all based on ± 1 RMQ [34, 47].

We finally focus on the non-systematic setting, where we show a preprocessing scheme of asymptotically *optimal* size $2n + O(\frac{n \lg \lg n}{\lg n})$ bits and $O(1)$ query time (Thm. 17). Compared to Sadakane’s $4n + O(\frac{n \lg^2 \lg n}{\lg n})$ -bit solution, the critical reader might call this “lowering the constants” or “micro-optimization,” but we believe that data structures using the smallest possible space are of high importance, both in theory and in practice. And indeed, there are many examples of this in literature: for instance, Munro and Raman [39] give a $2n + o(n)$ -bit solution for representing ordered trees, although a $O(n)$ -bit solution (roughly $10n$ bits [39]) had already been known for more than a decade before [33]. Also, halving the multiplicative constant of the first order term also almost halves the space for second-order terms, which are non-negligible in practice.

Construction *time* is linear for all our methods. 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. 9, 10, and 15 can be constructed in-place (apart from negligibly small terms), and the scheme from Thm. 17 needs only one additional bit-vector of length n , yielding the first construction algorithm for $O(1)$ -RMQs in the non-systematic setting with $O(n)$ working space. This is a significant improvement over the $O(n \lg n)$ -bit construction algorithm for Sadakane’s non-systematic scheme [48]. Note again that as the space for storing A is not necessarily $\Theta(n \lg n)$; for example, if the numbers in A are integers in the range $[1, \lg^{O(1)} n]$, 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.

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 $\{\ell, \ell + 1, \dots, r\}$.

Following Bender et al.'s notation [5], 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 $\llbracket s(n), t(n) \rrbracket$ if $s(n)$ is the space consumption at construction time, and $t(n)$ is the final space of the data structure.

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 [56] 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 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 $\mathcal{C}^{\text{can}}(A)$. Note also that this order results in unique answers to RMQs, because the minimum is unique.

Gabow et al. [23] give an algorithm for constructing $\mathcal{C}^{\text{can}}(A)$ incrementally, which we summarize as follows. Let $\mathcal{C}_i^{\text{can}}(A)$ be the Canonical Cartesian Tree for $A[1, i]$. Then $\mathcal{C}_{i+1}^{\text{can}}(A)$ is obtained by climbing up from the rightmost node of $\mathcal{C}_i^{\text{can}}(A)$ to the root, thereby finding the position where $A[i + 1]$ belongs. To be precise, let v_1, \dots, v_k be the nodes on the rightmost path in $\mathcal{C}_i^{\text{can}}(A)$ with labels ℓ_1, \dots, ℓ_k , respectively, where v_1 is the root, and v_k is the rightmost node. Let m be defined such that $A[\ell_m] \leq A[i + 1]$ and $A[\ell_{m+1}] > A[i + 1]$ (hence $A[\ell_{m'}] > A[i + 1]$ for all $m < m' \leq k$). To build $\mathcal{C}_{i+1}^{\text{can}}(A)$, 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 $\mathcal{C}^{\text{can}}(A)$.

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: $\text{rank}_1(S, i)$ gives the number of 1's in the prefix $S[1, i]$, and $\text{select}_1(S, i)$ gives the

position of the i 'th 1 in S , reading S from left to right ($1 \leq i \leq n$). Operations $rank_0(S, i)$ and $select_0(S, i)$ are defined analogously for 0-bits. There are data structures of size $O(\frac{n \lg \lg n}{\lg n})$ bits in addition to S that support rank- and select-operations in $O(1)$ time [27]. These data structures are also applicable to sequences of parentheses, interpreting a '(' as a '1', and a ')' as a '0'.

If the number of 1's in S is $O(n/\lg n)$, the bit vector S can be encoded in $O(\frac{n \lg \lg n}{\lg n})$ space, including structures for constant-time rank and select, using the *fully indexable dictionary* by Raman et al. [45].

2.4 Data Structures for ± 1 RMQ

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 $E[1] + rank_1(S, i) - rank_0(S, i) + 1 = E[1] + i - 2rank_0(S, i) + 1$. Under this setting, Sadakane [47] shows how to support RMQs on E in $O(1)$ time, using S and additional structures of size $O(\frac{n \lg^2 \lg n}{\lg n})$ bits. We denote this restricted version of RMQ by ± 1 RMQ.

2.5 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 $rank_1(B, i) - rank_0(B, i) = rank_1(B, j) - rank_0(B, j)$. The $findopen$ -operation can be computed in constant time [39]; the most space-efficient data structure for this needs $O(\frac{n \lg \lg n}{\lg n})$ bits on top of B [25].

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 [6]. If T is a leaf, U is given by '()'. Otherwise, if the root of T has w subtrees T_1, \dots, T_w in this order, U is given by the juxtaposition of $w + 1$ '('s, a ')', and the DFUDS's of T_1, \dots, 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 ± 1 RMQ-operations, in particular moving to the parent node [6], and finding the *lowest common ancestor* $LCA(u, v)$ of two nodes u and v [34], 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 $B_1, \dots, B_{\lceil n/s \rceil}$ of size $s = \lceil \frac{\lg n}{4} \rceil$, where $B_i = A[(i-1)s + 1, is]$.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* 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 inside of these three sub-queries. See also the top half of Fig. 4 on p. 15, where the in-block-queries are labeled by ① and ③, and the out-of-block-query by ②.

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 Sect. 3.2 with a novel variant of the Four-Russians-Trick [4] (precomputation of all answers for sufficiently small instances). However, also our solution to the long queries (Sect. 3.4) 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 2. *Let B_x and B_y be two blocks of size s . Then $\text{RMQ}_{B_x}(i, j) = \text{RMQ}_{B_y}(i, j)$ for all $1 \leq i \leq j \leq s$ if and only if $\mathcal{C}^{\text{can}}(B_x) = \mathcal{C}^{\text{can}}(B_y)$.*

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

1. The minimum under “ \prec ” occurs at the same position m , i.e., $\text{argmin } B_x = \text{argmin } B_y = m$.
2. For all i', j' with $1 \leq i' \leq j' < m$: $\text{RMQ}_{B_x[1, m-1]}(i', j') = \text{RMQ}_{B_y[1, m-1]}(i', j')$.
3. For all i', j' with $m < i' \leq j' \leq s$: $\text{RMQ}_{B_x[m+1, s]}(i', j') = \text{RMQ}_{B_y[m+1, s]}(i', j')$.

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

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 $C_s = 4^s / (\sqrt{\pi} s^{3/2}) (1 + O(s^{-1}))$ possible blocks, where C_s is the s 'th Catalan Number (number of rooted trees on s nodes). So if we have a table $P[1, C_s][1, \binom{s}{2}]$ that stores the answers to all $\binom{s}{2}$ proper RMQs inside of all C_s possible blocks, knowing the type $t(B_i)$ of block B_i would allow us to look-up the answer in $P[t(B_i)][\cdot]$. Here, by the *type* of B_i we mean a description of B_i 's Canonical Cartesian Tree $\mathcal{C}^{\text{can}}(B_i)$ that identifies it among all Cartesian Trees on s elements.

⁴In fact, any block size $s = \lceil \frac{\lg n}{n+\delta} \rceil$ 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_j

Input: a block B_j of size s

Output: the type of B_j , as defined by Eq. (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  $R[q + i - s] > B_j[i]$  do
6      $N \leftarrow N + C_{(s-i)q}$                                  $\{\text{add number of skipped paths}\}$ 
7      $q \leftarrow q - 1$                                         $\{\text{remove node from rightmost path}\}$ 
8   endw
9    $R[q + i + 1 - s] \leftarrow B_j[i]$                          $\{B_j[i] \text{ is new rightmost node}\}$ 
10 endfor
11 return  $N$ 

```

It thus remains to show how to compute the types of the $\lceil n/s \rceil$ blocks B_j occurring in A in linear time; i.e., how to fill an array $T[1, \lceil n/s \rceil]$ such that $T[j]$ is the type of block B_j . Lemma 2 implies that there are only C_s different types of blocks, so we are looking for a surjection

$$t : \mathcal{A}_s \rightarrow [0 : C_s - 1], \text{ and } t(B_i) = t(B_j) \text{ iff } \mathcal{C}^{\text{can}}(B_i) = \mathcal{C}^{\text{can}}(B_j), \quad (1)$$

where \mathcal{A}_s is the set of arrays of size s . We now claim that Alg. 1 computes a function defined by (1) in $O(s)$ time. It makes use of the so-called *ballot numbers* C_{pq} [35], defined by

$$C_{00} = 1, C_{pq} = C_{p(q-1)} + C_{(p-1)q}, \text{ if } 0 \leq p \leq q \neq 0, \text{ and } C_{pq} = 0 \text{ otherwise.} \quad (2)$$

It can be proved that a closed formula for C_{pq} is given by $\frac{q-p+1}{q+1} \binom{p+q}{p}$ [35], which immediately implies that C_{ss} equals the s 'th Catalan number C_s .

Lemma 3. *Algorithm 1 correctly computes the type of a block B_j of size s in $O(s)$ time, i.e., it computes a function satisfying the conditions given in (1).*

Proof. Intuitively, Alg. 1 simulates the algorithm for constructing $\mathcal{C}^{\text{can}}(B_j)$ given in Sect. 2.2 and “implements” an enumeration of binary trees, described in more detail in the following section. Intuitively, array $R[1, s+1]$ simulates the stack containing the labels of the nodes on the rightmost path of the partial Canonical Cartesian Tree $\mathcal{C}_i^{\text{can}}(B_j)$, 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 i th iteration of the outer for-loop, then ℓ_i equals the number of elements that are removed from the rightmost path when going from $\mathcal{C}_{i-1}^{\text{can}}(B_j)$ to $\mathcal{C}_i^{\text{can}}(B_j)$. Because one cannot remove more elements from this rightmost path than are currently on it, the sequence $\ell_1 \ell_2 \dots \ell_s$ satisfies

$$\sum_{k=1}^i \ell_k < i \text{ for all } 1 \leq i \leq s. \quad (3)$$











array A	$\mathcal{C}^{\text{can}}(A)$	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	$C_{13} + C_{02} = 4$

Table 2: Example-arrays of length 3, their Cartesian Trees, and their corresponding paths in the graph in Fig. 1. The last column shows how to calculate the index of $\mathcal{C}^{\text{can}}(A)$ in an enumeration of all Cartesian Trees.

The following section shows that such sequences uniquely characterize binary trees (and hence Cartesian Trees), and that the additions performed in line 6 of Alg. 1 yield a unique index in an enumeration of all binary trees (as they are exactly the additions in (4), of which bijectivity will be shown). We already conclude here that Alg. 1 computes a function defined by Eq. (1). ■

3.3 A New Code for Binary Trees

We now show that a sequence ℓ_1, \dots, ℓ_s satisfying (3) is a unique encoding of a binary trees on s nodes, and how this encoding is useful for computing an index in an enumeration of all binary trees. Throughout this section, the reader is encouraged to peek at Tbl. 2, where most of the concepts are illustrated.

3.3.1 Bijection between Binary Trees and $\ell_1\ell_2\dots\ell_s$

Let $\ell_1\ell_2\dots\ell_s$ be a sequence of s natural numbers satisfying (3). First observe that each binary tree is a Canonical Cartesian Tree for some array A with elements from a totally ordered set of sufficient size; and, by definition, each Canonical Cartesian Tree is also a binary tree. This implies that a binary tree T can, in principle, be represented by an array A : simply choose the numbers in A such that A 's Canonical Cartesian Tree is equal to T .

The crucial fact to observe now is that the actual *numbers* in A do not affect the topology of the Cartesian Tree, as it is only determined by the *positions* of the minima. Recall the algorithm for constructing the Canonical Cartesian Tree in Sect. 2.2. In step i it traverses the rightmost path of $\mathcal{C}_{i-1}^{\text{can}}(A)$ from the rightmost node towards the root, and removes some elements from it. Now let ℓ'_i be the number of nodes that are removed from the rightmost path when going from $\mathcal{C}_{i-1}^{\text{can}}(A)$ to $\mathcal{C}_i^{\text{can}}(A)$. Because one cannot remove more elements from the rightmost path than one has inserted before, and because each element is removed at most once, we have $\sum_{k=1}^i \ell'_k < i$ for all $1 \leq i \leq s$. Thus, the sequence $\ell'_1\dots\ell'_s$ satisfying (3) completely describes the output of the algorithm for constructing the Canonical Cartesian Tree, and thus uniquely represents a binary tree with s nodes.

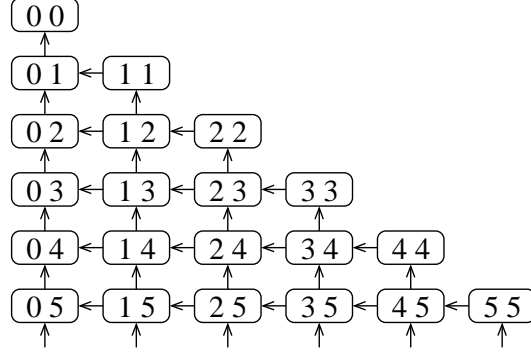


Figure 1: The infinite graph arising from the definition of the Ballot Numbers. Its vertices are $\boxed{p \ q}$ for all $0 \leq p \leq q$. There is an edge from $\boxed{p \ q}$ to $\boxed{(p-1) \ q}$ if $p > 0$ and to $\boxed{p \ (q-1)}$ if $q > p$.

On the other hand, given a binary tree with s nodes, we can easily find a sequence of s numbers satisfying (3), by first constructing an array A whose Cartesian Tree equals the given tree, and then running the construction algorithm for $\mathcal{C}^{\text{can}}(A)$. In total, we have a bijective mapping from binary trees to sequences $\ell_1 \ell_2 \dots \ell_s$ satisfying (3).

With some variations, this bijection is already described in earlier works [12].

3.3.2 Bijection between $\ell_1 \ell_2 \dots \ell_s$ and $[0 : C_s - 1]$

Let \mathcal{L}_s be the set of sequences $\ell_1 \ell_2 \dots \ell_s$ satisfying (3). Assume now we have defined a way to enumerate \mathcal{L}_s in some order. Then for a given $l \in \mathcal{L}_s$ we might wish to compute the position (or index) of l in this enumeration. We will show next how to compute this index directly.

Recall the definition of the Ballot Numbers (2) and look at the infinite directed graph shown in Fig. 1: C_{pq} equals the number of paths from $\boxed{p \ q}$ to $\boxed{0 \ 0}$, because of (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 $\boxed{0 \ 0}$.

The sequence $\ell_1 \dots \ell_s$ corresponds to a path from $\boxed{s \ s}$ to $\boxed{0 \ 0}$ in Fig. 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 $\ell_1 \dots \ell_s$ 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 $\boxed{0 \ 0}$. Because there are $C_{ss} = C_s$ such paths, the task of computing the index of $\ell_1 \dots \ell_s$ has thus become the task of finding a bijection from \mathcal{L}_s to $[0 : C_s - 1]$.

We claim that the desired bijection is given by the function

$$f : \mathcal{L}_s \rightarrow [0 : C_s - 1] : \ell_1 \ell_2 \dots \ell_s \mapsto \sum_{i=1}^s \sum_{0 \leq j < \ell_i} C_{(s-i)(s-j-\sum_{k < i} \ell_k)} . \quad (4)$$

This formula is actually not so hard to understand when viewed from an algorithmic standpoint.

The important thing to note is that it simulates a walk from $\boxed{s \ s}$ to $\boxed{0 \ 0}$ in the graph in Fig. 1. In step i of the outer sum, the current position in the graph is $\boxed{(s-i+1) \ (s-q)}$, with $q = \sum_{k < i} \ell_k$ being the total number of upwards steps that have already been made *before* step i . Now recall that ℓ_i corresponds to moving ℓ_i steps upwards, and then one step to the left. So for each of the ℓ_i upward steps, the inner sum increments the value of the function by the number of paths that have been “skipped” by going upwards. This is exactly $C_{(s-i)(s-q-j)}$, the value of the cell to the left of the current one if j runs through the upward steps. The effect of this addition is that paths going to the left from the current position are assigned lower numbers than paths going upwards from the current position. This implies that the sequence $\sigma = 00 \dots 0$ will be assigned the number $f(\sigma) = 0$, $\tau = 011 \dots 1$ will get $f(\tau) = C_s - 1$, and other sequences will receive numbers between 0 and $C_s - 1$. As an example, the index of $\ell = 0102$ is $f(\ell) = C_{35} + (C_{14} + C_{13})$, the summand outside the parenthesis coming from $\ell_2 = 1$, and the two summands inside coming from $\ell_4 = 2$. See also Tbl. 2.

Let us now prove that the function f defined by (4) is actually bijective. 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. 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 $[0 : C_s - 1]$, 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:

$$C_{pq} = \sum_{p \leq q' \leq q} C_{(p-1)q'} \text{ for } 1 \leq p \leq q \quad (5)$$

$$C_{(p-1)p} = 1 + \sum_{0 \leq i < p-1} C_{(p-i-2)(p-i)} \text{ for } p > 0 \quad (6)$$

Eq. (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 $\boxed{(p-1) \ q}$, $\boxed{(p-1) \ (q-1)}$, \dots , $\boxed{(p-1) \ p}$, as all paths starting at $\boxed{p \ q}$ can be expressed as the disjoint union over those paths. The formal proof of (5) is by induction on q : for $q = 1$, $C_{11} = C_{10} + C_{01} = 0 + 1 = 1$ by (2), and (5) gives $C_{11} = \sum_{1 \leq q' \leq 1} C_{0q'} = C_{01} = 1$. For the induction step, let the induction hypothesis (IH) be $C_{p(q-1)} = \sum_{p \leq q' \leq q-1} C_{(p-1)q'}$ for all $1 \leq p \leq q-1$. Then

$$C_{pq} \stackrel{(2)}{=} C_{p(q-1)} + C_{(p-1)q} \stackrel{(\text{IH})}{=} \sum_{p \leq q' \leq q-1} C_{(p-1)q'} + C_{(p-1)q} = \sum_{p \leq q' \leq q} C_{(p-1)q'}.$$

Eq. (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 (2), and (6) yields $C_{01} = 1 + \sum_{0 \leq i < 0} C_{(-i-2)(-i)} = 1$, as the sum is empty. For the induction step, let the induction hypothesis be $C_{(p-2)(p-1)} = 1 + \sum_{0 \leq i < p-2} C_{(p-1-i-2)(p-1-i)}$. Then

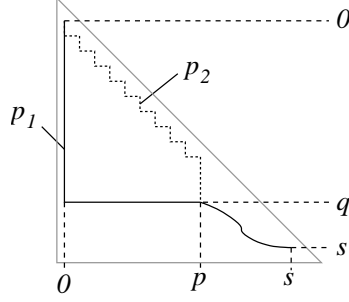


Figure 2: Smallest (p_1) and largest (p_2) paths (under f) among the paths that are equal up to

$\boxed{p \ q}$.

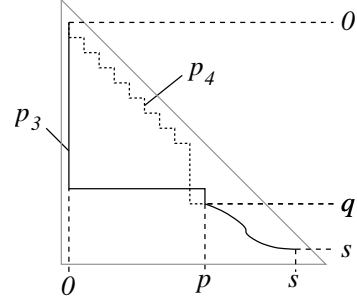


Figure 3: p_3 is the next-largest path (under f) after p_4 among those paths that are equal up to

$\boxed{p \ q}$.

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

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

Lemma 4. For $0 \leq p \leq q \leq s$ 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. 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 $0 \leq i+1 \leq j-1$ (this follows directly from $C_{(i+1)j} = C_{ij} + C_{(i+1)(j-1)}$ and the fact that for $0 \leq i+1 \leq j-1$, $C_{(i+1)(j-1)} > 0$). So taking the rightmost (i.e. highest) possible value from each row $q' \leq q$ must yield the highest sum (note that f can add at most one Ballot Number from each row $q' \leq q$). ■

Lemma 5. Let p , q , and $\mathcal{P}_s^{\overline{pq}}$ be as in Lemma 4. 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). 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

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

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

Then

$$\begin{aligned} f(p_3) &= S + C_{(p-1)q} \\ &= S + \sum_{p-1 \leq q' \leq q} C_{(p-2)q'} && \text{(by (5))} \\ &= S + \sum_{p \leq q' \leq q} C_{(p-2)q'} + C_{(p-2)(p-1)} \\ &= S + \sum_{p \leq q' \leq q} C_{(p-2)q'} + 1 + \sum_{0 \leq i < p-2} C_{(p-i-3)(p-i-1)} && \text{(by (6))} \\ &= f(p_4) + 1 . \end{aligned}$$

■

This gives us all the tools for

Lemma 6. Function f defined by (4) is a bijective mapping from \mathcal{L}_s to $[0 : C_s - 1]$.

Proof. Injectivity can be seen as follows: different paths p_3 and p_4 must have one point $\boxed{p \ q}$ where one path (w.l.o.g. p_3) continues with an upwards step, and the other (p_4) with a leftwards step. Combining Lemmas 4 and 5, f assigns a larger value to p_3 than to p_4 , regardless of how these paths continue afterwards.

Surjectivity follows from the fact that the smallest path receives number 0, and the largest path (crawling along the main diagonal) receives number $\sum_{0 \leq i < s-1} C_{(s-i-2)(s-i)} = C_{(s-1)s} - 1$ (by (6)). But $C_{(s-1)s} = C_{ss} = C_s$, and as there are exactly C_s paths from $\boxed{s \ s}$ to $\boxed{0 \ 0}$, all being assigned different numbers, there must be a path $p \in \mathcal{P}_s$ such that $f(p) = x$ for every $x \in [0 : C_s - 1]$. ■

We summarize this in the following

Lemma 7. For a binary tree T with s nodes, with (4) we can compute in $O(s)$ time the index of T in an enumeration of all binary trees with s nodes, with the help of a sequence of s numbers ℓ_1, \dots, ℓ_s that satisfy (3). ■

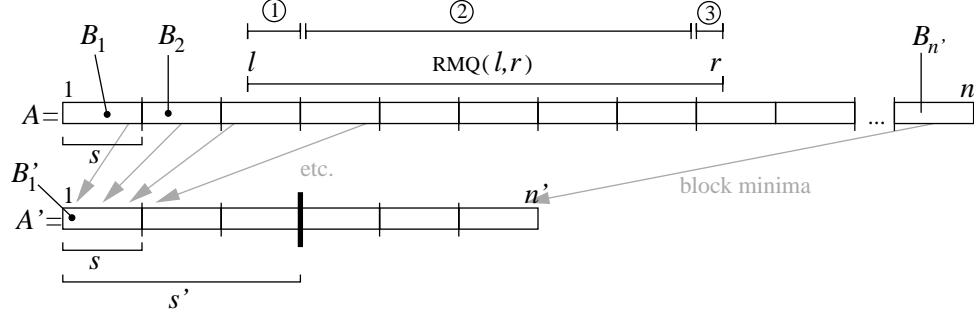


Figure 4: The input array A is divided into blocks $B_1, \dots, B_{n'}$ of size s , and a query $\text{RMQ}_A(i, j)$ is divided into three sub-queries ①–③. Array A' stores the block-minima, and is again divided into blocks $B'_1, \dots, B'_{\lceil n'/s \rceil}$ of size s . Further, s of these blocks are grouped into super-blocks of size s' .

3.4 Preprocessing for Out-of-Block-Queries

It remains to show how the out-of-block-queries are answered (queries aligned with block boundaries). Proceeding as in previous schemes [2, 5] would result in a super-linear bit space $O(n \lg n)$, so we need a different approach. In principle, we could adapt the solution of the non-systematic scheme due to Sadakane [48] to our setting, which would result in $O(\frac{n \lg^2 \lg n}{\lg n})$ bits of space. As this term can be quite large in practice, we opt for a less space consuming variant, explained as follows (see also Fig. 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 $A'[1, n']$, 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 Sect. 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 lookup-table P as in Sect. 3.2.

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 [47, 48]. 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 $M''[1, \lceil n'/s' \rceil][0, \lfloor \lg(\lceil n'/s' \rceil) \rfloor]$, 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 $M''[i][j] = \text{argmin}_{k \in \{M''[i][j-1], M''[i+2^{j-1}][j-1]\}} \{A'[k]\}$.

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 $p = \lfloor \lg(j-i+1) \rfloor$, the corresponding sub-ranges are $[i, i+2^p-1]$ and $[j-2^p+1, j]$. Hence, the minimum of B''_i, \dots, B''_j can be found by $\text{argmin}_{k \in \{M''[i][p], M''[j-i+1][p]\}} \{A'[k]\}$.

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 $M'[1, \lceil n'/s \rceil][0, \lfloor \lg(\lceil s'/s \rceil) \rfloor]$, where $M'[i][j]$ stores the position of the minimum of blocks $B'_i \dots, B'_{i+2^j-1}$ (minimum in $A'[(i-1)s+1, (i+2^j-1)s]$). 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'').

3.5 Space Analysis

Let us now analyze the space occupied by the scheme given in Sect. 3.2–3.4. We start with the structures from Sect. 3.2. Recall that the block size is $s = \lceil \frac{\lg n}{4} \rceil$. To store the type of each block, array T has length $\lceil n/s \rceil = \lceil 4n/\lg n \rceil$, and because of Lemma 2, the numbers are all within $O(4^s/s^{3/2})$. This means that the number of bits to encode T is

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

To analyze the space of the lookup-table P , by Lemma 2 we know that P has only $O(\frac{4^s}{s^{3/2}})$ rows, one for each possible block-type. For each type we need to precompute $\text{RMQ}(i, j)$ for all $1 \leq i \leq j \leq s$, so the number of columns in P is $O(s^2)$. If we use the method described by Alstrup et al. [2] to represent the answers to all RMQs inside one block, this takes $O(s \cdot s)$ bits of space for each possible block.⁵ 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 Sect. 3.4. First note that array A' need not be stored verbatim; instead, we only have to store the *positions* of the minima in the i 'th block, and this only relative to the beginning of the block. Hence, letting $A''[i] = \text{RMQ}_{B_i}(1, s)$, we can simulate array $A'[1, n']$ by $A'[i] = A[A''[i] + (i-1)s]$. As the numbers in A'' are in the range $[1, s]$, the number of bits needed for A' is thus

⁵We remark that the usage of Alstrup et al.'s method [2] for the precomputation of the in-block queries would not be necessary to achieve the $o(n)$ space bound, but it certainly saves some space.

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

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

$$\begin{aligned}
|M''| &= O\left(\frac{n'}{s'} \lg\left(\frac{n'}{s'}\right) \cdot \lg n\right) \\
&= O\left(\frac{n'}{\lg^2 n} \lg\left(\frac{n'}{\lg^2 n}\right) \cdot \lg n\right) \\
&= O\left(\frac{n}{\lg^3 n} \lg n \cdot \lg n\right) \\
&= O\left(\frac{n}{\lg n}\right).
\end{aligned}$$

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

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

The dominating second-order-term of the scheme are the $O(\frac{n \lg \lg n}{\lg n})$ bits for storing A'' . For constructing the scheme, we only need the space of the final scheme, plus $O(\lg n \lg \lg n)$ bits for array R in Alg. 1, and $O(\lg^3 n)$ bits for the Ballot-Numbers C_{pq} , which are all within $O(\frac{n \lg \lg n}{\lg n})$.

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 8. *For a static array A with n elements from a totally ordered set and access time t_A , there exists a preprocessing scheme for RMQ with time complexity $\langle O(n), O(t_A) \rangle$ and bit-space complexity $\left\lceil 2n + O(\frac{n \lg \lg n}{\lg n}) + |A|, 2n + O(\frac{n \lg \lg n}{\lg n}) + |A| \right\rceil$. ■*

3.6 The Final Result

Finally, we show how to lower the leading $2n$ -bit term from Lemma 8 to $2n/c(n)$ (arbitrary positive integer function c). 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 8 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 . This leads us to the following theorem.

Theorem 9. *For a static array A with n elements from a totally ordered set and access time t_A , there is a preprocessing scheme for RMQ with time complexity $\langle O(n), O(c(n) \cdot t_A) \rangle$ and space complexity $\left\lceil \frac{2n}{c(n)} + O\left(\frac{n \lg \lg n}{c(n) \lg n}\right) + |A|, \frac{2n}{c(n)} + O\left(\frac{n \lg \lg n}{c(n) \lg n}\right) + |A| \right\rceil$. Here, c can be any function from \mathbb{N}^+ to \mathbb{N}^+ . ■*

Function $c(\cdot)$ can be constant, in which case Thm. 9 gives optimal $O(1)$ query time. Notwithstanding, there are also applications where $\omega(1)$ query time suffices [22]. In this case, Thm. 9 is stronger than the currently best solution for RMQs with sublinear space [22, Lemma 2]. For example, we can achieve $O(\lg \lg n)$ query time with $O(\frac{n}{\lg n})$ space on top of A (setting $c(n) = \lceil \lg \lg n \rceil$ in Thm. 9), whereas [22] would give $O(\lg \lg n \cdot \lg^2 \lg \lg n)$ 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 [53].

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 in 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 first show that the simple *text*-encoding by Ferragina and Venturini [16] is also effective for our type-array T , and then give a new variant of this scheme that is conceptually even simpler. 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 a first entropy-bounded preprocessing scheme for RMQs. The basis is the RMQ-algorithm from Lemma 8, so 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 , 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 : $\mathcal{T} = \{T[i] : i \in [1 : \lceil n/s \rceil]\}$.
- Sort the elements from \mathcal{T} by decreasing frequency of occurrence in T and let $r(B_j)$ be the rank of block B_j in this ordering.

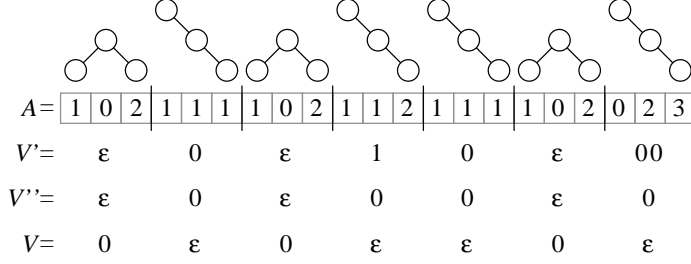


Figure 5: Illustration to the compressed representation of RMQ-information. On top of each block of size $s = 3$ we display its Canonical Cartesian Tree. The final encoding can be found in the row labeled V ; the rows labeled V' and V'' are solely for the proof of Thm. 10.

- 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. 5 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 encoding 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 10. *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 RMQ with time complexity $\langle O(n), O(t_A) \rangle$ and bit-space complexity $\left\lceil nH_k + o(n) + |A|, \min \left\{ nH_k(A) + O\left(\frac{nk \lg \sigma}{\lg n}\right), 2n \right\} + O\left(\frac{n \lg \lg n}{\lg n}\right) + |A| \right\rceil$, simultaneously over all $k \in o(\lg_\sigma n)$.*

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. 5, 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. 3.1] 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 [28] 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 $b = O(\lg n)$).

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. 5 for an example. Now our encoding V cannot be longer than V'' , as it assigns even shorter codewords to more frequent types, and therefore obeys the “golden rule of data compression.” Finally, by noting that the compressed V is never larger than the uncompressed T , we can conclude that $|V| = \min \left\{ nH_k(A) + O(\frac{nk \lg \sigma}{\lg n}), 2n \right\}$.

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 $|D| = O(n/s \cdot \lg s') = O(\frac{n \lg \lg n}{\lg n})$ bits. The size of table D' is simply $|D'| = n/s' \cdot \lg |V| = O(\frac{n}{\lg n})$.

Because arrays A'' , M' and M'' of the RMQ-scheme from Sect. 3 are still needed for answering the out-of-block-queries, the leading second order term remains $O(\frac{n \lg \lg n}{\lg n})$. The claim on the final space follows.

We finally show that the scheme can be constructed within its final 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 $O(C_s \cdot \lg n) = O(\sqrt{n/\lg n})$ bits. We then sort this array in-place and assign the codes according to the frequency, needing at most additional $O(\sqrt{n/\lg n})$ bits. A second scan over A constructs the block types *again*, and directly writes the output stream V . ■

This analysis is quite coarse and certainly “wastes” some space; however, it matches the currently best known results for storing the array A itself in compressed form while still being able to access any $O(\lg n)$ contiguous bits in constant time under the RAM model [16, 28, 49]. Therefore, even if we proved a better bound on the space of our compressed type array, the space needed for storing A itself would be asymptotically larger.

4.2 Simpler Decompression

We present a variant of the above compression scheme that is conceptually even simpler, because it does not need the concepts of super-blocks. We emphasize that our new ideas can also be applied to the original string compression scheme [16], yielding an “even simpler” storage scheme for strings, although the original compression scheme [16] is already quite simple!

The difference to Sect. 4.1 is as follows. Instead of storing the beginnings of blocks and super-blocks in tables D and D' , needed to “recover” the encodings $c(B_j)$ in V , we encode the length of the codewords $c(B_j)$ in *unary* as $0^{|c(B_j)|}1$. Here, 0^x denotes the juxtaposition of x 0’s. For $j \in [1 : \lceil n/s \rceil]$ in this order, these unary encodings are concatenated in a new bit-vector W . We prepare W for constant-time *select*₁-queries. Then the beginning p of the encoding of block B_j in V can be found by $p \leftarrow \text{select}_1(W, j) - j + 1$, as this gives the total length of all encodings *before* B_j . We show that this encoding matches the bounds from Thm. 10.

The size of V is again $nH_k(A) + O(\frac{nk \lg \sigma}{\lg n})$ bits (see the proof of Thm. 10), and the leading second-order-term for answering the out-of-block-queries remains $O(n \lg \lg n / \lg n)$.

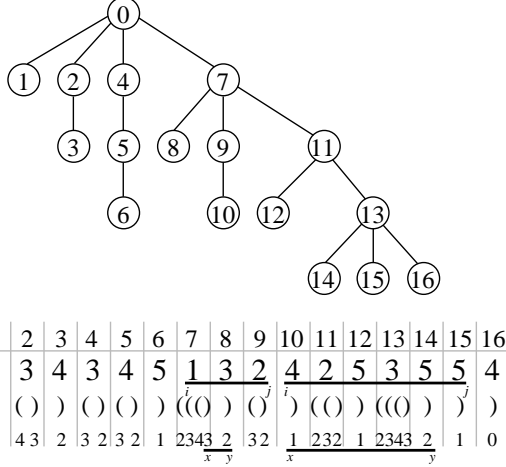


Figure 6: Top: The 2d-Min-Heap \mathcal{M}_A of the input array A . Bottom: \mathcal{M}_A 's DFUDS U and U 's excess sequence E . Two example queries $\text{RMQ}_A(i, j)$ are underlined, including their corresponding queries $\pm 1\text{RMQ}_E(x, y)$.

To bound the size of W , note that as there are exactly $\lceil n/s \rceil$ 1's in W and exactly as many 0's as the length of V , the size of W is $|W| = |V| + n/\lg n$. To represent W and the select-queries on it, we use the fully indexable dictionary (see Sect. 2.3). This takes $O(\frac{n \lg \lg n}{\lg n})$ bits.

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 *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 2d-Min-Heaps

The basis will be a new tree, the *2d-Min-Heap*, defined as follows. 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.

Definition 11. *The 2d-Min-Heap \mathcal{M}_A of A is a labeled and ordered tree with vertices v_0, \dots, v_n , where v_i is labeled with i for all $0 \leq i \leq n$. For $1 \leq i \leq n$, the parent node of v_i is v_j iff $j < i$, $A[j] < A[i]$, and $A[k] \geq A[i]$ for all $j < k \leq i$. The order of the children is chosen such that their labels are increasing from left to right.*

Observe that this is a well-defined tree with the root being always labeled as 0, and that a node v_i can be uniquely identified by its label i , which we will do henceforth. See Fig. 6 for an example.

We note the following useful properties of \mathcal{M}_A .

Lemma 12. *Let \mathcal{M}_A be the 2d-Min-Heap of A .*

1. *The node labels correspond to the preorder-numbers of \mathcal{M}_A (counting starts at 0).*
2. *Let i be a node in \mathcal{M}_A with children x_1, \dots, x_k . Then $A[i] < A[x_j]$ for all $1 \leq j \leq k$.*
3. *Again, let i be a node in \mathcal{M}_A with children x_1, \dots, x_k . Then $A[x_j] \leq A[x_{j-1}]$ for all $1 < j \leq k$.*

Proof. Because the root of \mathcal{M}_A is always labeled with 0 and the order of the children is induced by their labels, property 1 holds. Property 2 follows immediately from Def. 11. For property 3, assume for the sake of contradiction that $A[x_j] > A[x_{j-1}]$ for two children x_j and x_{j-1} of i . From property 1, we know that $i < x_{j-1} < x_j$, contradicting the definition of the parent-child-relationship in \mathcal{M}_A , which says that $A[k] \geq A[x_j]$ for all $i < k \leq x_j$. ■

Properties 2 and 3 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 gives the desired connection of 2d-Min-Heaps and RMQs.

Lemma 13. *Let \mathcal{M}_A be the 2d-Min-Heap of A . For arbitrary nodes i and j , $1 \leq i < j \leq n$, 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. For an arbitrary node x in \mathcal{M}_A , let T_x denote the subtree of \mathcal{M}_A that is rooted at x . There are two cases to prove.

$\ell = i$. This means that j is a descendant of i . Due to property 1 of Lemma 12, this implies that all nodes $i, i+1, \dots, j$ are in T_i , and the recursive application of property 2 implies that $A[i]$ is the minimum in the query range $[i, j]$.

$\ell \neq i$. Let x_1, \dots, x_k be the children of ℓ . Further, let α and β ($1 \leq \alpha \leq \beta \leq k$) be defined such that T_{x_α} contains i , and T_{x_β} contains j . Because $\ell \neq i$ and property 1 of Lemma 12, we must have $\ell < i$; in other words, the LCA is not in the query range. But also due to property 1, every node in $[i, j]$ is in T_{x_γ} for some $\alpha \leq \gamma \leq \beta$, and in particular $x_\gamma \in [i, j]$ for all $\alpha < \gamma \leq \beta$. Taking this together with property 2, we see that $\{x_\gamma : \alpha < \gamma \leq \beta\}$ are the only candidate positions for the minimum in $A[i, j]$. Due to property 3, 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 and $o(n)$ structures for $\text{rank}_()$ -, $\text{select}_()$ -, and $\text{findopen}_()$ -operations on U (see Sect. 2.3). We further need structures for $\pm 1\text{RMQ}$ on the *excess-sequence* $E[1, 2n]$ of U , defined as $E[i] = \text{rank}_l(U, i) - \text{rank}_r(U, i)$. 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 Sect. 2.4.

The reasons for preferring the DFUDS over the BPS-representation [39] 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 14. *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 ($1 \leq i < j \leq n$).*

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

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 [34, 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 13) — it follows from the description of the parent-operation in the original article on DFUDS [6] 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 13). ■

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. 6, where the respective RMQs in both A and E are underlined and labeled with the variables from Cor. 14.

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 (Sect. 5.2.1), and then show how to reduce its space to $n + o(n)$ bits, while still having linear running time (Sect. 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

⁶In line 1, we correct a minor error in the original article [34] 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).

properties that $A[S[h]] \geq \dots \geq A[S[1]]$, and $i < S[h] < \dots < S[1] \leq n$. 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 closest smaller value (properties 2 and 3 of Lemma 12). 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* $B_0, \dots, B_{\lfloor \frac{n-1}{s} \rfloor}$. Further, $s' = s^2$ elements are grouped into *super-blocks* $B'_0, \dots, B'_{\lfloor \frac{n-1}{s'} \rfloor}$.

For each such (super-)block B that contains at least one 1, in a new table M (or M' , respectively) at position x we store the block number of the leftmost (super-)block to the right of B that contains a 1, in M only relative to the beginning of the super-block. These tables need $O(\frac{n}{s} \cdot \lg \frac{s'}{s}) = O(\frac{n \lg \lg n}{\lg n})$ and $O(\frac{n}{s'} \cdot \lg \frac{n}{s}) = O(\frac{n}{\lg n})$ bits of space, respectively. 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 $O(2^s \cdot \lg s) = O(\sqrt{n} \lg \lg n)$ 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 $i+1$. For finding the leftmost 1 in S' to the right of $j > i$ (position j has just been popped from S), we first check if j 's block B_x , $x = \lfloor \frac{j-1}{s} \rfloor$, contains a 1, and if so, find this leftmost 1 by consulting P . If B_x does not contain a 1, we jump to the next block B_y containing a 1 by first jumping to $y = x + M[x]$, and if this block does not contain a 1, by further jumping to $y = M'[\lfloor \frac{j-1}{s'} \rfloor]$. In block y , we can again use P to find the leftmost 1. Thus, we can find the new top of S in constant time.

In order to keep 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). Because in step i only i is pushed on S , for operation (1) we just need to store the block number y of the former top in $M[x]$ ($x = \lfloor \frac{i-1}{s} \rfloor$), if this is in a different block (i.e., if $x \neq y$). Changes to M' are similar. For operation (2), nothing has to be done at all, because even if the popped index was the last 1 in its (super-)block, we know that all (super-)blocks to the left of it do not contain a 1, so no values in M and M' have to be changed. Note that this only works because elements to the right of i 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 $O(\frac{n \lg^2 \lg n}{\lg n})$ bits from Sadakane's preprocessing scheme for ± 1 RMQ (Sect. 2.4), while all other terms (for *rank*, *select* and *findopen*) are $O(\frac{n \lg \lg n}{\lg n})$. We show in this section a simple way to lower the space for ± 1 RMQ to $O(\frac{n \lg \lg n}{\lg n})$, thereby completing the proof of Thm. 17. The techniques are similar to the ones presented in Sect. 3.4.

As in the original algorithm [47], we divide the input array E into $n' = \lfloor \frac{n-1}{s} \rfloor$ blocks of size $s = \lceil \frac{\lg n}{2} \rceil$. 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 $O(\sqrt{n} \lg^2 n)$ 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 $O(\frac{n \lg \lg n}{\lg n})$ 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 Sect. 3, using $2n' + o(n') = O(\frac{n}{\lg n})$ bits of space. Thus, by querying $\text{RMQ}_{E'}(i, j)$ for $1 \leq i \leq j \leq n'$, 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 Sect. 3 does allow the input array to be represented implicitly, as in our case. And second, it does not use Sadakane's solution for ± 1 RMQ, so there are no circular dependencies. Hence, we get:

Theorem 15. *Let $E[1, n]$ be an array of numbers with the property $E[i] - E[i-1] = \pm 1$ for all $1 < i \leq n$, encoded as an n -bit-vector $S[1, n]$ by $S[1] = 0$, and $S[i] = 1$ iff $E[i] - E[i-1] = +1$ for all $1 < i \leq n$. Then there is a preprocessing scheme for RMQs on E with time complexity $\langle O(n), O(1) \rangle$ and bit-space complexity $\left\lceil O(\frac{n \lg \lg n}{\lg n}) + |S|, O(\frac{n \lg \lg n}{\lg n}) + |S| \right\rceil$.* ■

As a corollary, this approach also lowers the space for LCA-computation in BPS [47] and DFUDS [34] from $O(\frac{n \lg^2 \lg n}{\lg n})$ to $O(\frac{n \lg \lg n}{\lg n})$, as these are both based on ± 1 RMQ:

Corollary 16. *Given the BPS or DFUDS of an ordered tree T , there is a data structure of size $O(\frac{n \lg \lg n}{\lg n})$ bits that allows to answer LCA-queries in T in constant time.* ■

5.4 The Final Result

We summarize this section in a final theorem.

Theorem 17. *For an array A of n objects from a totally ordered universe, there is a preprocessing scheme for RMQ with time complexity $\langle O(n), O(1) \rangle$ and bit-space complexity*

$$\left\lceil 3n + O\left(\frac{n \lg \lg n}{\lg n}\right) + |A|, 2n + O\left(\frac{n \lg \lg n}{\lg n}\right) \right\rceil.$$

■

References

- [1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.
- [2] S. Alstrup, C. Gavaille, 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.
- [3] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. In *Proc. SODA*, pages 344–357. ACM/SIAM, 1990.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.
- [8] I. Bialynicka-Birula and R. Grossi. Amortized rigidity in dynamic Cartesian Trees. In *Proc. STACS*, volume 3884 of *LNCS*, pages 80–91. Springer, 2006.
- [9] G. Chen, S. J. Puglisi, and W. F. Smyth. LZ factorization using less time and space. *Mathematics in Computer Science*, 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] 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.
- [12] N. Dershowitz and S. Zaks. Enumerations of ordered trees. *Discrete Math.*, 31(1):9–28, 1980.
- [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 Transactions on 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. Wee LCP. CoRR, abs/0910.3123, 2009.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.
- [23] 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.
- [24] A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007.
- [25] 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.
- [26] L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *Proc. SODA*, pages 869–878. ACM/SIAM, 2004.
- [27] A. Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.
- [28] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. CPM*, volume 4009 of *LNCS*, pages 294–305. Springer, 2006.
- [29] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.
- [30] 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.
- [31] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- [32] W. K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top- k string retrieval problems. In *Proc. FOCS*, pages xx–xx. IEEE Computer Society, to appear 2009.
- [33] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

- [34] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.
- [35] D. E. Knuth. *The Art of Computer Programming Volume 4 Fascicle 4: Generating All Trees; History of Combinatorial Generation*. Addison-Wesley, 2006.
- [36] 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.
- [37] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- [38] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
- [39] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
- [40] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666. ACM/SIAM, 2002.
- [41] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.
- [42] 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.
- [43] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.
- [44] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time. In *Proc. AWOC*, volume 319 of *LNCS*, pages 33–42. Springer, 1988.
- [45] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. *ACM Transactions on Algorithms*, 3(4):Article No. 43, 2007.
- [46] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
- [47] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [48] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
- [49] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.
- [50] K. Sadakane and G. Navarro. Fully-functional succinct trees. Accepted for SODA’10. See also CoRR, abs/0905.0768v1, 2009.

- [51] S. Saxena. Dominance made simple. *Inform. Process. Lett.*, 109(9):409–421, 2009.
- [52] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
- [53] J. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, 1980.
- [54] T. Shibuya and I. Kurochkin. Match chaining algorithms for cDNA mapping. In *Proc. Workshop on Algorithms in Bioinformatics (WABI)*, volume 2812 of *LNCS*, pages 462–475. Springer, 2003.
- [55] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, volume 4580 of *LNCS*, pages 205–215. Springer, 2007.
- [56] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.