Optimal Succinctness for Range Minimum Queries

Johannes Fischer

Universität Tübingen, Center for Bioinformatics (ZBIT), Sand 14, D-72076 Tübingen
fischer@informatik.uni-tuebingen.de

Abstract. For a static array A of n totally ordered objects, a *range minimum query* asks for the position of the minimum between two specified array indices. We show how to preprocess A into a scheme of size $2n+o(n)$ bits that allows to answer range minimum queries on A in constant time. This space is asymptotically optimal in the important setting where access to A is not permitted after the preprocessing step. Our scheme can be computed in linear time, using only $n+o(n)$ additional bits for construction. We also improve on LCA-computation in BPS- or DFUDS-encoded trees.

1 Introduction

For an array $A[1,n]$ of n natural numbers or other objects from a totally ordered universe, a *range minimum query* $\text{RMQ}_A(i,j)$ for $i \leq j$ returns the *position* of where the minimum element in the sub-array $A[i,j]$ occurs; i.e., $\text{RMQ}_A(i,j) = \text{argmin}_{i \leq k \leq j}\{A[k]\}$. This fundamental algorithmic problem has numerous applications, e.g., in text indexing [1,15,35], text compression [7], document retrieval [30, 36, 41], flowgraphs [19], range queries [39], position-restricted pattern matching [8], just to mention a few.

In all of these applications, the array A in which the range minimum queries (RMQs) are performed is static and known in advance, which is also the scenario considered in this article. In this 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.

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 answer RMQs on A in constant time.

The historically first such scheme due to Gabow et al. [16] is based on the following idea: because an RMQ-instance can be transformed into an instance of *lowest common ancestors* (LCAs) in the *Cartesian Tree* [42], one can use any linear-time preprocessing scheme for $O(1)$-LCAs [3, 5, 22, 40] in order to answer RMQs in constant time.

The problem of this transformation [16], both in theory and in practice, can be seen by the following dilemma: storing the Cartesian Tree *explicitly* (i.e., with labels and pointers) needs $O(n \log n)$ bits of space, while storing it *succinctly* in $2n + o(n)$ bits [4, 29] does not allow to map the array-indices to the corresponding nodes (see Sect. 1.1 for more details on why this is difficult).

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 $(1 + o(1)) \log L$ bits.[1] Research on succinct data structures is very active, and we just mention some examples from the realm of trees [4, 9, 18, 25, 29, 38], dictionaries [32, 33], and strings [10, 11, 20, 21, 34, 37], being well aware of the fact that this list is far from complete. This article presents the first succinct data structure for

[1] Throughout this article, space is measured in bits, and log denotes the binary logarithm.

Table 1. Preprocessing schemes for $O(1)$-RMQs, where $|A|$ denotes the space for the (read-only) input array.

reference	final space	construction space	comments
[5, 22, 40]	$O(n \log n) + \|A\|$	$O(n \log n) + \|A\|$	originally devised for LCA, but solve RMQ via Cartesian Tree
[3]	$O(n \log n) + \|A\|$	$O(n \log n) + \|A\|$	significantly simpler than previous schemes
[2]	$O(n \log n) + \|A\|$	$O(n \log n) + \|A\|$	only solution not based on Cartesian Trees
[13]	$2n + o(n) + \|A\|$	$2n + o(n) + \|A\|$	generalizes to $\frac{2}{c}n + o(n) + \|A\|$ bits, const. c (see Footnote 2)
[14]	$O(nH_k) + o(n)$	$2n + o(n) + \|A\|$	H_k is the empirical entropy [27] of A (small if A is compressible)
[35]	$n + o(n)$	$n + o(n)$	only for ± 1RMQ; A *must* be encoded as an n-bit-vector
[36]	$4n + o(n)$	$O(n \log n) + \|A\|$	only non-systematic data structure so far
this article	$\boldsymbol{2n + o(n)}$	$\boldsymbol{3n + o(n) + \|A\|}$	**final space requirement optimal**

$O(1)$-RMQs in the standard word-RAM model of computation (which is also the model used in all LCA- and RMQ-schemes cited in this article).

Before detailing our contribution, we first classify and summarize existing solutions for $O(1)$-RMQs.

1.1 Previous Solutions for RMQ

In accordance with common nomenclature [17], 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. In such a case the query algorithm can consult 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 important for at least two reasons:

1. In some applications, e.g., in algorithms for document retrieval [30, 36] or position restricted substring matching [8], only the *position* of the minimum matters, but *not* the value of this minimum. In such cases it would be a waste of space (both in theory and in practice) 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* [26]. However, storing the LCP-array efficiently in $2n + o(n)$ bits [35] increases the access-time to the time needed to retrieve an entry from the corresponding *suffix array* [26], which is $\Omega(\log^{\epsilon} n)$ (constant $\epsilon > 0$) at the very best if the suffix array is also stored in compressed form [20, 34]. 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 [15] (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 third column we list the peak space consumption at construction time of each scheme, which sometimes differs from the former term.

Systematic Schemes. Most schemes are based on the Cartesian Tree [42], the only exception being the scheme due to Alstrup et al. [2]. All direct schemes [2, 3, 13, 35] 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, with Bender et al.'s scheme [3] being less complex than the previous ones, and Alstrup et al.'s [2] being even simpler (and most practical). The only $O(n)$-bit scheme is due to Fischer and Heun [13] and achieves $2n + o(n)$ bits of space in addition to the space for the input array A. It is based on an "implicit" enumeration of Cartesian Trees only for very small blocks (instead of the whole array A). Its further advantage is that it can be adapted to achieve entropy-bounds for compressible inputs [14]. For systematic schemes, no lower bound on space is known.[2]

An important special case is Sadakane's $n+o(n)$-bit solution [35] for ± 1RMQ, where it is assumed that A has the property that $A[i] - A[i-1] = \pm 1$ for all $1 < i \leq n$, and can hence be encoded as a bit-vector $S[1, n]$, where a '1' at position i in S indicates that A increases by 1 at position i, and a '0' that it decreases. Because we will make use of this scheme in our new algorithm, and also improve on its space consumption in Sect. 5, we will describe it in greater detail in Sect. 2.2.

Non-Systematic Schemes. The only existing scheme is due to Sadakane [36] and uses $4n + o(n)$ bits. It is based on the balanced-parentheses-encoding (BPS) [29] of the Cartesian Tree T of the input array A and a $o(n)$-LCA-computation therein [35]. 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 T); 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 $2n - \Theta(\log n)$ 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); 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 $\log(\binom{2n-1}{n-1}/(2n-1)) = 2n - \Theta(\log n)$ bits [29].
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 $O(n \log n)$ bits, as each node occupies $O(\log n)$ 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 [6].

1.2 Our Results

We address the two aforementioned problems of Sadakane's solution [36] and resolve them in the following way:

[2] The claimed lower bound of $2n + o(n) + |A|$ bits under the "min-probe-model" [13] turned out to be wrong, as was kindly pointed out to the authors by S. Srinivasa Rao (personal communication, November 2007). In fact, it is easy to lower the space consumption of [13] to $\frac{2}{c}n + o(n) + |A|$ bits (constant integer $c > 0$) by grouping c adjacent elements in A's blocks together, and "building" the Cartesian Trees only on the minima of these groups.

1. We introduce a new preprocessing scheme for $O(1)$-RMQs that occupies only $2n + o(n)$ bits in memory, thus being the first that asymptotically achieves the information-theoretic lower bound for non-systematic schemes. 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 [29] give a $2n + o(n)$-bit-solution for representing ordered trees, while supporting most navigational operations in constant time, although a $O(n)$-bit-solution (roughly $10n$ bits [29]) had already been known for some 10 years before [24]. Another example comes from compressed text indexing [31], where a lot of effort has been put into achieving indexes of size $nH_k + o(n \log \sigma)$ [11], although indexes of size $O(nH_k) + o(n \log \sigma)$ had been known earlier [10, 21, 34]. (Here, H_k is the k-th-order empirical entropy of the input text T [27] and measures the "compressibility" of T, while σ is T's alphabet size.)
2. We give a *direct* construction algorithm for the above scheme that needs only $n + o(n)$ bits of space in addition to the space for the final scheme, thus lowering the construction space for non-systematic schemes from $O(n \log n)$ to $O(n)$ bits (on top of A). This is a significant improvement, as the space for storing A is not necessarily $\Theta(n \log n)$; for example, if the numbers in A are integers in the range $[1, \log^{O(1)} n]$, A can be stored as an array of packed words using only $O(n \log \log n)$ bits of space. See Sect. 6 for a different example. The construction space 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 intuitive explanation why our scheme works better than Sadakane's scheme [36] is that ours is based on a new tree in which the preorder-numbers of the nodes correspond to the array-indices in A, thereby rendering the introduction of "fake" leaves (as described earlier) unnecessary. In summary, this article is devoted to proving

Theorem 1. *For an array A of n objects from a totally ordered universe, there is a preprocessing scheme for $O(1)$-RMQs on A that occupies only $2n+O(\frac{n \log \log n}{\log n})$ bits of memory, while not needing access to A after its construction, thus meeting the information-theoretic lower bound. This scheme can be constructed in $O(n)$ time, using only $n + o(n)$ bits of space in addition to the space for the input and the final scheme.*

This result is not only appealing in theory, but also important in practice. For example, when RMQs are used in conjunction with sequences of DNA (genomic data), where the alphabet size σ is 4, storing the DNA even in *uncompressed* form takes only $2n$ bits, already less than then $4n$ bits of Sadakane's solution [36]. Hence, halving the space for RMQs leads to a significant reduction of total space. Further, because n is typically very large ($n \approx 2^{32}$ for the human genome), a construction space of $O(n \log n)$ bits is much higher than the $O(n \log \sigma)$ bits for the DNA itself. An additional (practical) advantage of our new scheme is that it also halves the space of the lower order terms ("$o(2n)$ vs. $o(4n)$ bits"). This is particularly relevant for realistic problem sizes, where the lower order terms dominate the linear term. An implementation in C++ of our new scheme can be downloaded from http://www-ab.informatik.uni-tuebingen.de/people/fischer/optimalRMQ.tgz.

1.3 Outline

Sect. 2 presents some basic tools. Sect. 3 introduces the new preprocessing scheme. Sect. 4 addresses the linear-time construction of the scheme. Sect. 5 lowers the second-order term by giving a new data structure for LCA-computation in succinct trees. Sect. 6 shows a concrete example of an application where our new preprocessing scheme improves on the total space.

2 Preliminaries

This section sketches some *known* data structures that we are going to make use of. Throughout this article, we use the standard *word-RAM* model of computation, where fundamental arithmetic operations on words consisting of $\Theta(\log n)$ consecutive bits can be computed in $O(1)$ time.

2.1 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: $rank_1(S,i)$ gives the number of 1's in the prefix $S[1,i]$, and $select_1(S,i)$ 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 similarly for 0-bits. There are data structures of size $O(\frac{n \log\log n}{\log n})$ bits in addition to S that support rank- and select-operations in $O(1)$ time [28].

2.2 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 $E[1] + rank_1(S,i) - rank_0(S,i) + 1 = E[1] + i - 2rank_0(S,i) + 1$. Under this setting, Sadakane [35] shows how to support RMQs on E in $O(1)$ time, using S and additional structures of size $O(\frac{n \log^2 \log n}{\log n})$ bits. We will improve this space to $O(\frac{n \log\log n}{\log n})$ in Sect. 5. A technical detail is that $\pm 1\textsc{rmq}(i,j)$ yields the position of the *leftmost* minimum in $E[i,j]$ if there are multiple occurrences of this minimum.

2.3 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 $rank_((B,i) - rank_)(B,i) = rank_((B,j) - rank_)(B,j)$. The *findopen*-operation can be computed in constant time [29]; the most space-efficient data structure for this needs $O(\frac{n \log\log n}{\log n})$ bits [18].

2.4 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 [4]. 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).

3 The New Preprocessing Scheme

We are now ready to dive into the technical details of our new preprocessing scheme. 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.

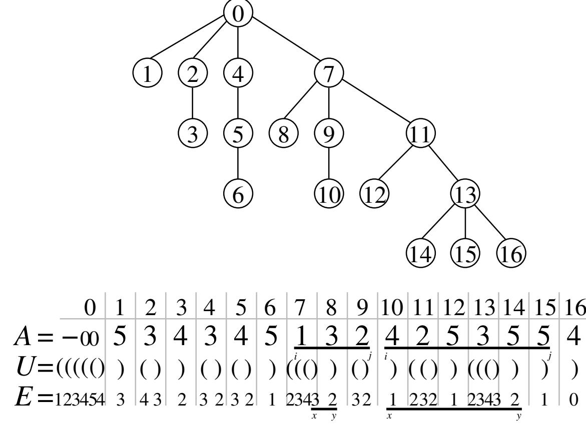

Fig. 1. Top: The 2d-Min-Heap $\mathcal{M}_A$ of the input array A. Bottom: $\mathcal{M}_A$'s DFUDS U and U's excess sequence E. Two example queries $\text{RMQ}_A(i,j)$ are underlined, including their corresponding queries $\pm 1\text{RMQ}_E(x,y)$.

Definition 1. *The* 2d-Min-Heap $\mathcal{M}_A$ *of* A *is a labeled and ordered 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* v_j *iff* $j < i$*,* $A[j] < A[i]$*, and* $A[k] \ge A[i]$ *for all* $j < k \le i$*. 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. 1 for an example.

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

Lemma 1. *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* $x_1, \ldots, x_k$*. Then* $A[i] < A[x_j]$ *for all* $1 \le j \le k$*.*
3. *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. 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. 1. 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 $A[k] \ge A[x_j]$ for all $i < k \le x_j$. $\square$

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 2. *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. 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 1, this implies that all nodes $i, i+1, \ldots, j$ 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 $x_1, \ldots, x_k$ 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 1, 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. □

Note that (unlike for ±1RMQ) this algorithm yields the *rightmost* minimum in the query range if this is not unique. However, it can be easily arranged to return the leftmost minimum by adapting the definition of the 2d-Min-Heap, if this is desired.

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 $rank_)$-, $select_)$-, and *findopen*-operations on U (see Sect. 2). We further need structures for ±1RMQ on the *excess-sequence* $E[1, 2n]$ of U, defined as $E[i] = rank_((U, i) - rank_)(U, i)$. 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 ±1RMQ-scheme from Sect. 2.2.

The reasons for preferring the DFUDS over the BPS-representation [29] 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 1. *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. $x \leftarrow select_)(U, i+1)$
2. $y \leftarrow select_)(U, j)$
3. $w \leftarrow \pm 1\text{RMQ}_E(x, y)$
4. if $rank_)(U, findopen(U, w)) = i$ then return i
5. else return $rank_)(U, w)$

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 [25, Lemma 3.2], we see that after the ±1RMQ-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.[3] Line 4 checks if $\ell = i$ by comparing their preorder-numbers and returns i in that case (case 1 of Lemma 2) — it follows from the description of the parent-operation in the original article on DFUDS [4] 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 2). □

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. 1, where the respective RMQs in both A and E are underlined and labeled with the variables from Cor. 1.

[3] In line 1, we correct a minor error in the original article [25] 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).

4 Construction of 2d-Min-Heaps

We now 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 $O(n \log n)$ bits (Sect. 4.1), and then show how to reduce its space to $n + o(n)$ bits, while still having linear running time (Sect. 4.2).

4.1 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 $A[S[h]] \geq \cdots \geq A[S[1]]$, 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 which have $A[i]$ as their closest smaller value (properties 2 and 3 of Lemma 1). 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.

4.2 $O(n)$-bit Solution

The only drawback of the above algorithm is that stack S requires $O(n \log n)$ 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 $s = \lceil \frac{\log n}{2} \rceil$ 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 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 $O(\frac{n}{s} \log(s'/s)) = O(\frac{n \log\log n}{\log n})$ and $O(\frac{n}{s'} \log(n/s)) = O(\frac{n}{\log n})$ 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 $O(2^s \cdot \log s) = O(\sqrt{n} \log\log n) = o(n)$ 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 known 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 Lowering the Second-Order-Term

Until now, the second-order-term is dominated by the $O(\frac{n \log^2 \log n}{\log n})$ bits from Sadakane's preprocessing scheme for ± 1RMQ (Sect. 2.2), while all other terms (for *rank*, *select* and *findopen*) are $O(\frac{n \log \log n}{\log n})$. We show in this section a simple way to lower the space for ± 1RMQ to $O(\frac{n \log \log n}{\log n})$, thereby completing the proof of Thm. 1.

As in the original algorithm [35], we divide the input array E into $n' = \lfloor \frac{n-1}{s} \rfloor$ blocks of size $s = \lceil \frac{\log n}{2} \rceil$. 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 $O(\sqrt{n} \log^2 n)$ 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 $E'[i] = is + E[E''[i]]$. Because E'' stores $n / \log n$ numbers from the range $[1, s]$, the size for storing E' is thus $O(\frac{n \log \log n}{\log n})$ bits. Note that unlike E, E' does not necessarily fulfill the ± 1-property. E' is now preprocessed for constant-time RMQs with the systematic scheme of Fischer and Heun [13], using $2n' + o(n') = O(\frac{n}{\log n})$ 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 used RMQ-scheme [13] 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.

As a corollary, this approach also lowers the space for LCA-computation in BPS [35] and DFUDS [25] from $O(\frac{n \log^2 \log n}{\log n})$ to $O(\frac{n \log \log n}{\log n})$, as these are based on ± 1RMQ:

Corollary 2. *Given the BPS or DFUDS of an ordered tree T, there is a data structure of size $O(\frac{n \log \log n}{\log n})$ bits that allows to answer LCA-queries in T in constant time.*

6 Application in Document Retrieval Systems

We now sketch a concrete example of where Thm. 1 lowers the construction space of a different data structure. This section is meant to show that there are indeed applications where the memory bottleneck is the construction space for RMQs. We consider the following problem:

Problem 2 (Document Listing Problem [30]).

Given: a collection of k text documents $\mathcal{D} = \{D_1, \ldots, D_k\}$ of total length n.

Compute: an index that, given a search pattern P of length m, returns all d documents from $\mathcal{D}$ that contain P, in time proportional to m and d (in contrast to *all* occurrences of P in $\mathcal{D}$).

Sadakane [36, Sect. 4] gives a succinct index for this problem. It uses three parts, for convenience listed here together with their *final* size:

- Compressed suffix array [34] A of the concatenation of all k documents, $|A| = \frac{1}{\epsilon}H_0 n + O(n)$ bits.
- Array of document identifiers D, defined by $D[i] = j$ iff the $A[i]$'th suffix "belongs to" document j. Its size is $O(k \log \frac{n}{k})$ bits
- Range minimum queries on an array C, $|RMQ| = 4n + o(n)$ bits. Here, C stores positions in A of nearest previous occurrences of indexed positions from the *same* document, $C[i] = \max\{j < i : D[j] = D[i]\}$. In the query algorithm, only the *positions* of the minima matter; hence, this is a *non-systematic* setting.

Apart from halving the space for RMQ from $4n$ to $2n$ bits, our new scheme also lowers the peak space consumption of Sadakane's index for the Document Listing Problem. Let us consider the *construction* time and space for each part in turn:

- Array A can be built in $O(n)$ time and $O(n)$ bits (constant alphabet), or $O(n \log\log |\Sigma|)$ time using $O(n \log |\Sigma|)$ bits (arbitrary alphabet Σ) of space [23].
- Array D is actually implemented as a fully indexable dictionary [33] called D', and can certainly be built in linear time using $O(n)$ bits working space, as we can always couple the block-encodings [33] with the $o(n)$-bit structures for uncompressed solutions for rank and select [28].
- As already mentioned before, for a fast construction of Sadakane's scheme for $O(1)$-RMQs on C, we would have needed $\Theta(n \log n)$ bits. Our new method lowers this to $O(n)$ bits construction space. Note that array C needs never be stored *plainly* during the construction: because C is scanned only once when building the DFUDS (Sect. 4) and is thus accessed only sequentially, we only need to store the positions in A of the last seen document identifier for each of the k documents. This can be done using a plain array, so $|C| = O(k \log n)$ bits.

In summary, we get:

Theorem 2. *The construction space for Sadakane's Index for Document Listing [36] is lowered from $O(n \log n)$ bits to $O(n+k \log n)$ bits (constant alphabet) or $O(n \log |\Sigma| + k \log n)$ bits (arbitrary alphabet Σ) with our scheme for RMQs from Thm. 1, while not increasing the construction time.*

This is especially interesting if k, the number of documents, is not too large, $k = O(\frac{n \log |\Sigma|}{\log n})$.

7 Concluding Remarks

We have given the first optimal preprocessing scheme for $O(1)$-RMQs under the important assumption that the input array is not available after preprocessing. To the expert, it might come as a surprise that our algorithm is *not* based on the Cartesian Tree, a concept that has proved to be very successful in former schemes. Instead, we have introduced a new tree, the 2d-Min-Heap, which seems to be better suited for our task.[4] We hope to have thereby introduced a new versatile data structure to the algorithms community. And indeed, we are already aware of the fact that the

[4] The Cartesian Tree and the 2d-Min-Heap are certainly related, as they are both obtained from the array, and it would certainly be possible to derive the 2d-Min-Heap (or a related structure obtained from the natural bijection between binary and ordered rooted trees) from the Cartesian Tree, and then convert it to the BPS/DFUDS. But see the second point in Sect. 1.2 why this is *not a good idea.*

2d-Min-Heap, made public via a preprint of this article [12], is pivotal to a new data structure for succinct trees [38].

We leave it as an open research problem whether the $3n+o(n)$-bit construction space be lowered to an optimal $2n+o(n)$-bit "in-place" construction algorithm. (A simple example shows that it is *not* possible to use the leading n bits of the DFUDS for the stack.)

Acknowledgments

The author wishes to thank Volker Heun, Veli Mäkinen, and Gonzalo Navarro for their helpful comments on a draft of this article.

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. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. SPAA*, pages 258–264. ACM Press, 2002.
3. 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.
4. 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.
5. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.
6. I. Bialynicka-Birula and R. Grossi. Amortized rigidness in dynamic Cartesian Trees. In *Proc. STACS*, volume 3884 of *LNCS*, pages 80–91. Springer, 2006.
7. 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.
8. 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.
9. 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.
10. P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
11. 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.
12. J. Fischer. Optimal succinctness for range minimum queries. CoRR, abs/0812.2775, 2008.
13. 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.
14. 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.
15. J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. CPM*, volume 5029 of *LNCS*, pages 152–165. Springer, 2008.
16. 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.
17. A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007.
18. 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.
19. L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *Proc. SODA*, pages 869–878. ACM/SIAM, 2004.
20. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.
21. 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.
22. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
23. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009.

24. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.
25. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.
26. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
27. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
28. J. I. Munro. Tables. In *Proc. FSTTCS*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.
29. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
30. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666. ACM/SIAM, 2002.
31. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.
32. R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.
33. 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.
34. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
35. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
36. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
37. K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.
38. K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. Accepted for SODA'10. See CoRR, abs/0905.0768v1, 2010.
39. S. Saxena. Dominance made simple. *Inform. Process. Lett.*, 109(9):409–421, 2009.
40. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
41. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, volume 4580 of *LNCS*, pages 205–215. Springer, 2007.
42. J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.