Practical Implementation of Encoding Range Top-2 Queries

Seungbum Jo
Chungbuk National University, Cheongju, South Korea

Wooyoung Park
Seoul National University, South Korea

Srinivasa Rao Satti
Norwegian University of Science and Technology, Trondheim, Norway

Abstract

We design a practical variant of an encoding for *range Top-2 queries (RT2Q)*, and evaluate its performance. Given an array $A[1, n]$ of n elements from a total order, the range Top-2 encoding problem is to construct a data structure that can answer RT2Q queries, which return the positions of the first and the second largest elements within a given query range of A, without accessing the array A at query time. Davoodi et al. [Phil. Trans. Royal Soc. A, 2016] proposed a $(3.272n + o(n))$-bit encoding, which answers RT2Q queries in $O(1)$ time, while Gawrychowski and Nicholson [ICALP, 2015] gave an optimal $(2.755n + (n))$-bit encoding which doesn't support efficient queries. In this paper, we propose the first practical implementation of the encoding data structure for answering RT2Q. Our implementation is based on an alternative representation of Davoodi et al.'s data structure. The experimental results show that our implementation is efficient in practice, and gives improved time-space trade-offs compared to the indexing data structures (which keep the original array A as part of the data structure) for range maximum queries.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Range top-2 query, Range minimum query, Cartesian tree, Succinct encoding

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.10

Supplementary Material *Software (Source Code)*: `https://github.com/wyptcs/R2MQ`
archived at `swh:1:dir:684698b8ae0bcc6ada509f22f8ff743411de26d9`

Funding Seungbum Jo was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1G1A1101477).

1 Introduction

Given an array $A[1, n]$ of n elements from a total order, the *range maximum query* on $A[i, j]$ (denoted by RMQ(i, j)) returns the position of the largest element in $A[i, j]$. We assume that all elements in A are distinct (if there are equal elements, we can break the ties according to their positions, i.e., the leftmost one is considered as the largest value among them). The problem of constructing space and/or time-efficient data structures for answering RMQ is one of the fundamental problems in data structures, and has been extensively studied both theoretically and practically [2, 8, 9].

In general, the data structures for answering specific queries can be categorized into two types: (i) *indexing data structures*, and (ii) *encoding data structures*. In indexing data structures, one can access the input array A at query time, while it is not allowed in encoding data structures. For many problems including RMQ problem, the minimum size for an encoding data structure (referred to as *effective entropy* [17]) is much less than the input size – for example, the effective entropy for answering RMQ on A is $2n - o(n)$ bits [9], whereas storing A requires at least $n \log n$ bits[1], if all the elements in A are distinct. Thus,

[1] throughout the paper, we use log to denote the logarithm to the base 2

encoding data structures can be highly space-efficient in some cases compared to their indexing counterparts. Recent results [2, 8] show that encoding data structures for RMQ perform well both in theory and in practice.

In this paper, we consider the problem of answering *range Top-2 queries*, which extends the RMQ problem. The range Top-2 query on $A[i, j]$ (denoted by RT2Q) returns the positions of the largest and the second largest elements in $A[i, j]$. If $k = \mathsf{RMQ}(i, j)$, one can easily observe that the position of the second largest element in $A[i, j]$ is one of $k_1 = \mathsf{RMQ}(i, k-1)$ or $k_2 = \mathsf{RMQ}(k+1, j)$. Thus, any indexing data structure for answering RMQ also can answer RT2Q by comparing $A[k_1]$ and $A[k_2]$. Davoodi et al. [4] proposed the first encoding data structure for answering RT2Q in $O(1)$ time using $3.272n + o(n)$ bits, which is close to the effective entropy of $2.755n - \Theta(polylog(n))$ bits [11] for RT2Q. However, their encoding is not very practical since it represents the *Cartesian tree* [20] of A succinctly using the *tree-covering* approach of Farzan and Munro [7], which is hard to implement (compared to other succinct tree representations [1]). In this paper, we give the first practical implementation of an encoding for RT2Q. Our implementation is based on the data structure of Davoodi et al. [4], but instead of using the tree-covering approach, we use the DFUDS representation [3] of *2d-max heap* [9] which is easier to implement, and works well in practice. Our implementation supports RT2Q in $\log n \cdot g(n)$ time, for any increasing function $g(n) = \omega(1)$, using at most $3.5n + o(n)$ bits. The experimental results show that our data structure gives a better space-time trade-off, compared to the indexing data structures for RT2Q (that have access to the input array A, along with an auxiliary data structure for answering the RMQ queries).[2]

2 Preliminaries

2.1 Range Maximum Queries and Cartesian Trees

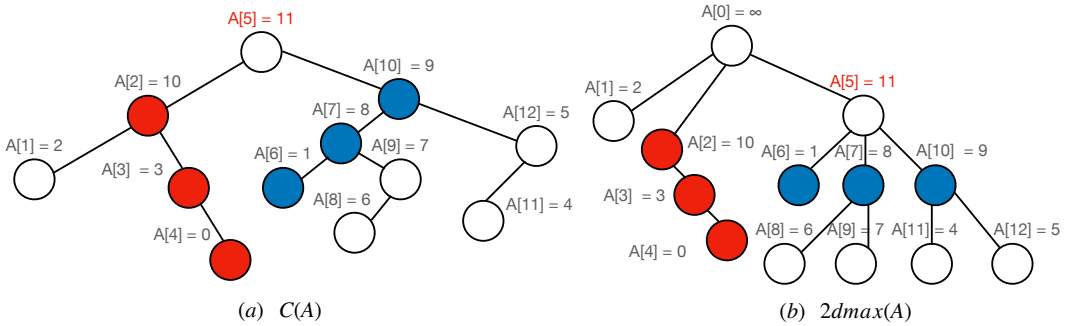
Given an array $A[1, n]$ of size n, the *Cartesian tree* [20] of A, denoted by $C(A)$, is a binary tree where (i) the root node of $C(A)$ corresponds to $A[i]$ where $i = \mathsf{RMQ}(1, n)$, and (ii) the left and right subtrees of $C(A)$ are the Cartesian trees of $A[1, i-1]$ and $A[i+1, n]$ respectively. From the definition, the i-th node in the inorder traversal of $C(A)$ corresponds to the i-th position of A (see Figure 1 (a) for an example). In the rest of this paper, we refer to the nodes in the Cartesian tree by their inorder numbers (i.e., their corresponding positions in the array A). Also, one can convert the RMQ problem on A into the LCA (lowest common ancestor) problem on $C(A)$ [10]. More precisely, for any $i, j \in [1, n]$, $\mathsf{RMQ}(i, j)$ is the same as $\mathsf{LCA}(i, j)$, which is the LCA of the node i and j in $C(A)$. This implies that one can support RMQ on A by storing $C(A)$ instead if A (thus this gives an encoding for answering RMQ on A). All the existing encoding data structures for answering RMQ use a Cartesian tree or its variants.

2.2 Davoodi et al.'s encoding data structure for RT2Q

We introduce the $(3.272n + o(n))$-bit data structure of Davoodi et al. [4], which answers RT2Q in $O(1)$ time on an array $A[1, n]$ of size n. Their data structure answers the $\mathsf{RT2Q}(i, j)$ query by performing the following three steps:

1. Compute and return the position $k = \mathsf{RMQ}(i, j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compare $A[k_1]$ and $A[k_2]$, and return k_1 if $A[k_1] > A[k_2]$, or k_2 otherwise.

[2] our implementation is available at `https://github.com/wyptcs/R2MQ`.



(a) $C(A)$ (b) $2dmax(A)$

$A[1,12] = 2\ \ 10\ \ 3\ \ 0\ \ 11\ \ 1\ \ 8\ \ 6\ \ 7\ \ 9\ \ 4\ \ 5$

$D(2dmax(A)) = (\ \ (\ \ (\ \ (\ \ )\ \ )\ \ (\ \ )\ \ (\ \ )\ \ )\ \ (\ \ (\ \ (\ \ )\ \ )\ \ (\ \ (\ \ )\ \ )\ \ )\ \ (\ \ (\ \ )\ \ )\ \ )$

$S_2 = 1\ \ S_5 = 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ \ S_7 = 1\ \ 1\ \ \ S_{10} = 0\ \ 0\ \ 1\ \ \ S_1, S_3, S_4, S_6, S_8, S_9, S_{11}, S_{12} = \epsilon$

$S = 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1$

Figure 1 Example of (a) $C(A)$ and (b) $2dmax(A)$ of the array $A[1, 12]$. Red and blue colored nodes are the nodes in $\mathsf{linspine}(5)$ and $\mathsf{rinspine}(5)$ of $C(A)$ respectively.

For answering $k = \mathsf{RMQ}(i, j)$, they maintain the *tree-covering* [7] representation of $C(A)$ to support LCA queries in $O(1)$ time, using $2n + o(n)$ bits. Next, to compare $A[k_1]$ and $A[k_2]$ without storing A, they store the *spine sequence* S of A defined as follows. For any node i which has left child i_l and right child i_r, let *left spine* (resp., *right spine*) of i, denoted by $\mathsf{lspine}(i)$ (resp., $\mathsf{rspine}(i)$), be the path from the node i to the leftmost (resp., rightmost) descendant of i. Also, let *left inner spine* (resp., *right inner spine*) of i, denoted by $\mathsf{linspine}(i)$ (resp., $\mathsf{rinspine}(i)$), be the $\mathsf{rspine}(i_l)$ (resp., $\mathsf{lspine}(i_r)$), and define L_i, R_i, l_i, and r_i to be a number of nodes in $\mathsf{lspine}(i)$, $\mathsf{rspine}(i)$, $\mathsf{linspine}(i)$, and $\mathsf{rinspine}(i)$ respectively. Then by the property of $C(A)$, the nodes k_1 and k_2 in $C(A)$ are always on $\mathsf{linspine}(k)$ and $\mathsf{rinspine}(k)$, respectively. Now we define the array $S_k[1, m_k]$ to be a bit array of size $m_k = \max(l_k + r_k - 1, 0)$ where $S_k[i] = 0$ if the i-th largest element of A among the positions corresponding to $\mathsf{linspine}(k) \cup \mathsf{rinspine}(k)$ is in $\mathsf{linspine}(k)$, and 1 otherwise. Let $\mathsf{depth}(k)$ be the depth of the node k, and for any given pattern b and sequence S, let $\mathsf{rank}_b(S, i)$ be the number of occurrences of b in the first i positions of S, and $\mathsf{select}_b(S, i)$ be the position of i-th occurrence of b in S. Then one can compare $A[k_1]$ and $A[k_2]$ by comparing $\mathsf{select}_0(S_k, \mathsf{depth}(k_1) - (\mathsf{depth}(k) + 1) + 1)$ and $\mathsf{select}_1(S_k, \mathsf{depth}(k_2) - (\mathsf{depth}(k) + 1) + 1)$ (i.e., by checking which of the two bits corresponding to the nodes k_1 and k_2 come first in S_k). The sequence S is simply defined by concatenating all S_k's for all nodes $k \in C(A)$ in the increasing order of their inorder numbers. Finally, to locate the starting position of S_k in S efficiently, they introduce the following lemma.

▶ **Lemma 1** ([4]). *For any $u \in C(A)$,*

$$\sum_{j<u} m_j = 2u - L_\tau - l_u + \mathit{Ldepth}(u) - \mathit{Rdepth}(u) + 1 - (u - \mathit{Lleaves}(u))$$

In the above lemma, τ denotes the root of $C(A)$. Also, for any node $u \in C(A)$, $\mathsf{Ldepth}(u)$ (resp., $\mathsf{Rdepth}(u)$) denotes the number of nodes which have their left (resp., right) child, in the path from τ to u. Finally, $\mathsf{Lleaves}(u)$ denotes the number of leaf nodes $v \in C(A)$ which satisfies $v < u$.

Davoodi et al. [4] showed that all the operations used in the lemma can be computed in $O(1)$ time using the tree covering representation of $C(A)$ along with some auxiliary data structures. Furthermore, they showed that the size of S is at most $1.5n$, which implies that

there exists the data structure for answering RT2Q in $O(1)$ time using at most $3.5n + o(n)$ bits. With further optimization, they improved the space usage to $3.272n + o(n)$ bits while still supporting RT2Q in $O(1)$ time.

▶ **Example 2.** We show how to answer the $\mathsf{RT2Q}(3, 9)$ on the array $A[1, 12]$ in Figure 1 using $C(A)$ with the spine sequence S of A. First, we compute and return $\mathsf{RMQ}(3, 9) = \mathsf{LCA}(3, 9) = 5$. Next, to compare $A[3]$ and $A[7]$ (note that $\mathsf{RMQ}(3, 4) = 3$ and $\mathsf{RMQ}(6, 9) = 7$), we first locate the starting position of S_5 in S by $\sum_{j<5} m_j = 2 \cdot 5 - 3 - 3 + 0 - 0 + 1 - (5 - 2) = 2$. Since $\mathsf{depth}(5) = 0$, $\mathsf{depth}(3) = \mathsf{depth}(7) = 2$ and $\mathsf{select}_0(S_5, 2) > \mathsf{select}_1(S_5, 2)$, we return 7 as the position of the second largest element in $A[3, 9]$.

3 A Practical Implementation

Davoodi et al.'s data structure [4] in the previous section uses the tree-covering method for encoding $C(A)$, which is not practical compared to other succinct tree representations such as BP (balanced parenthesis) [13] and DFUDS (depth-first unary degree sequence) [3]. In this section, we describe a practical implementation of Davoodi et al.'s data structure for answering RT2Q on $A[1, n]$, which uses the DFUDS representation of the *2d-max heap* of A [9]. We first describe the general definition of DFUDS and 2d-max heap, and show how to convert Davoodi et al.'s data structure using these tools.

3.1 DFUDS and 2d-max heap

Given an ordinal tree T with n nodes, DFUDS of T (denoted by $D(T)$) is a balanced parenthesis sequence of size $2n$ defined as follows. (i) if $n = 1$, $D(T)$ is (). (ii) Otherwise, if T has k subtrees $T_1, T_2, \ldots, T_k$, $D(T)$ is $(^{k+1})$ followed by $d(T_1), d(T_2), \ldots, d(T_k)$, where $d(T_i)$ is $D(T_i)$ with the first open parenthesis removed (see Figure 1 for an example). Since $D(T)[1, 2n]$ is a balanced parenthesis sequence, one can define two operations $\mathsf{findopen}(i)$ / $\mathsf{findclose}(i)$ which find the matching open / closed parenthesis of the closed / open parenthesis in $D(T)[i]$. It is known that by storing a $o(n)$-bit auxiliary structure along with $D(T)$, one can support rank, select, findopen and findclose operations in $O(1)$ time. This in turn enables us to represent T to support a comprehensive list of navigation queries on T in $O(1)$ time using $2n + o(n)$ bits [16] (see Table 2 in [1] for the list of operations).

One of the main reasons for using the tree-covering based approach for representing $C(A)$ in Davoodi's et al.'s structure, is to find the i-th node in the inorder traversal of $C(A)$ (let this operation be $\mathsf{inorder}(i)$). To our best knowledge, one cannot support this operation on the BP or DFUDS of $C(A)$ (note that LCA can be supported in $O(1)$ time on both BP and DFUDS [1]). Sadakane [19] showed that (i) if the difference between any two consecutive values in A is ± 1, then one can answer RMQ on A (we refer the such query as ± 1RMQ) in $O(1)$ time using $2n + o(n)$ bits, and (ii) for general A, one can support both inorder and LCA operations on $D(C(A))$ (thus, RMQ on A) in $O(1)$ time using $4n + o(n)$ bits, by converting $C(A)$ into a ternary tree by adding a dummy leaf to each node in $C(A)$.

Fischer and Heun [9] proposed the *2d-max heap* to support RMQ without the need for the inorder operation. The 2d-max heap on A (denoted by $2dmax(A)$) is an alternative representation of $C(A)$, defined as follows. $2dmax(A)$ is an ordered tree with $n + 1$ nodes, where for $1 \leq i \leq n$,

1. The i-th node in the preorder traversal of $2dmax(A)$ corresponds to $A[i - 1]$ (we assume that $A[0] = \infty$). In the rest of this paper, we refer to this node as node $(i-1) \in 2dmax(A)$. Therefore, the root of $2dmax(A)$ is 0.
2. For any non-root node $i \in 2dmax(A)$, the parent of i is the node j where j is the rightmost position in $A[0, i - 1]$ such that $A[i] < A[j]$.

The above definition implies that for $1 \leq i \leq n$, the node $i \in C(A)$ and the node $i \in 2dmax(A)$ both correspond to the position i in A. The example of Figure 1 (a) and (b) shows the $C(A)$ and $2dmax(A)$ of input array A respectively. Fischer and Heun also showed that $\mathsf{RMQ}(i, j)$ operation can be supported in $O(1)$ time by using $D(2dmax(A))$ along with $o(n)$-bit auxiliary structures for supporting rank, select, findopen, and $\pm 1\mathsf{RMQ}$ queries on $D(2dmax(A))$ – using $2n + o(n)$ bits in total.

3.2 Practical implementation of encoding RT2Q

In this section, we propose an alternative implementation of the data structure of [4] on A using $D(2dmax(A))$. Since one can support RMQ using $D(2dmax(A))$ [9], it is enough to show how to find the position of the second largest element in $A[i, j]$. One can observe that, for any node k in $C(A)$, the nodes on the $\mathsf{linspine}(k)$ in $C(A)$ are the same as the nodes on the right spine of the previous sibling of $k \in 2dmax(A)$. (The left/right spine of a node $i \in 2dmax(A)$ is defined as the path from node i to the leftmost/rightmost descendant of i.) Also the nodes on $\mathsf{rinspine}(k)$ in $C(A)$ are the same as the children of $k \in 2dmax(A)$. We define the spine sequence S of A, analogous to the same sequence in [4] (that is, concatenating all the S_k's for each non-root node $k \in 2dmax(A)$ according to their preorder value in $2dmax(A)$). Then we can answer $\mathsf{RT2Q}(i, j)$ using the following procedure:

1. Compute and return the position $k = \mathsf{RMQ}(i, j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compute two nodes $k_l = \mathsf{presibling}(k)$ and $k_r = \mathsf{childrank}(k_2)$ in $2dmax(A)$ where $\mathsf{presibling}(k)$ denotes the previous sibling of k, and $\mathsf{childrank}(k_2)$ denotes the number of left siblings of k_2.
4. Locate the starting position of S_k in S, and return k_1 if $\mathsf{select}_0(S_k, \mathsf{depth}(k_1) - \mathsf{depth}(k_l) + 1) < \mathsf{select}_1(S_k, k_r)$, or k_2 otherwise.

Note that the operations used in the above procedure (RMQ, presibling, childrank, and depth) can be supported in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [14]. Also, to locate the position of S_k in S, we need to compute $\sum_{1 \leq j < k} m_j$ (recall that $m_j = |S_j|$). The following lemma shows that we can compute each of the terms in Lemma 1 (therefore, $\sum_{1 \leq j < k} m_j$) using $2dmax(A)$.

▶ **Lemma 3.** *Given an array $A[1, n]$ of size n where all elements in A are distinct, the following properties hold for any node k in the Cartesian tree, $C(A)$, of A.*

1. *l_k (number of nodes in $\mathsf{linspine}(k)$) is equal to the number of nodes in $\mathsf{rspine}(k_l)$ in $2dmax(A)$, where $k_l = \mathsf{presibling}(k)$.*
2. *$\mathsf{Ldepth}(k)$ is equal to the number of right siblings of all the nodes on the path from node k to the root in $2dmax(A)$.*
3. *$\mathsf{Rdepth}(k) = d_k - 1$, where d_k is the depth of $k \in 2dmax(A)$.*
4. *$\mathsf{Lleaves}(k)$ is equal the number of leftmost children $u < k$ which are also leaves in $2dmax(A)$.*

Proof.

1. Let $i_0 < k$ be the rightmost position of A which satisfies $\mathsf{RMQ}(i_0, k) \neq k$. Then by the definition of $C(A)$, $\mathsf{linspine}(k)$ is composed of the nodes $\{i_1, i_2, \ldots, i_{l_k}\}$ of $C(A)$ where $i_j = \mathsf{RMQ}(i_{j-1}+1, k-1)$. Thus, if $l_k > 0$, the node k in $2dmax(A)$ always has the previous sibling k_l (otherwise, k has no left child in $C(A)$, which implies $l_k = 0$). Furthermore, since $k - 1$ is the rightmost leaf of the subtree of $2dmax(A)$ rooted at k_l, all the nodes $i_1, i_2, \ldots, i_{l_k}$ are on the $\mathsf{rspine}(k_l)$ in $2dmax(A)$.

2. Let $Lpath(k)$ be the set of nodes in $C(A)$ which have their left child in the path from k to the root (hence, $|Lpath(k)|= \mathsf{Ldepth}(k)$). Now suppose $Lpath(k) = \{i_1, i_2, \ldots, i_{\mathsf{Ldepth}(k)}\}$ where $i_1 < i_2 < \cdots < i_{\mathsf{Ldepth}(k)}$. Then for any $j \in \{1, 2, \ldots, \mathsf{Ldepth}(k)\}$, (i) $i_j > k$, and (ii) $\mathsf{RMQ}(k, i_j) = i_j$. Therefore, for the node $k \in 2dmax(A)$, $Lpath(k)$ is the same as the set of nodes in $2dmax(A)$ which are the right siblings of the nodes on the path from the node k to the root.
3. Similar to the case of $\mathsf{Ldepth}(k)$, let $Rpath(k)$ be the set of nodes in $C(A)$ which have their right child in the path from k to the root (hence, $|Rpath(k)|= \mathsf{Rdepth}(k)$). Then $Rpath(k)$ consists all the nodes i_j in $C(A)$ which satisfy: (i) $i_j < k$, and (ii) $\mathsf{RMQ}(i_j, k) = i_j$. Thus by the definition of $2dmax(A)$, $Rpath(k)$ is the same as the set of proper ancestors of k in $2dmax(A)$.
4. Note that a node in $C(A)$ is a leaf if and only if its corresponding node in $2dmax(A)$ is a leftmost child which is also a leaf. Thus, set of all leaf nodes in $C(A)$ are the same as the set of all leftmost children $u < k$ which are also leaves in $2dmax(A)$. ◀

Now we describe how to compute each value in Lemma 1 using $D(2dmax(A))$ with Lemma 3.

1. L_τ: The node $\tau = \mathsf{RMQ}(1, n)$ is the rightmost child of the node 0 (the root of $2dmax(A)$). Also all the nodes of $\mathsf{lspine}(\tau)$ in $C(A)$ are on the left siblings of τ in $2dmax(A)$. Thus this value can be computed in $O(1)$ time by $\mathsf{degree}(0)$ (note that degree can be computed in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [1]).
2. l_k: By Lemma 3, $\mathsf{linspine}(k)$ of $C(A)$ are the same as the $\mathsf{rspine}(k_l)$ of $2dmax(A)$. Since the rightmost leaf of k_l is $k-1$, This can be computed in $O(1)$ time by $\mathsf{depth}(k-1) - \mathsf{depth}(k_l) + 1$.
3. $\mathsf{Ldepth}(k)$: For $k \in 2dmax(A)$, let $L(k)$ be the number right siblings of the nodes on the path from the node k to the root in $2dmax(A)$. Now we describe how to compute $L(k)$ using $D(2dmax(A))$. Let d be a depth of $2dmax(A)$, and suppose $f(n) = \log n \cdot g(n)$ where $g(n)$ is any increasing function which satisfies $g(n) = \omega(1)$. Then we fix the value $0 \le i < f(n)$, and define the array E which stores all the values of $L(k)$ for every node $k \in 2dmax(A)$ whose depth is $i + j \cdot f(n)$, for all $0 \le j \le \lfloor (d-i)/f(n) \rfloor$. The values in E are stored according to the preorder number of corresponding nodes in $2dmax(A)$. By a simple counting argument, we can choose i to satisfy $|E| \le n/f(n)$. Thus, at most $n/f(n) \cdot \log n = o(n)$ bits of space are necessary to store E. In addition to that, we maintain the bit array $B[1, n]$ of size n where for $1 \le i \le n$, $B[i] = 1$ if and only if the $L(i)$ is stored in E. Using the data structure of Raman et al. [18], we can store B using $\log \binom{n}{f(n)} + o(n) = o(n)$ bits while supporting rank queries in $O(1)$ time (we can also access any position of B in $O(1)$ time by two rank queries). To answer $L(k)$, we initialize the counter $c = 0$, and start the scanning nodes on $Lpath(k)$ starting from the node k. During this scan, when we are at node j, we first check $B[j]$. If $B[j] = 0$, we increase c to be $c + r$ where $r = \mathsf{degree}(\mathsf{parent}(j)) - \mathsf{childrank}(j)$ (note that parent can be computed in $O(1)$ time using $D(2dmax(A))$ [1]), and move to the parent of j. If $B[j] = 1$, we return $L(k) = c + \mathsf{rank}_1(B, j)$. Thus, using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures, we can answer $L(k)$ in $O(f(n))$ time.
4. $\mathsf{Rdepth}(k)$: By Lemma 3, this is the same as the number of proper ancestors of k in $2dmax(A)$, which can be computed $O(1)$ time by $\mathsf{depth}(k) - 1$.
5. $\mathsf{Lleaves}(k)$: By Lemma 3, this is the same as the number of leftmost children $u < k$ which are also leaves in $2dmax(A)$. This value can be computed by counting the number of occurrences of the pattern '())' before the closing parenthesis corresponding to node k in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary data structures [15].

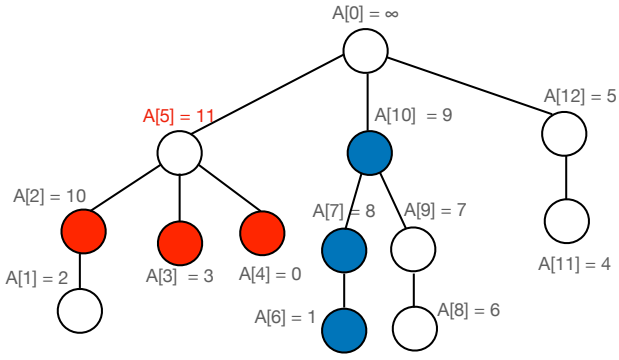


Figure 2 $r2dmax(A)$ of the array in Figure 1. Red and Blue colored nodes correspond to the nodes in $C(A)$ in Figure 1 with the same colors.

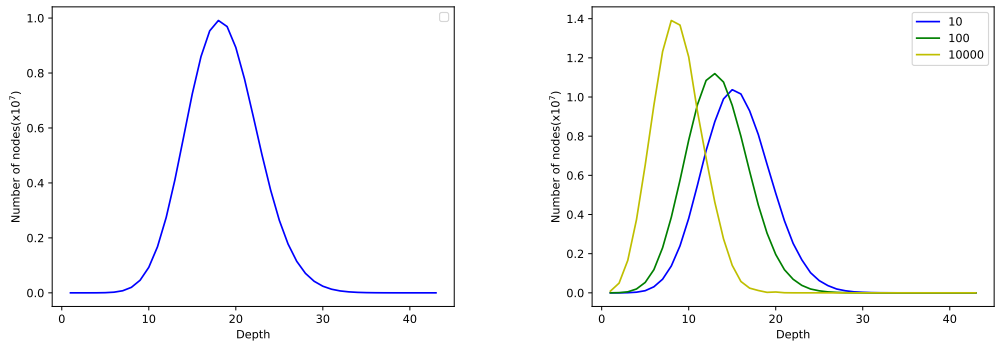
▶ **Example 4.** We show how to locate the starting position of S_5 in S using the $2dmax(A)$ in Figure 1 (b). From node $5 \in 2dmax(A)$ in the figure, one can observe that $5_l = 2$, $L(5) = 0$, and $\mathsf{select}_)(D(2dmax(A)), 5) + 1 = 12$. Also $\mathsf{degree}(0) = 3$, $\mathsf{depth}(4) - \mathsf{depth}(2) + 1 = 3$, $\mathsf{depth}(5) - 1 = 0$, and $\mathsf{rank}_{()}(D(2dmax(A)), 12) = 2$. Thus, the starting position of S_5 in S is $\sum_{j<5} m_j = 2 \cdot 5 - 3 - 3 + 0 - 0 + 1 - (5 - 2) = 2$.

We summarize the result in the following theorem.

▶ **Theorem 5.** *Given an array $A[1, n]$ of size n, RT2Q on A can be computed in $O(\log n \cdot g(n))$ time, for any increasing function $g(n) = \omega(1)$. The data structure uses at most $1.5n + o(n)$ additional bits, along with the DFUDS sequence of the 2d-max heap of A, $D(2dmax(A))$.*

Alternative representation of $2dmax(A)$. In practice, the performance of the data structure of Theorem 5 highly depends on the depth of $2dmax(A)$. To reduce the depth of $2dmax(A)$, Ferrada and Navarro [8] considered *rightmost-path* $2dmax(A)$ (denoted as $r2dmax(A)$), which can be obtained from $C(A)$ by applying τ_1 (first-child, next-sibling) transformation [5]. Note that the original $2dmax(A)$ can be obtained from $C(A)$ by applying τ_4 (previous-sibling, last-child) transformation [5]. One can observe that i-th position of A corresponds to the node in $r2dmax(A)$ whose postorder number is i. (See Figure 2 for an example.) For example, if A is strictly decreasing array from 1 to n, the depths of $2dmax(A)$ and $r2dmax(A)$ are n and 1, respectively. Ferrada and Navarro [8] showed that one can answer RMQ queries on A as using $r2dmax(A)$ with $o(n)$-bit auxiliary structures, which are different from the structures used for answering the same query using $2dmax(A)$. Baumstark et al. [2] showed that $r2dmax(A)$ is isomorphic to $2dmax(\overleftarrow{A})$, where $\overleftarrow{A}$ is an array of size n constructed by reversing the all elements of A. Thus, one can simulate the $\mathsf{RMQ}(i, j)$ on A using $r2dmax(A)$ by answering $\mathsf{RMQ}(n + 1 - j, n + 1 - i)$ on $\overleftarrow{A}$ using $2dmax(\overleftarrow{A})$ (note that in this case, one breaks the ties with rightmost policy when constructing $2dmax(\overleftarrow{A})$, i.e., among all the equal elements in a range, the rightmost element is considered as the largest).

To implement the data structure of Theorem 5, we first check the depth of $2dmax(A)$ and $2dmax(\overleftarrow{A})$ at pre-processing step, and maintain the one with smaller depth (along with the auxiliary structures).



(a) (b)

Figure 3 (a) The distribution of the depth of nodes in $2dmax(A)$. (b) The distribution of the depth of the nodes in $2dmax(A)$ which correspond to the RMQ of A.

4 Experimental results

Our data structure is implemented in C++ (compiled by g++ 9.3.0 with O3 optimization), and all the experiments were done on the Desktop PC (Intel i7-9900KS CPU with 128GB of RAM). We use the input array $A[1, n]$ which stores 32-bit unsigned integers. We consider three different types of input arrays: (a) *random*, (b) *pseudo-increasing*, and (c) *pseudo-decreasing*, where each $A[i]$ is randomly generated from the range (a) $[1, n]$, (b) $[i - \delta, i + \delta]$, and (c) $[n - i - \delta, n - i + \delta]$, respectively for given parameter $\delta > 0$. We compare the space usage (bits per element) and query time (μs) of our encoding structure (referred to as R2MQ-ENCODING) with the following four indexing data structures for answering RT2Q: (i) A + RMQ encoding of Fisher and Heun [9] (FH-DFUDS), (ii) A + RMQ encoding of Ferrada and Navarro [8] (FN-BP), (iii) A + RMQ encoding of BaumStark et al. [8] (BGHL-BP), and (iv) A + Fischer and Huen's indexing data structure for RMQ queries [9] (FH-INDEXING). Note that the encoding of (i) uses $D(2dmax(A))$, and both the encoding of (ii) and (iii) use the BP of $2dmax(A)$. For (i) and (ii), we use the implementation of Ferrada and Navarro [8][3], and for (iii), we use the implementation of BaumStark et al. [8][4]. Finally for (iv), we use our own implementation.

To support RMQ on A, and navigation queries on $2dmax(A)$ except depth, we use sdsl-lite [12] to support rank, select, findopen, findclose (for presibling operation), and $\pm$1RMQ on $D(2dmax(A))$. Note that for findopen, findclose and $\pm$1RMQ, we use a simplified RMM-tree [16] which maintains only the *min* field for these queries. For computing depth(k) queries, we use the same data structure for computing $L(k)$. More precisely, if $L(k)$ is stored in E, we also store depth(k) in a separate array E' at the same position (thus, the same bit array B can be used for $L(k)$ and depth(k)). For computing depth(k), we perform the parent query iteratively until we find the node whose depth is stored in E'. Note that we do not keep any additional data structures for both depth and $L(K)$ queries if the depth of the tree is less than $\lceil \log n \rceil$.

[3] the codes are available `https://github.com/hferrada/rmqFischerDFUDS` and `https://github.com/hferrada/rmq`.
[4] the code is available at `https://github.com/kittobi1992/rmq-experiments`

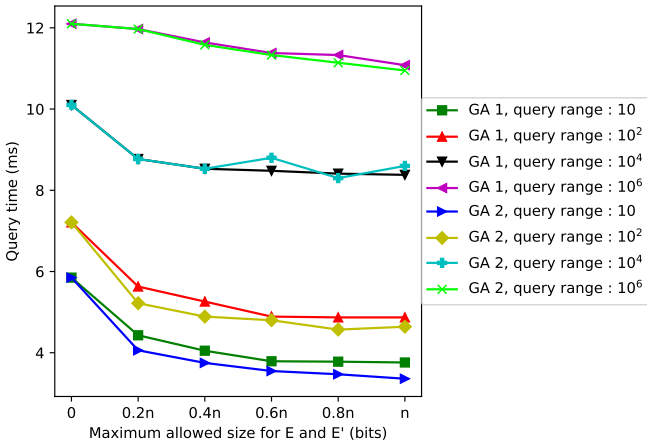


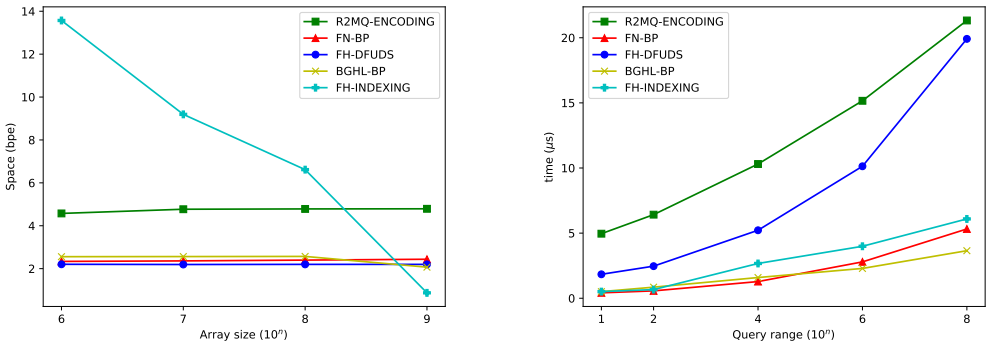
Figure 4 Query time based on the allotted space for E and E'.

Since the overhead for depth and $L(k)$ is the main drawback of our implementation, we do an empirical evaluation to decide the sizes of E and E'. When A is a randomly generated array of size 10^8, the depth of $2dmax(A)$ is less than 50 in most cases (in theory, the expected depth of $C(A)$ for a random array A is about $\Theta(\log n)$, and the depth of $2dmax(A)$ is at most the depth of $C(A)$ [6]), and the depth of nodes has close to the normal distribution (see Figure 3(a)). Next, we evaluate the distribution of the depth of the nodes $2dmax(A)$ which correspond to the RMQ of A (note that we only need the value of depth(k) and $L(k)$ when $k = \mathsf{RMQ}(i, j)$ for some $1 \leq i \leq j \leq n$). As shown in Figure 3(b), when the query range is 10^4, the depth of all the nodes corresponding to RMQ is less than half of the depth of $2dmax(A)$. Furthermore, even for the small query ranges (10), still, the depth of 95.5% of the nodes is less than half of the depth of $2dmax(A)$. From the distribution of the nodes corresponding to RMQ of A, we consider two greedy algorithms for selecting the nodes to be stored in E and E'. Suppose at most N nodes can be stored in E and E', and let D_N be the smallest depth where the number of the nodes with depth D_N is more than N (if there is no such depth, D_N is the depth of $2dmax(A)$); and let d be the value $\min(\lfloor D/2 \rfloor, D_N - 1)$, where D is the depth of $2dmax(A)$. Then the greedy algorithm 1 (GA1) repeats the following procedure from $i = 0$ to d:

1. Choose all the nodes with depth i, if the total number of chosen nodes is at most N.
2. Increase i by 1.

Similarly, the greedy algorithm 2 (GA2) repeats the first step of the above procedure by decreasing the value i from d to 0.

We evaluate the time for answering RT2Q with different amounts of space allotted for E and E'. As shown in Figure 4, increasing the allotted space does not significantly improve the query time when the size of the query range is 10^6 since most of the nodes corresponding to the answer of RMQ are close to the root node. The same tendency is shown for other sizes of query ranges (10, 10^2, and 10^4) when allotting more than $0.4n$ bits for E and E', since both GA1 and GA2 cannot significantly increase the number of nodes to be stored (note that the number of nodes increases roughly exponentially with the depth, from 1 to d). In our implementation, we choose GA2 which shows better query time for small query ranges. Also, the space allotted for storing E and E' is determined based on the maximum values



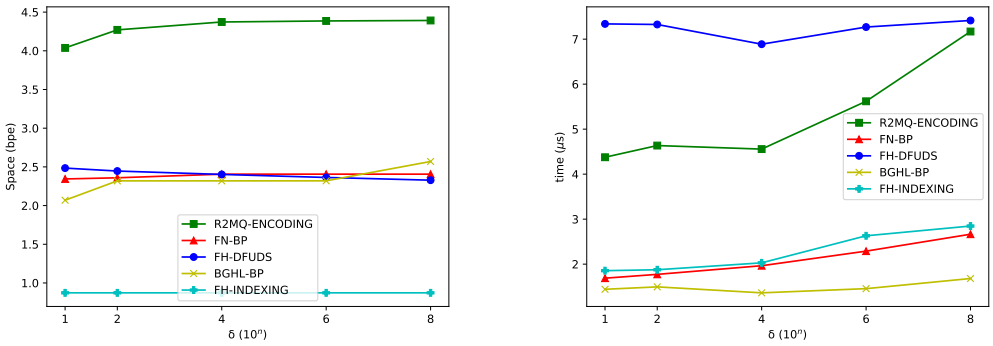
(a) **(b)**

Figure 5 (a) The space (without the input array) on random array, and (b) Query time on the random array of size 10^9.

(either 8, 16 or 32-bit values) stored in those arrays, as follows. We allot $n/\lfloor \log\log n \rfloor$ bits if the maximum values of E and E' are both at most 2^8 (in this case, we use 8-bit integer arrays for storing these). In general, if the maximum values of E and E' are at most 2^{8c} and $2^{8c'}$, respectively, for some $c, c' \in \{1, 2, 4\}$, then we allot $(\frac{c+c'}{2})n$ bits for storing these arrays. For example, if 32 and 8-bit integer arrays are necessary to store E and E' respectively for an array A of size 10^8, we use $(\frac{4+1}{2}) \cdot n/\lfloor \log\log 10^8 \rfloor = 0.625n$ bits for storing E and E'.

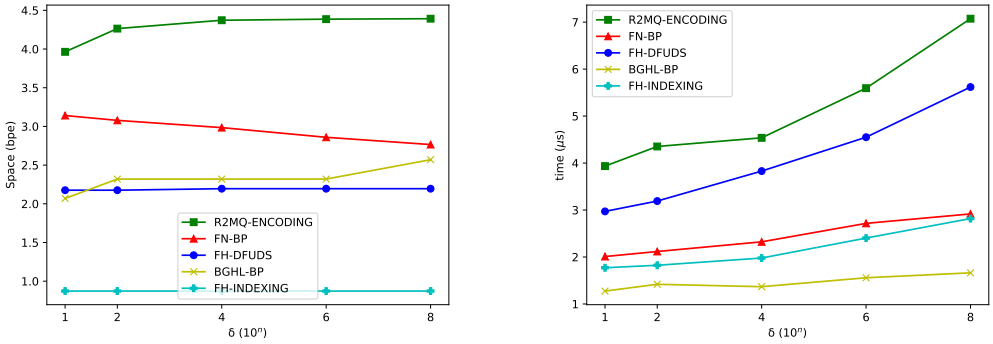
Next, we evaluate the space usage on randomly generated arrays of size $n = 10^7$ to $n = 10^9$ (see Figure 5 (a)). Our structure uses up to 4.6 and 4.8 bpe (bits per element) for $n = 10^7$ and $n = 10^9$ respectively. This shows that our data structure's average space is not much changed by increasing the array size, like other indexing structures except FH-INDEXING. For FH-INDEXING, each pre-computed value needs 32 bits even for an array of size 10^6 (note that $\lceil \log 10^6 \rceil$ is 19), which is wasteful in terms of space. Since the input array is necessary to answer RT2Q queries using indexing data structures, our data structure takes at least 7.1 times less space than the existing indexing data structures. Next, we fix the size of the (randomly-generated) input array to be 10^9, and evaluate query time for various query ranges (see Figure 5 (b)). Our data structure and FH-DFUDS are highly dependent on the query range, compared to BP-based indexing structures. This is because, in the implementation, the running time of findopen operation is an increasing function of the range (note that findopen operation is used for computing RMQ, depth, and $L(k)$ when 2d-max heap is represented by DFUDS). Interestingly, when the query range is changed from 10^6 to 10^8, the query time of FH-DFUDS increases much rapidly than our data structures. This shows that the overhead for answering RMQ on FH-DFUDS is more than computing $L(k)$ and depth(k) for the nodes with small depths. The query time on FH-INDEXING is also rapidly increased by increasing the query range because the structure needs to access more sub-structures when the query range increases. Compared to the fastest indexing solution (BGHL-BP and FH-INDEXING), our data structure shows up to 10 and 4.1 times slower query times when the query range is 10 and 10^8 respectively and shows better time-space trade-off for most cases except the small query ranges up to 100.

Next, we evaluate the space and query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ with various δ values (see Figure 6 and 7. the size of the query range is fixed to $\sqrt{n}$.). Note that when A is pseudo-increasing (resp. pseudo-decreasing), $\overleftarrow{A}$ is pseudo-decreasing (resp. pseudo-increasing). Thus, our data structure and BGHL-BP show



(a) (b)

Figure 6 The (a) space usage (without the input array) and (b) query time on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

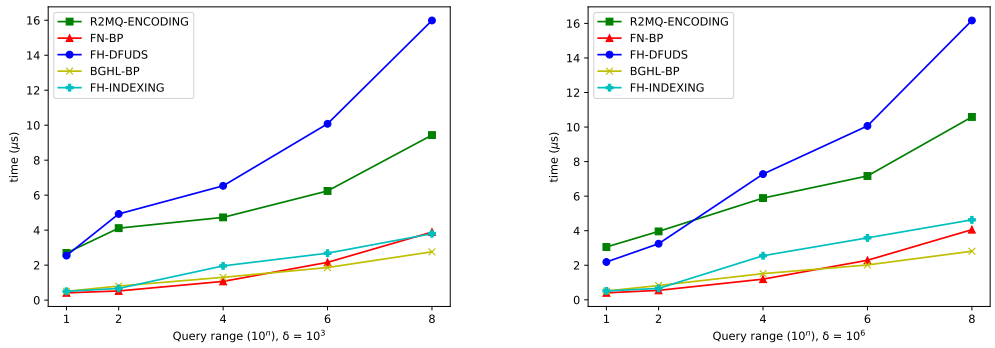


(a) (b)

Figure 7 The (a) space usage (without the input array) and (b) query time on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil \sqrt{n} \rceil = 31623$.

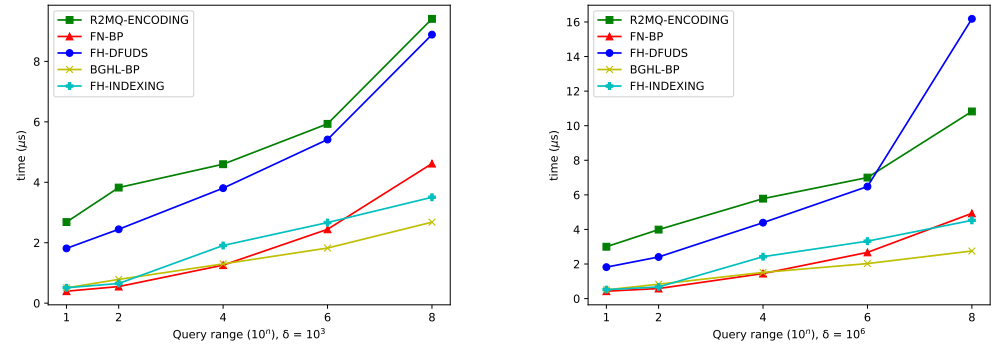
similar space usage and query time on both pseudo-increasing and pseudo-decreasing arrays (note that for FH-DFUDS, the query time on pseudo-increasing arrays is up to 3 times slower than the query time on pseudo-decreasing arrays because of the depth of $2dmax(A)$. Note that the average distance between two matching parenthesis in DFUDS decreases proportional to the depth of $2dmax(A)$). The space usage of our data structure is not much affected by δ (up to 4.03 bpe to 4.39 bpe) since we do not maintain the arrays E and E' for all the cases (the depth of $2dmax(A)$ is still less than $\log 10^9 \sim 30$ even for large $\delta = 10^8$). Also, the query time for our data structure increases with δ because the average depth of the nodes corresponding to RMQ is increases with δ. Overall, our data structure shows better time-space trade-off (takes up to 7.5 times less space while spending up to 4.2 times slower the query time) than all other indexing data structures in the evaluation.

Finally, for $\delta = 10^3$ and 10^6, we evaluate the query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ for various query ranges (see Figure 8 and 9). Again, DFUDS-based implementations (R2MQ-ENCODING and FH-DFUDS) highly depend on the depth of $2dmax(A)$ and query ranges because of findopen operation, whereas the BP-based implementations (FN-BP and BGHL-BP) have similar results compared to the random array



(a) **(b)**

Figure 8 Query time on the pseudo-increasing array with (a) $\delta = 10^3$, and (b) $delta = 10^6$.



(a) **(b)**

Figure 9 (Query time on the pseudo-decreasing array with (a) $\delta = 10^3$, and (b) $delta = 10^6$.

case. Especially compared to the random array case, our data structure supports much faster (up to 2.2 times) queries on pseudo-increasing and decreasing arrays for most query ranges since the extra overhead for accessing E and E' does not occur for both cases.

5 Conclusion

In this paper, we propose a practical implementation of an encoding for answering RT2Q queries. Our data structure takes much less space than the current indexing data structure implementations, while still giving better time-space trade-off for most cases in practice. An interesting open problem is to implement the data structure based on the BP of $2dmax(A)$ – here, an efficient and practical implementation of degree and childrank queries would be a challenging problem.

References

1 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proceedings of ALENEX 2010*, pages 84–97, 2010.

2 Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In *SEA 2017*, pages 12:1–12:16, 2017.

3 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

4 Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A*, 372(2016):20130131, 2014.

5 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Math. Comput. Sci.*, 11(2):177–189, 2017.

6 Luc Devroye. On random cartesian trees. *Random Struct. Algorithms*, 5(2):305–328, 1994.

7 A. Farzan and J. I. Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, January 2014.

8 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017.

9 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

10 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of STOC 1984*, pages 135–143, 1984.

11 Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top-k, selection, and min-max. In *ICALP 2015, Proceedings, Part I*, pages 593–604, 2015.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

13 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS 1989*, pages 549–554, 1989.

14 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

15 J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

16 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

17 Rajeev Raman. Encoding data structures. In *WALCOM 2015*, pages 1–7, 2015.

18 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

20 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.