Combined Data Structure for Previous- and Next-Smaller-Values

Johannes Fischer

Karlsruhe Institute of Technology, Institut für Theoretische Informatik, 76131 Karlsruhe, Germany

Abstract

Let A be a static array storing n elements from a totally ordered set. We present a data structure of optimal size at most $n \log_2(3 + 2\sqrt{2}) + o(n)$ bits that allows us to answer the following queries on A in constant time, without accessing A: (1) previous smaller value queries, where given an index i, we wish to find the first index to the left of i where A is strictly smaller than at i, and (2) next smaller value queries, which search to the right of i. As an additional bonus, our data structure also allows to answer a third kind of query: given indices $i < j$, find the position of the minimum in $A[i..j]$. Our data structure has direct consequences for the space-efficient storage of suffix trees.

Key words: Succinct Data Structures, Compressed Text Indexing

1. Introduction

We consider the situation where a static array $A[1, n]$ can be preprocessed such that the following three queries can be answered in constant time: previous- and next-smaller-value-queries, where given a position i in A, one searches for the next position p to the left (or right) of i with $A[p] < A[i]$, and range minimum queries, where for two given indices i and j we look for the position of the minimum element within the subarray $A[i..j]$.

Our work is situated in the field of succinct data structures, where the aim is to store objects of size n from a universe of size $L(n)$ in $\lg L(n) + (1 + o(1))$ bits[1], while still being able to perform all operations on they data as if they were uncompressed. All succinct data structures work in the word-RAM model of computation, where fundamental operations on a contiguous field of w bits can be performed in constant time (w is the word size, and we assume $\lg n = O(w)$).

Succinct data structures can be further classified into *indexing* and *encoding* data structures. An indexing data structure enhances an object (such as an array) with additional functionality (such as queries) and needs access to the object itself, whereas an encoding data structure recodes all necessary parts of the data for answering the queries without accessing the object.

Email address: `Johannes.Fischer@kit.edu` (Johannes Fischer)

[1]Throughout this article, lg denotes the binary logarithm.

For range minimum queries alone, there is a data structure in the encoding model of asymptotically optimal size $2n + o(n)$ bits that allows to answer queries in constant time [6]. Previous- and next-smaller-value queries originate from parallel computing [2]. For all three queries combined, the only existing data structure uses $3n + o(n)$ bits [16].

In this short note, we present an encoding data structure of size at most $n \lg(3+2\sqrt{2})+o(n) \approx 2.54n+o(n)$ bits that allows to answer all three queries in constant time. It is interesting to note that although we do not have a closed formula for the exact size of our data structure, we prove that it is asymptotically optimal. The reason for this slight oddity is that we are not aware of a closed formula for the size L of the universe of objects that we encode; however, we prove that we encode them in an asymptotically optimal way.

Although our data structure is independent of the underlying array A and may have other applications, our research is clearly motivated by the compact storage of full-text indices [15]. Precisely, we show that our data structure automatically yields the smallest compressed suffix tree with constant-time navigation (we refer the reader to Sect. 4 for more details and preliminary work on compressed suffix trees).

The rest of this note is structured as follows. Sect. 2 introduces some notation and known results. Sect. 3 presents the core idea of the paper, a combined data structure for s and PSV-/NSV-queries. Finally, Sect. 4 describes how that data structure yields improvements in compressed suffix trees.

2. Preliminaries

For integers i and j, we write $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$, and (i, j) to denote $\{i+1, \ldots, j-1\}$. For a rooted tree $\mathcal{T}$ and a node v, we write $\mathcal{T}_v$ to denote the subtree of $\mathcal{T}$ rooted at v.

2.1. Queries

Let $A[1, n]$ be an array of totally ordered objects. For technical reasons, we define $A[0] = -\infty = A[n+1]$ as the "artificial" overall minima of the array. We start by formally defining previous smaller values:

Definition 1. For $1 \leq i \leq n$, let $\text{PSV}_A(i) = \max\{j < i \: : \: A[j] < A[i]\}$ denote the previous smaller value of position i.

As mentioned in the introduction, we also consider next smaller values and range minima, for completeness formally defined as follows.

Definition 2. For $1 \leq i \leq n$, let $\text{NSV}_A(i) = \min\{j > i \: : \: A[j] < A[i]\}$ denote the next smaller value of position i.

Definition 3. For $1 \leq i \leq j \leq n$, let $\text{RMQ}_A(i) = \text{argmin}\{A[k] \: : \: i \leq k \leq j\}$ denote a range minimum query between positions i and j. If the minimum in the query range is not unique, the leftmost (or rightmost) minimum is chosen as a representative.

In the following, the subscript A from $\mathtt{RMQ}_A$ etc. will be omitted if the underlying array A is clear from the context.

2.2. LRM-Trees

LRM-Trees are the basis of our new data structure. They were introduced under this name as an internal tool for basic navigational operations in ordinal trees [21], and, under the name of "2d-Min Heaps," to encode integer arrays in order to support range minimum queries on them [6].

Definition 4 (Sadakane and Navarro [21]; Fischer [6]). *The* LRM-Tree *of A is an ordered labeled tree with vertices $0, \ldots, n$. For $1 \le i \le n$, $\mathtt{PSV}(i)$ is the parent node of i. The children are ordered in increasing order from left to right.*

We note the following useful properties of the LRM-Tree (observe that we use nodes and array indices interchangeably throughout this article):

Lemma 5 (Fischer [6]). *Let $\mathcal{T}$ be the LRM-Tree of A.*

1. *The node labels correspond to the preorder-numbers of $\mathcal{T}$ (counting starts at 0).*

2. *Let i be a node in $\mathcal{T}$ 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 $\mathcal{T}$ with children $x_1, \ldots, x_k$. Then $A[x_j] \le A[x_{j-1}]$ for all $1 < j \le k$.*

2.3. Succinct Tree Encodings

A rooted ordered tree on n nodes can be encoded in $2n + o(n)$ bits in various ways such that it still permits (the simulation of) all navigational operations in constant time, such as BPS [13] or DFUDS [1]. Of particular importance to this article are methods based on tree covering (TC) [9, 11, 5]. They support most navigational operations on trees in constant time, among others ROOT(), PARENT(u), FIRSTCHILD(u), NEXTSIBLING(u), SUBTREESIZE(u), selecting the i'th child (ITHCHILD(u, i)), computing the rank of a child among its siblings (CHILDRANK(u)), and computing lowest common ancestors (LCA(u, v)). Farzan and Munro's approach [5] has the further advantage that it can also optimally encode other types of trees, such as those described in the following section.

2.4. Schröder Trees

The term *Schröder Tree* is used for various types of rooted ordered trees [22]: trees with no nodes of out-degree 1, trees with labeled edges, or trees with labeled nodes. For our purposes, we define them as follows.

Definition 6. *A* Schröder Tree *is a rooted ordered tree, where any node except the first child in a list of siblings may be colored red or blue. First children are always colored blue.*

The number of Schröder Trees on n nodes is counted by the little Schröder numbers S_n. Although we do not have a closed formula for S_n, it is known [12] that $S_n = \frac{\rho^n}{\sqrt{\pi n(2n-1)}}(1 + O(n^{-1}))$ with $\rho := 3 + 2\sqrt{2}$. In particular, $S_n \leq \rho^n$.

3. Data Structure

In this section, we present the new data structure for answering RMQ/PSV/NSV on an input array A. We start by introducing the general ideas behind our data structure, and then show how this data structure can be encoded succinctly.

3.1. Basic Solution

The LRM-Tree (Def. 4) encodes all information for answering PSV-queries in a natural way, as it suffices to move to the parent node of i for answering PSV(i). It also captures all sufficient information for answering RMQs:

Lemma 7 (Fischer [6]). *For arbitrary nodes i and j in the LRM-Tree of A, $1 \leq i < j \leq n$, let $\ell = \mathrm{LCA}(i,j)$. Then if $\ell = i$, RMQ(i,j) is given by i, and otherwise, RMQ(i,j) is given by the child of ℓ that is on the path from ℓ to j.*

Thus, it remains to show how NSV-queries can be answered. It is easy to see that the LRM-Tree alone is not enough for this task: consider $A = [0,0]$ and $A' = [1,0]$. These arrays have the same LRM-Tree (and hence the same answers to all RMQs and PSV-queries); yet, their NSV-queries differ, as $\mathrm{NSV}_A(1) = 3$, and $\mathrm{NSV}_{A'}(1) = 2$.

In principle, we could build another LRM-Tree $\mathcal{T}^{\mathrm{R}}$ on the reversed sequence A^{R} for answering NSV=queries, as $\mathrm{NSV}_A(i) = n - \mathrm{PSV}_{A^{\mathrm{R}}}(n-i+1) + 1$. As this would double the space of the resulting data structure, we now present a more sophisticated solution.

The general idea of our data structure can be seen as follows. Recall property 3 of Lemma 5: the children $x_1, \ldots, x_k$ of a node v in the LRM-Tree are ordered decreasingly from left to right: $A[x_1] \geq A[x_2] \geq \cdots \geq A[x_k]$. Now suppose we wish to calculate $\mathrm{NSV}(x_i)$ for some $1 \leq i \leq k-1$, and assume that $A[x_i] > A[x_{i+1}]$. Then $\mathrm{NSV}(x_i) = x_{i+1}$, as all A-values in the subtree $\mathcal{T}_{x_i}$ are strictly greater than at position x_i (property 2 of Lemma 5). If, on the other hand, $A[x_i] = A[x_{i+1}]$, then the next "candidate" for $\mathrm{NSV}(x_i)$ is x_{i+2} (assuming $i \leq k-2$), as again all A-values in $\mathcal{T}_{x_{i+1}}$ are strictly greater than $A[x_{i+1}] = A[x_i]$.

This suggests the following general approach. In the LRM-Tree $\mathcal{T}$ of A, a node is colored *red* if the corresponding value in A is smaller than the A-value at its left sibling (if such a sibling exists). More formally, let v be a node in $\mathcal{T}$ with children $x_1, \ldots, x_k$. Then for all $2 \leq i \leq k$, node x_i is colored red if and only if $A[x_i] < A[x_{i-1}]$. All other nodes (including the root) are colored blue. We call the resulting tree the *Colored LRM-Tree.*

To get the connection to NSV-queries, we need the following definition:

Definition 8. *Let $\mathcal{T}^C$ the Colored LRM-Tree of $A[1,n]$, and let v be a node in $\mathcal{T}^C$ with children $x_1, \ldots, x_k$. The* next red sibling $\text{NRS}(x_i)$ *of a node x_i is the leftmost sibling to the right of x_i that is colored red. If such a sibling does not exist, we define $\text{NRS}(x_i) = \perp$. In symbols, let $M = \{i < j \leq k \; : \; x_j$ is colored red$\}$. Then $\text{NRS}(x_i) = \perp$ if $M = \emptyset$, and otherwise $\text{NRS}(x_i) = x_{\min M}$.*

We can then show the following lemma:

Lemma 9. *Let $\mathcal{T}^C$ the Colored LRM-Tree of $A[1,n]$, and let v be a node in $\mathcal{T}^C$ with children $x_1, \ldots, x_k$, $x_1 \leq x_2 \leq \cdots \leq x_k$. Then*

$$
\text{NSV}(x_i) = \begin{cases} \text{NRS}(x_i) & \text{if } \text{NRS}(x_i) \neq \perp \\ x_k + \text{SUBTREESIZE}(x_k) & \text{otherwise.} \end{cases}
$$

Proof. We consider each case in turn.

$\text{NRS}(x_i) \neq \perp$. Let j be defined by $x_j = \text{NRS}(x_i)$. From Def. 8 and the fact that node x_j is red, we know that $A[x_j] < A[x_i]$. Hence, we need to show that $A[h] \geq A[x_i]$ for all $h \in (x_i, x_j)$. From property 1 of Lemma 5, we know that all values in (x_i, x_j) occur in $\mathcal{T}^C_{x_i}, \ldots, \mathcal{T}^C_{x_j-1}$. Because j is minimal and due to property 3 of Lemma 5, $A[h] = A[x_i]$ for $h = x_{i+1}, \ldots, x_{j-1}$. But due to property 2 of Lemma 5, $A[h] > A[x_i]$ for all $h \in [x_\alpha + 1, x_{\alpha+1} - 1]$ and all $i \leq \alpha \leq j - 1$. Hence, $\text{NSV}(x_i) = x_j$.

$\text{NRS}(x_i) = \perp$. Let $y = x_k + \text{SUBTREESIZE}(x_k)$. As above, we can show that $A[h] \geq A[x_i]$ for all $x_i < h < y$. It thus remains to show that $A[y] < A[x_i]$. For the sake of contradiction, assume that $A[y] \geq A[x_i]$, where we further distinguish between the cases "=" and ">." If $A[y] = A[x_i]$, then $\text{PSV}(y) = v$ (the parent node of x_i), so y is the right sibling of x_k, a contradiction to the definition of x_k. If $A[y] > A[x_i] = A[x_k]$, then again due to property 2 of Lemma 5, we have $\text{PSV}(y) \in [x_k, y - 1]$. So $\mathcal{T}^C_{x_k}$ contains y, a contradiction to the size of $\mathcal{T}^C_{x_k}$, which is $y - x_i$, as $\mathcal{T}^C_{x_k}$ contains exactly those elements from $[x_k, y)$.

$\square$

3.2. Succinct Encoding

We represent the Colored LRM-Tree $\mathcal{T}^C$ from Sect. 3.1 similar to Farzan and Munro's succinct TC-encoding for ordinal trees [5]. This approach is based on a two-level decomposition of the tree into mini- and micro-trees. In our scenario, the encoding of a micro-tree is simply its index in an enumeration of all Schröder Trees of the micro-tree size (called "enumeration code" in [5]). In total, this uses optimal $\lg S_n + o(n)$ bits of space.

It remains to show how we implement the query algorithms for RMQ, PSV, and NSV.

As $\text{PSV}(i) = \text{PARENT}(i)$ and the parent-operation is directly supported by TC, we can directly focus on NSV. Recall Lemma 9: given i, we need to find $\text{NRS}(i)$ in order to answer $\text{NSV}(i)$. The NRS-method can be implemented as the combination of *modified* ITHCHILD- and CHILDRANK-operations, as they are described by Farzan and Munro [5] (see [4, p. 23] for further details). In particular, given node i, we find the parent p of i, and then determine the rank r of i among all its red siblings, from where we select the $r + 1$'st red node. To this end, if p is a root of a mini- or micro-tree, we use a *modified* fully indexable dictionary (FID) [18] to rank/select among the red nodes. These FIDs are similar to the ones already stored at each mini- or micro-tree root, with the difference that they index only the red nodes. Similar to the original analysis, their overall space contributes only $o(n)$ bits to the final space. If, on the other hand, p is not a mini- or micro-tree root, we use the lookup-tables stored along with the micro-trees to rank/select among the red nodes. These lookup-tables also use only $o(n)$ bits, as we use micro-trees of size $O(\log_\rho n/4)$. Finally, if $\text{NRS}(i) = \bot$, we move to the rightmost sibling j of i and count the subtree size at j; both operations are supported in $O(1)$ time by TC.

For implementing $\text{RMQ}(i, j)$, we have to show how the operations in Lemma 7 can be performed in constant time. We cannot resort to the method described by Fischer [6], as it is inherently connected to DFUDS. We thus do the following: first compute $\ell = \text{LCA}(i, j)$; this is supported by TC [9, 11]. Then if $\ell \neq i$ (otherwise we return i), compute the depth d of ℓ (depth is supported by TC). Finally, compute the child of ℓ that is on the path to j by a level-ancestor query $\text{LAQ}(j, d + 1)$ (supported by TC); this is the answer.

Theorem 1. *For an array of n totally ordered objects, there is a data structure using $\lg S_n + o(n) \leq n \lg(3 + 2\sqrt{2}) + o(n) \approx 2.54n + o(n)$ bits of space that supports RMQs, PSV- and NSV-queries on A in $O(1)$ time, without accessing A at query time.*

3.3. Optimality

It is easy to see that the encoding from Sect. 3.2 is optimal. Given any data structure $\mathscr{D}_A$ supporting PSV and NSV on some underlying input array A, we can reconstruct the Colored LRM-Tree $\mathcal{T}^C$ of A, without knowing A: We first create $\mathcal{T}^C$'s rightmost path $n = x_1, x_2, \ldots, x_k = 0$ in a bottom-up manner, by successively querying $x_{i+1} = \text{PSV}(x_i)$, until arriving at $x_k = 0$. All nodes are initially colored blue. This leaves us with unprocessed intervals $[x_i + 1, x_{i+1} - 1]$, which are handled recursively. During these recursive calls, suppose that a query $\text{PSV}(v)$ brings us to a node u which is already present in the (partial) LRM-Tree $\mathcal{T}^C$. Let w be the smallest child of u greater than v (i.e., the leftmost child of u to the right of v). We then check if $\text{NSV}(v) = w$, in which case we color w red. Otherwise ($\text{NSV}(v) > w$), w remains blue, as in this case $A[v] = A[w]$. This procedure correctly reconstructs the Colored LRM-Tree $\mathcal{T}^C$ of A.

As every Schröder Tree is also a Colored LRM-Tree for some array A (starting at the root with children $x_1, \ldots, x_k$, set $A[x_k]$ to 0, and $A[x_{i-1}]$ to $A[x_i]$ or $A[x_i] + 1$, depending on whether x_i is colored blue or red;

the unprocessed intervals are handled recursively), we need at least $\lg S_n$ bits to encode $\mathscr{D}_A$ in the worst case. This proves the optimality of the data structure from Thm. 1.

4. Application to Compressed Suffix Trees

The result from Thm. 1 has direct consequences for compressed suffix trees (CSTs). A suffix tree (ST) for a string S of length n is a compact trie storing all the suffixes of S, in the sense that the characters on any root-to-leaf path spell out exactly a suffix. The ST is an extremely important data structure with applications in exact or approximate string matching, bioinformatics, and document retrieval, to mention only a few examples.

The drawback of STs is their huge space consumption of 20–40 times the text size ($O(n \lg n)$ bits in theory), even when using carefully engineered implementations. To reduce their size, in recent years several authors provided compressed variants of STs [14, 10, 20, 19, 8, 17, 3, 16, 7].

We regard the CST as an abstract data type supporting the following operations (apart from the usual navigational operations on trees as those mentioned in Sect. 2.3): LEAFCOUNT(u) gives the number of leaves (suffixes) below u, LEAFLABEL(u) for a leaf u yields the position in S where the corresponding suffix begins, STRINGDEPTH(u) gives u's string-depth (number of characters on the root-to-u path), SUFFIXLINK(u) gives the unique node v with root-to-v label $\alpha \in \Sigma^\star$ if the root-to-u label is $a\alpha$ for some $a \in \Sigma$, and CHILD(u, a) gives the child v of u such that the label on the edge (u, v) starts with $a \in \Sigma$. Here and in the following, Σ denotes the underlying alphabet of size σ. See the first column of Tbl. 1 for all operations (*level ancestor queries* are excluded as we are not a aware of any actual algorithm that needs them in a suffix tree).

A CST on S can be divided into three components: (1) the *suffix array* SA, specifying the lexicographic order of S's suffixes, defined by $S_{\mathsf{SA}[1]..n} < S_{\mathsf{SA}[2]..n} < \cdots < S_{\mathsf{SA}[n]..n}$ (hence SA captures information on the *leaves*); (2) the *LCP-array* LCP, storing the lengths of the *longest common prefixes* of lexicographically adjacent suffixes: $\mathsf{LCP}[1] = -1$ and for $2 \leq i \leq n$, $\mathsf{LCP}[i] = \max\{k \geq 0 : S_{\mathsf{SA}[i]..\mathsf{SA}[i]+k-1} = S_{\mathsf{SA}[i-1]..\mathsf{SA}[i-1]+k-1}\}$, which is the string-depth of the LCA of the lexicographically i'th and $i-1$'st suffix (hence LCP captures information on *internal* nodes); and (3) additional data structures for simulating the *navigational operations*. The goal of a CST is to compress each of these three components.

We do not discuss here the different time/space tradeoffs for compressing SA and LCP; we just mention that both can be compressed into space proportional to the entropy of the underlying text, at the cost of increased access times, which we denote by t_{SA} and t_{LCP}, respectively.

Of more interest to us is the fact that most recent CSTs [8, 17, 16] represent a node v as an interval $[v_l : v_r]$ in LCP and base their navigation on RMQs and PSV-/NSV-queries in LCP. There are two basic strategies for supporting these queries: we can either use structures of size $o(n)$ [8, 3] or $2n + o(n)$ [17] bits and substitute "missing information" by a (sub-)logarithmic number of lookups to LCP (indexing model),

Table 1: Comparison of different CSTs (space in bits on top of SA and LCP). The $O(\cdot)$ is omitted in all operations. Trees [19, 8, 17] are incomparable to our approach, as they use less space in exchange for higher navigation times. t_ψ denotes the time to compute the position of $\mathsf{SA}[\cdot]+1$ in SA, which is $O(1)$ in most compressed suffix arrays.

	[19]	**[8, 3]**	**[14, 10, 20]**	**[17]**	**[16]**	**NEW**
space	$o(n)$	$o(n)$	$4n$	$2n$	$3n$	$\mathbf{2.54n}$
Root	1	1	1	1	1	**1**
IsAncestor	1	1	1	1	1	**1**
SubtreeSize	—	—	1	—	—	—
LeafCount	1	1	1	1	1	**1**
LeafLabel	$\lg^{1+\alpha} n$	t_{SA}	t_{SA}	t_{SA}	t_{SA}	$\boldsymbol{t_{\mathsf{SA}}}$
StringDepth	$\lg^{1+\alpha} n$	t_{LCP}	t_{LCP}	t_{LCP}	t_{LCP}	$\boldsymbol{t_{\mathsf{LCP}}}$
Parent	$\lg^{1+\alpha} n$	$t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n$	1	$t_{\mathsf{LCP}}\lg\sigma$	1	**1**
FirstChild	$\lg^{1+\alpha} n$	$t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n$	1	t_{LCP}	1	**1**
NextSibling	$\lg^{1+\alpha} n$	$t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n$	1	t_{LCP}	1	**1**
SuffixLink	$\lg^{1+\alpha} n$	$t_\psi + t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n$	t_ψ	$t_\psi + t_{\mathsf{LCP}}\lg\sigma$	t_ψ	$\boldsymbol{t_\psi}$
Lca	$\lg^{1+\alpha} n$	$t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n$	1	$t_{\mathsf{LCP}}\lg\sigma$	1	**1**
TreeDepth	$\lg^{2+2\alpha} n$	—	1	—	—	—
Child	$\lg\sigma + \lg^{1+\alpha} n$	$t_{\mathsf{LCP}}\,\mathrm{polylglg}\,n + t_{\mathsf{SA}}\lg\sigma$	$t_{\mathsf{SA}}\lg\sigma$	$t_{\mathsf{SA}}\lg\sigma$	$t_{\mathsf{SA}}\lg\sigma$	$\boldsymbol{t_{\mathsf{SA}}\lg\sigma}$

resulting in increased navigation time (see 3rd and 5th column Tbl. 1). The other option [16] is to use a data structure that computes RMQ/PSV/NSV without needing access to the underlying LCP-array (encoding model).

Given these observations, the index from Thm. 1 almost directly yields a CST with $\approx 2.54n + o(n)$ bits on top of SA and LCP with constant-time support of all operations that do not necessarily need access to SA or LCP. See again Tbl. 1 for a comparison. In particular, we get the smallest CST with constant-time navigation. Note that it is of utmost theoretical and practical importance to have the smallest possible data structure for the navigational component of a CST, as its $O(n)$-term is incompressible, whereas the space of the other two components of a CST (SA and LCP) vanishes if the entropy of the underlying text does.

All suffix tree operations (apart from LeafCount, StringDepth, and Child) from Tbl. 1 can be implemented solely by performing RMQs and PSV-/NSV-queries in LCP, see [8, 16]. Only the implementation of the NextSibling-operation relies on structures that are proprietary to [16] (and the one in [8] accesses LCP); we therefore give our own implementation as follows: let $v = [v_l : v_r]$ be the node whose next sibling we want to compute. First check if v equals the root, and return NULL in this case. Otherwise, compute $w = [w_l : w_r] = \text{Parent}(v)$. If $v_r = w_r$, return NULL, as v does not have a right sibling in this case. We

now know that $v_r + 1$ is the leftmost index of NEXTSIBLING(v). To determine the rightmost index, check if NSV$(v_r + 1) = w_r + 1$, and return $[v_r + 1, w_r]$ in this case, as then v is the second-to-last child of w. Otherwise, return $[v_r + 1, \text{RMQ}(v_r + 2, w_r) - 1]$, as the range minimum query returns a position in LCP where the string-depth of w is stored.

Theorem 2. *Let S be a text of size n with characters from an alphabet of size σ. Given S's suffix array with access time t_{SA} and its LCP-array with access time t_{LCP}, there is a CST with additional $\lg S_n + o(n) \leq 2.54n + o(n)$ bits that supports the operations as indicated in the last column of Tbl. 1.* $\qquad\square$

Our CST resides in between [17] and [16]: it is smaller than [16] and larger than [17], but equally fast as the larger of these [16].

It is interesting to note that our $\lg S_n \approx 2.54n$ bits are also optimal for encoding the topology of a suffix tree, as it is a tree with exactly n leaves and no nodes of out-degree 1; the number of such trees is also counted by the little Schröder number S_n. However, we cannot make an optimality claim for the CST from Thm. 2, as it builds on SA and LCP, who already capture the topology of the suffix tree.

References

[1] 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.

[2] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 14(3):344–370, 1993.

[3] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Proc. SEA*, volume 6049 of *LNCS*, pages 94–105. Springer, 2010.

[4] A. Farzan. *Succinct Representation of Trees and Graphs*. PhD thesis, University of Waterloo, Canada, 2009. Available at `https://uwspace.uwaterloo.ca/bitstream/10012/4832/1/main.pdf`.

[5] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. SWAT*, volume 5124 of *LNCS*, pages 173–184. Springer, 2008.

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

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

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

[9] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[10] 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.

[11] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. ICALP*, volume 4596 of *LNCS*, pages 509–520. Springer, 2007.

[12] D. Merlini, R. Sprugnoli, and M. C. Verri. Waiting patterns for a printer. *Discrete Applied Mathematics*, 144(3):359–373, 2004.

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

[14] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

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

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

[17] E. Ohlebusch and S. Gog. A compressed enhanced suffix array supporting fast string matching. In *Proc. SPIRE*, volume 5721 of *LNCS*, pages 51–62. Springer, 2009.

[18] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242. ACM/SIAM, 2002.

[19] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully-compressed suffix trees. In *Proc. LATIN*, volume 4957 of *LNCS*, pages 362–373. Springer, 2008.

[20] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

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

[22] R. P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.