CST++

Enno Ohlebusch[1], Johannes Fischer[2], and Simon Gog[1]

[1] Universität Ulm, Institut für Theoretische Informatik, 89069 Ulm, Germany
{enno.ohlebusch,simon.gog}@uni-ulm.de
[2] KIT, Institut für Theoretische Informatik, 76131 Karlsruhe, Germany
johannes.fischer@kit.edu

Abstract. Let A be an array of n elements taken from a totally ordered set. We present a data structure of size $3n + o(n)$ bits that allows us to answer the following queries on A in constant time, without accessing A: (1) given indices $i < j$, find the position of the minimum in $A[i..j]$, (2) given index i, find the first index to the left of i where A is strictly smaller than at i, and (3) same as (2), but to the right of the query index. Based on this, we present a new compressed suffix tree (CST) with $O(1)$-navigation that is smaller than previous CSTs. Our data structure also provides a new (practical) approach to compress the LCP-array.

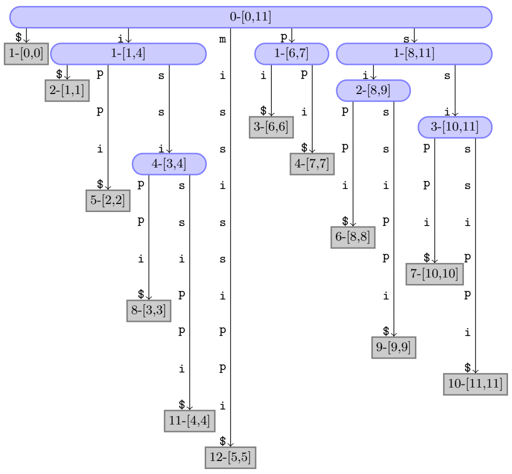
1 Introduction

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. An example is shown in Fig. 1. 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 (CSTs), both in theory [1,2,3,4,5,6,7] and in practice [8,9,10].

We regard the CST as an abstract data type supporting the following operations (u and v are nodes): Root() yields the root, IsAncestor(u, v) is true iff u is an ancestor of v, Count(u) gives the number of leaves (suffixes) below u, LeafLabel(u) for leaf u yields the position in S where the corresponding suffix begins, SDepth(u) gives u's string-depth (number of characters on root-to-u path), Parent(u)/FChild(u)/NSibling(u) yields the parent/first child/next sibling of u (if existent), SLink(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 $a \in \Sigma$, Lca(u, v) yields the lowest common ancestor of u and v, TDepth(u) gives the tree depth of u, and Child(u, a) gives the child v of u such that the label on edge (u, v) starts with $a \in \Sigma$. Here and in the following, Σ denotes the underlying alphabet of size σ.

In all CSTs, there is a trade-off between the *space* it occupies and the *time* it needs to support the operations from the previous paragraph. We first make an extensive survey of existing approaches in the following section (along with some



i	SA	LCP	$S_{\mathsf{SA}[i]}$
0	11	-1	`$`
1	10	0	`i$`
2	7	1	`ippi$`
3	4	1	`issippi$`
4	1	4	`ississippi$`
5	0	0	`mississippi$`
6	9	0	`pi$`
7	8	1	`ppi$`
8	6	0	`sippi$`
9	3	2	`sissippi$`
10	5	1	`ssippi$`
11	2	3	`ssissippi$`
12		-1	

Fig. 1. Left: suffix tree for $S =$ `mississippi$`. Right: suffix- and LCP-array.

simplifications and unifying observations); our technical results and an outline on the rest of this article will be given at the end of that survey.

2 A Guided Tour through Compressed Suffix Trees

Let S_i denote the i'th suffix of S. 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 $S_{\mathsf{SA}[0]} < S_{\mathsf{SA}[1]} < \cdots < S_{\mathsf{SA}[n-1]}$ (hence SA captures information about the *leaves*); (2) the *LCP-array* LCP, storing the lengths of the *longest common prefixes* of lexicographically adjacent suffixes: $\mathsf{LCP}[0] = -1 = \mathsf{LCP}[n]$ and for $1 \leq i < n$, $\mathsf{LCP}[i] = \max\{k \geq 0 \ : \ S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i-1]}$ share a length-k-prefix$\}$ (hence LCP captures information about *internal* nodes); and (3) additional data structures for simulating the *navigational operations* mentioned in the introduction. The goal of a CST is to compress each of these three components.

Compressed Suffix Arrays. There is a wealth of literature on Compressed Suffix Arrays (CSAs), offering different trade-offs between the space they require and the lookup-time t_{SA} they provide. We refer the reader to Navarro's and Mäkinen's survey paper [11] for an in-depth overview of the different alternatives. One of the best choices is the CSA due to Grossi et al. [12], achieving $t_{\mathsf{SA}} = O(\lg^\epsilon n)$ with space $(1 + \frac{1}{\epsilon})nH_k + o(n)$ bits (assuming an alphabet of size $\sigma = O(\text{polylg}\, n)$). Here and in the following, H_k denotes the *empirical entropy* of order k of the input text S, which, at least in the realm of text indexing, is used as the *de facto* standard for the compressibility of a text [11].

Compressed LCP-Arrays. There are also different options for compressing the second component (LCP-array), for convenience summarized in Tbl. 1. Although we focus primarily on theoretical results, we also included the simple byte-aligned variant [13] in Tbl. 1, as it is still competitive [10] with recent

Table 1. Trade-offs for storing and accessing the LCP-array. $^{(*)}$worst case $O(n \lg n)$.

ref.	space in bits	t_{LCP}	comment
[14]	$n \lg n$	$O(1)$	uncompressed variant
[13]	$8n^{(*)}$	$O(1)$	practical byte-aligned variant
NEW	**4–$6n^{(*)}$**	**$O(1)$**	**only in conjunction with our new CST**
[3]	$2n + o(n)$	$O(t_{\mathsf{SA}})$	
[4]	$nH_k + o(n)$	$O(\lg^{1+\alpha} n)$	constant $0 < \alpha < 1$
[5]	$O(nH_k \lg \frac{1}{H_k})$	$O(t_{\mathsf{SA}})$	
[15]	$O(\frac{n \lg n}{q})$	$O(t_{\mathsf{SA}} \cdot q)$	time amortized; $O(t_{\mathsf{SA}} \cdot n)$ time in the worst case
[7]	$O(\frac{n}{\lg \lg n})$	$O(t_{\mathsf{SA}} \lg^{\beta} n)$	constant $0 < \beta < 1$
[16]	$O(nH_k \lg \frac{1}{H_k})$	$O(t_{\psi} \frac{n}{R^{1-\gamma}})$	R = number of equal-letter runs in BWT of S

practical implementations [8,9]. Apart from the first two [14,13], all other variants exploit the redundancy arising from listing the LCP-values in *text* order (as opposed to *suffix array* order); this is sometimes called the *permuted LCP-array* [15,16]. Note in particular that all LCP-variants with $o(n \lg n)$ bits have non-constant access-time t_{LCP} (assuming an underlying *compressed* SA).

In principle, implementations of suffix- and LCP-arrays are interchangeable in CSTs. Hence, combining the different variants mentioned above already yields a rich variety of CSTs, although some CSTs favor certain suffix- or LCP-arrays.

Succinct Tree Navigation. The real difference between the various known CSTs lies in the way they support the navigational operations (third component above), which we are going to discuss next. There are two main lines of research:

(a) Storing the tree topology *explicitly* by a sequence of balanced parentheses (either BPS [17] or DFUDS [18]).
(b) Deriving the tree topology *implicitly* from intervals in the LCP-array as follows: let v be a node in ST such that the labels on the root-to-v path form the string ω of length ℓ. Then there is an interval $[v_l..v_r]$ in the suffix array such that ω is the longest common prefix of $S_{\mathsf{SA}[v_l]}, \ldots, S_{\mathsf{SA}[v_r]}$, and ω in not a prefix of any other suffix. In the LCP-array, this interval is called an *LCP-interval of LCP-value* ℓ [13]. It has the properties $\mathsf{LCP}[v_l] < \ell$, $\mathsf{LCP}[v_r + 1] < \ell$, and $\mathsf{LCP}[k] \geq \ell$ for all k with $v_l < k \leq v_r$. The indices $i_1 < i_2 < \cdots < i_m$ in $[v_l..v_r]$ with $\mathsf{LCP}[i_j] = \ell$ (where $1 \leq m \leq \sigma - 1$) are called LCP-indices, and the child intervals of $[v_l..v_r]$ are $[v_l..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_m..v_r]$, where some of them may be singleton intervals $[i_j..i_j]$.

Tbl. 2 summarizes all known existing CSTs, and shows those with type-(a)-navigation in the left half [1,2,3,4], and those of type (b) in the right half [5,6].

Approach (a), pursued either verbatim [1,2,3] or on a subset of certain sampled nodes [4], has the advantage that the rich results on navigation in general-purpose succinct trees (see [17,18,19,20] and references therein) can be re-used for CSTs. Hence, whenever a new operation for BPS or DFUDS is discovered, this automatically carries over to CSTs using this representation.

Table 2. Comparison of different CSTs (space in bits on top of CSA & LCP). The $O(\cdot)$ is omitted in all operations. Trees with type-(a) navigation are in the left half.

	[1, 2, 3]	**[4]**	**[5]**	**[6]**	**NEW**
space	$4n$	$o(n)$	$o(n)$	$2n$	$\boldsymbol{3n}$
Root, IsAncestor, Count	1	1	1	1	**1**
LeafLabel	t_{SA}	$\lg^{1+\alpha} n$	t_{SA}	t_{SA}	$\boldsymbol{t_{\mathsf{SA}}}$
SDepth	t_{LCP}	$\lg^{1+\alpha} n$	t_{LCP}	t_{LCP}	$\boldsymbol{t_{\mathsf{LCP}}}$
Parent	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	$t_{\mathsf{LCP}} \lg \sigma$	**1**
FChild, NSibling	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	t_{LCP}	**1**
SLink, Lca	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	$t_{\mathsf{LCP}} \lg \sigma$	**1**
TDepth	1	$\lg^{2+2\alpha} n$	t_{LCP} polylglg n	1	**1**
Child	$t_{\mathsf{SA}} \lg \sigma$	$\lg \sigma + \lg^{1+\alpha} n$	$t_{\mathsf{SA}} \lg \sigma$	$t_{\mathsf{SA}} \lg \sigma$	$\boldsymbol{t_{\mathsf{SA}} \lg \sigma}$

The most prominent CST from group (a) is the one due to Sadakane [3], called Sad-CST henceforth. Sad-CST supports very fast operations, but its disadvantage is that it needs up to $4n + o(n)$ bits for the sequence of parentheses, as the ST consists of n leaves and up to $n - 1$ additional internal nodes (the $o(n)$-term comes from the extra data structures for navigation). There are at least two explanations why the resulting $4n$ bits are a waste of space. First, a suffix tree is not an arbitrary tree, but a *compact* one, meaning that it does not contain nodes of out-degree 1. Thus, in principle it should be possible to represent such a compact tree on n' nodes with less than $2n'$ bits ($n' < 2n$ is the number of nodes in ST). The second (and even more convincing) reason is that the LCP-array itself already captures the topology of the ST by means of LCP-intervals. Hence, in theory no space at all has to be spent for the topology, as the LCP-array is already part of any CST.

Motivated by the observations from the previous paragraph, more recent CSTs [5,6] base their navigation on intervals in the LCP-array LCP. CSTs from this (b)-group express all navigational operations by a combination of three basic queries on LCP: (i) range minimum queries (`RMQ`), where for two given indices $i \leq j$ in LCP one seeks the (first) position of the minimum in $\mathsf{LCP}[i, j]$, (ii) previous smaller value queries (`PSV`), where for an index i in LCP one searches for the rightmost position to the left of i where LCP is strictly smaller than at position i, and (iii) next smaller value queries (`NSV`), which are defined analogously for the sub-array to the right of the query point.

One could use separate data structures for each of the three queries mentioned above (`RMQ`/`PSV`/`NSV`), amounting to $6n + o(n)$ bits in total: $2n + o(n)$ for RMQs [21] and $2n + o(n)$ each for `PSV` and `NSV` [5]. This, however, would constitute a disadvantage over the $4n + o(n)$ bits used by the methods storing the explicit topology (a). To cope with this situation, Fischer et al. [5] proposed to "sparsify" the `RMQ`/`PSV`/`NSV`-structures to use only $o(n)$ bits in total, thereby giving up constant-time retrieval of `RMQ`/`PSV`/`NSV`-values. This, nonetheless, constitutes no theoretical slowdown for the suffix-tree operations, as they have to make at least one lookup to LCP, which already costs $\Omega(t_{\mathsf{SA}}) = \Omega(\lg^{\epsilon} n)$ at the very best in the presence of a compressed LCP-array (see Tbl. 1).

A different idea to reduce the $6n$-bit term is to use a *combined* data structure for RMQ/PSV/NSV. This is the idea of Ohlebusch and Gog's OG-CST [6], who noted that a $2n$-bit balanced parentheses representation BPR of LCP's Super-Cartesian Tree [22] provides this functionality, at least for RMQ and NSV. Although BPR is not defined in mathematically rigorous terms, the authors provide an algorithm that constructs BPR in linear time [6, Alg. 3]. It works by scanning LCP from left to right, and writing '$)^k($' in step i if $\mathsf{LCP}[i]$ is the NSV of k preceding positions. This results in a $2n$-bit sequence BPR that supports RMQ and NSV in $O(1)$ time (using additional $o(n)$ bits); PSV-queries are supported in $O(t_{\mathsf{LCP}} \lg \sigma)$ time by a binary search over the at most σ closing parentheses (which are consecutive), taking the LCP-values of the corresponding positions as search keys. A different (practical) proposal [9] of a combined data structure is based on the *range min-max tree* [20].

The parentheses sequence BPR from OG-CST has an interesting connection to a seemingly different data structure, the 2d-Min-Heap [21]: BPR is DFUDS of the 2d-Min-Heap of LCP read from right to left (reversed). To see why this is so, recall the definition of the 2d-Min-Heap $\mathcal{M}_A$ of an array A [21, Def. 1]: it is an ordered tree on nodes $\{1, 2, \ldots, n\}$, defined such that i is the parent of j iff $A[i]$ is the PSV of $A[j]$. Writing the DFUDS of $\mathcal{M}_A$, where a node with k children is encoded as '$(^k)$', results in a sequence of parentheses where node i appears as '$(^k)$' if $A[i]$ is the PSV of k succeeding positions (as opposed to NSV in BPR). Given these similarities, it is also not surprising that the construction algorithms for BPR [6, Alg. 3] and the DFUDS of $\mathcal{M}_{\mathsf{LCP}}$ [21, Sect. 4] are strikingly similar.

Unifying View. The separation between CSTs with navigation of type (a) and those of type (b) is actually not as strict as it may seem at first sight. Indeed, if for (a) we restrict ourselves to the BPS, then the resulting sequence $U[1, 2n']$ *implicitly* lists the depths of the ST-nodes as visited in an Euler-Tour. For such a sequence U, Sadakane and Navarro [20] showed that the efficient support of operations very similar to (if not the same as) RMQ/PSV/NSV yields all known navigational operations in the underlying tree. The only conceptual difference is that the underlying BPS U lists the *tree*-depths of ST, whereas type-(b)-navigation works on LCP, which lists the *string*-depths (a similar observation is made by [9]).

Construction. Particular emphasis has been put on efficient construction algorithms for all three components of CSTs. Here, "efficiency" encompasses both construction *time* and *space*, as the latter can cause a significant memory bottleneck. Most CSAs can be built in $O(n)$ time and $O(n)$ bits (constant alphabet), or $O(n \lg \lg \sigma)$ time using $O(n \lg \sigma)$ bits (arbitrary alphabet size σ) of space [23].

The $2n + o(n)$-bit LCP-array [3] can be constructed with no extra space in addition to CSA and the text [8]. Other smaller LCP-variants from Tbl. 1 would need these $2n$ bits as their intermediate working space.

Concerning the navigational component, the most space-consuming part is the sequence of balanced parentheses, either that of the ST (Sad-CST), or of the Super-Cartesian Tree (OG-CST). For Sad-CST, it has been shown how to

Algorithm 1. Space-efficient construction of the DFUDS U of a suffix tree

```
$push(n+1)$            /* $\mathsf{LCP}[n+1] = -1$ */
for $i \leftarrow n$ downto 1 do
  write ')' to $U$'s current beginning          /* accounts for leaf $\mathsf{SA}[i]$ */
  while $\mathsf{LCP}[i] < \mathsf{LCP}[top()]$ do
    write '(()' to $U$'s current beginning          /* node with $\geq 2$ children */
    $\lambda \leftarrow pop()$
    while $\mathsf{LCP}[\lambda] = \mathsf{LCP}[top()]$ do
      write '(' to $U$'s current beginning          /* additional children */
      $\lambda \leftarrow pop()$
  $push(i)$
write '(' to $U$'s current beginning          /* to make $U$ balanced */
```

construct the BPS in $O(n)$ time using $O(n)$ bits of working space [24, 8]. However, these algorithms are quite complex (using Elias δ-codes, batched updates, ...) and involve large big-O-constants, and are therefore not used in existing implementations [8, 9].

There is a simpler way if we construct ST's DFUDS U instead of its BPS, shown in Alg. 1.[1] As in previous approaches [24, 8], we construct U from the LCP-array of S. We scan LCP from *right to left* and build the DFUDS from *back to front*. Note the similarity of our algorithm to the construction of the balanced parentheses representation of LCP's Super-Cartesian Tree [6, Alg. 3] (and to the algorithm for constructing the DFUDS of LCP's 2d-Min-Heap [21, Sect. 4.1]).

The correctness follows from the fact that if v is a node in ST with k children $v_1, \ldots, v_k$, then there are $k-1$ LCP-indices $p_1, \ldots, p_{k-1}$ in v's LCP-interval $[v_l..v_r]$. These indices p_j must have v_l as their PSV; hence, when scanning position $i = v_l$ in Alg. 1, we write k opening parentheses (2 in the outer and $k-2$ in the inner while-loop). The outer while-loop accounts for the fact that v_l could be the left delimiter of several nested LCP-intervals with decreasing LCP-values.

The only drawback of Alg. 1 is that the stack might still use $O(n \lg n)$ bits in the worst case. To cope with such a situation, Fischer [21, Sect. 4.2] shows how to represent a stack containing at most n in- or decreasing elements from $[1, n]$ with a bit-vector of length n, assuming that elements are only pushed in a right-to-left (or left-to-right) manner. To retain constant-time access to the elements on the stack, we need three further tables of size at most $O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits. Thus, $n + o(n)$ bits of working space suffice for constructing ST's DFUDS.

It is interesting to note that if one substitutes the '(()' by a simple '(' in line 3 of Alg. 1, then the result is again the DFUDS of LCP's 2d-Min-Heap [21]! This shows yet another interesting connection between the triumvirate suffix tree/Super-Cartesian Tree/2d-Min-Heap.

[1] Note that BPS and DFUDS support the same set of operations [25], so we can choose whatever representation is more suitable for our purposes.

2.1 Our Results in Context

In Sect. 3, we first show a general result that is independent from CSTs:

Theorem 1. *Given an array A of n elements taken from a totally ordered set, there is a data structure of size $3n+o(n)$ bits that supports* RMQ/PSV/NSV*-queries on A in $O(1)$ time, without needing access to A at query time.*

In Sect. 4, we then use Thm. 2 to improve upon OG-CST:

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 $3n + o(n)$ bits that supports* Count, IsAncestor, Parent, FChild, NSibling, SLink, Lca *and* TDepth *in $O(1)$,* LeafLabel *in t_{SA},* SDepth *in $O(t_{\mathsf{LCP}})$, and* Child *in $O(t_{\mathsf{SA}} \lg \sigma)$ time.*

This latter result should be seen in the context of other CSTs; see again Tbl. 2. In terms of space, our new CST resides between Sad-CST with $4n$ additional bits [3] and OG-CST with $2n$ bits [6]. However, it is equally fast as the larger of these (Sad-CST). Clearly, there are smaller variants of CSTs [4,5], but due to their increased navigation time they are incomparable to Thm. 2.

Finally, we briefly sketch how the data structure from Thm. 2 yields a practicable and small LCP-array.

3 A New Representation of Super-Cartesian Trees

In this section, we prove Thm. 1 (with linear construction time).

Definition 1. *Let $A[1..n]$ be an array of elements of a totally ordered set $(M, \leq)$. To deal with boundary cases, we add an element $-\infty$ to M which is smaller than any other element of M, and define $A[0] = -\infty = A[n+1]$. For an index $1 \leq i \leq n$, we define:*

$$\mathtt{FEV}(i) = \min\{k \mid \mathtt{PSV}(i) < k < \mathtt{NSV}(i) \text{ and } A[k] = A[i]\}$$
$$\mathtt{LEV}(i) = \max\{k \mid \mathtt{PSV}(i) < k < \mathtt{NSV}(i) \text{ and } A[k] = A[i]\}$$

An interval $[i..j]$, where $1 \leq i \leq j \leq n$, is called mound-interval *if $A[k] > \max\{A[i-1], A[j+1]\}$ for all k with $i \leq k \leq j$.*

Let us call the interval $[\mathtt{PSV}(i) + 1..\mathtt{NSV}(i) - 1]$ the mound-interval of i. Then, $\mathtt{FEV}(i)$ ($\mathtt{LEV}(i)$) is the first (last) index in the mound-interval of i at which a value equal to $A[i]$ can be found. As an example consider index 5 in the array A from Fig. 2: Its mound-interval is $[1..11]$ because $\mathtt{PSV}(5) = 0$ and $\mathtt{NSV}(5) = 12$. Furthermore, $\mathtt{FEV}(5) = 1$ and $\mathtt{LEV}(5) = 8$.

The definition of the Super-Cartesian tree is taken from [22]; cf. Fig. 3.

Definition 2. *Let $A[l..r]$ be an array of elements of a totally ordered set $(M, \leq)$ and suppose that the minima of $A[l..r]$ appear at positions $p_1 < p_2 < \cdots < p_k$ for some $k \geq 1$. The* Super-Cartesian tree $\mathcal{C}^{sup}(A[l..r])$ *of $A[l..r]$ is recursively constructed as follows:*

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$A[i]$	-1	0	1	1	4	0	0	1	0	2	1	3	-1

Fig. 2. $A[1..11]$ is an array of natural numbers. We choose -1 as the element that is smaller than every natural number and set $A[0] = -1 = A[12]$.

- *If $l > r$, then $\mathcal{C}^{sup}(A[l..r])$ is the empty tree.*
- *Otherwise create k nodes $v_1, v_2, \ldots, v_k$, label each v_j with p_j, and for each j with $1 < j \le k$ the node v_j is the right sibling of node v_{j-1} (in Fig. 3, node v_{j-1} is connected with v_j by a* horizontal *edge). Node v_1 is the root of $\mathcal{C}^{sup}(A[l..r])$. Recursively construct $\mathcal{C}_1 = \mathcal{C}^{sup}(A[l..p_1 - 1])$, $\mathcal{C}_2 = \mathcal{C}^{sup}(A[p_1 + 1..p_2 - 1])$, $\ldots$, $\mathcal{C}_{k+1} = \mathcal{C}^{sup}(A[p_k + 1..r])$. For each j with $1 \le j < k$, the left child of v_j is the root of $\mathcal{C}_j$. The left and right children of v_k are the roots of $\mathcal{C}_k$ and $\mathcal{C}_{k+1}$, respectively.*

Ohlebusch and Gog [6] showed that the Super-Cartesian tree of $A[1..n]$ can be represented by a sequence of balanced parentheses BPR, and they give a construction algorithm that is solely based on A. However, the sequence BPR lacks some information, as the cases "right child" and "right sibling" are treated in the same fashion, and the array A itself is required to compensate for this. Here, we will compensate for the lack of information by enhancing the BPR with a bitstring B of length $n + 2$. A closing parentheses corresponding to a node that is a right sibling is marked with a 0, otherwise it is marked with a 1. The construction of the enhanced BPR of the Super-Cartesian tree of array A starts at the root of the tree and proceeds as follows (see Fig. 4 for an example):

1. Write the balanced parentheses sequence of the left child.
2. Write an opening parenthesis.
3. Write the balanced parentheses sequence of the right child/sibling.
4. Write a closing parenthesis. If the node under consideration is a right sibling, append 0 to B; otherwise append 1 to B.

Similar to [6, Alg. 3], the enhanced BPR of (the Super-Cartesian tree of) an array can be constructed *without* knowing the Super-Cartesian tree itself; see Alg. 2. Again, the stack can be implemented with $n + o(n)$ bits [21, Sect. 4.2].

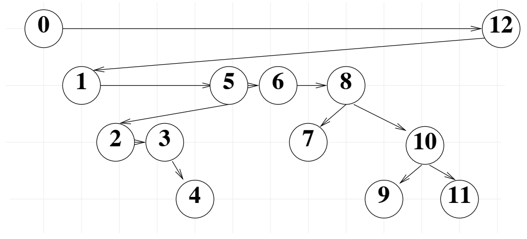


Fig. 3. The Super-Cartesian tree of the array A from Fig. 2

```
                 1  0  1           1        1        1  1  0  0  0  1     0  1
  (  (  (  (  (  )  )  )  (  (  (  )  (  (  )  (  (  )  )  )  )  )  )  (  )  )
  0  1  2  3  4  4  3  2  5  6  7  7  8  9  9 10 11 11 10  8  6  5  1 12 12  0
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Fig. 4. Enhanced BPR of the Super-Cartesian tree of Fig. 3. The lower row of numbers shows the positions of the parentheses in the sequence. The row below BPR is only for illustrative purposes: The opening parentheses are numbered consecutively and a closing parenthesis has the number of its matching opening parenthesis. The row above the BPR shows the bitstring B.

A few thoughts on the space of our data structure: For an array of n elements, there are $2n + 4$ parentheses and the bitstring has length $n + 2$. Note that the class of Super-Cartesian trees is isomorphic to the class of Schröder Trees, whose number is given by the n'th Small Schröder Number $\mathcal{C}_n$, and $\lg \mathcal{C}_n \approx 2.54n - \Theta(\lg n)$ [22]. Hence, although our $3n$-bit representation does not meet this lower bound, it is also not too far away from it.

Given a balanced parentheses sequence, the following operations can be supported in $O(1)$ time with $o(n)$ bits of extra space (see [17] and references therein): $\mathit{rank}_{(}(i)$ returns the number of opening parentheses up to position i; $\mathit{select}_{(}(i)$ returns the position of the i-th opening parenthesis; $\mathit{findclose}(i)$ returns the position of the closing parenthesis matching the opening parenthesis at position i; and $\mathit{enclose}(i)$ for an opening parenthesis at position i returns the position j of the opening parenthesis such that $(j, \mathit{findclose}(j))$ encloses $(i, \mathit{findclose}(i))$ most tightly. Operations $\mathit{rank}_{)}(i)$, $\mathit{select}_{)}(i)$, and $\mathit{findopen}(i)$ are defined analogously.

Let us denote the i-th opening parenthesis by ${}_i($ and the matching closing parenthesis by $)_i^b$, where b is its mark (bit in B). Note that ${}_i($ and $)_i^b$ occur at positions $\mathit{ipos} = \mathit{select}_{(}(i)$ and $\mathit{cipos} = \mathit{findclose}(\mathit{ipos})$ in BPR, respectively. Moreover, b occurs at position $\mathit{bcipos} = \mathit{rank}_{)}(\mathit{cipos})$ in the bitstring B. Vice versa, given the position bpos in B, its corresponding index in A can be determined by $\mathit{rank}_{(}(\mathit{findopen}(\mathit{select}_{)}(\mathit{bpos})))$.

Algorithm 2. Construction of the enhanced BPR of an array A.

```
push(0)          /* $A[0] = -\infty$ */
$B \leftarrow \varepsilon$            /* bitstring $B$ is initially empty */
write '('
for $i \leftarrow 1$ to $n+1$ do
  while $A[i] < A[top()]$ do
    $\lambda \leftarrow pop()$
    write ')'
    if $A[\lambda] = A[top()]$ then append 0 to the bitstring $B$
    else append 1 to the bitstring $B$
  push($i$) and write '('
write '))' and append 01 to the bitstring $B$          /* for $A[0]$ and $A[n+1]$ */
```

Algorithm 3. Finding $\mathtt{FEV}(i)$ and $\mathtt{LEV}(i)$ in constant time.

```
$cipos \leftarrow findclose(select_{(}(i))$
$bcipos \leftarrow rank_{)}(cipos)$
if $\mathsf{BPR}[cipos-1] = $"(" or ($\mathsf{BPR}[cipos-1] = $")" and $B[bcipos-1] = 1$) then
  $\mathtt{LEV}(i) \leftarrow i$
else        /* $\mathsf{BPR}[cipos-1] = $")" and $B[bcipos-1] = 0$ */
  $blpos \leftarrow select_1(rank_1(bcipos) - 1) + 1$
  $\mathtt{LEV}(i) \leftarrow rank_{(}(findopen(select_{)}(blpos)))$
if $B[bcipos] = 1$ then
  $\mathtt{FEV}(i) \leftarrow i$
else        /* $B[bcipos] = 0$ */
  $brpos \leftarrow select_1(rank_1(bcipos) + 1)$
  $\mathtt{FEV}(i) \leftarrow rank_{(}(findopen(select_{)}(brpos)))$
```

The enhanced BPR has the following crucial property. If $j_1 < j_2 < \cdots < j_m$ are the indices in the mound-interval of i such that $A[i] = A[j_k]$, then $)^0_{j_m} \ldots)^0_{j_2})^1_{j_1}$ form a contiguous subsequence in BPR (note that the order is reversed, so that $\mathtt{LEV}(i) = j_m$ is first and $\mathtt{FEV}(i) = j_1$ is last), where the first $m-1$ closing parentheses are marked 0 and the last one is marked 1. This allows us to compute $\mathtt{FEV}(i)$ and $\mathtt{LEV}(i)$ by a case distinction (see Alg. 3): If $)^b_i$ is preceded by ${}_i($ or by $)^1_j$, then $\mathtt{LEV}(i) = i$. Otherwise, $)^b_i$ is preceded by $)^0_j$. In this case, we search for the position *bpos* of the first 1 in B that is left to *bcipos* (to ensure that there is always such a 1, an additional 1 is added at the beginning of B). The index corresponding to $blpos = bpos + 1$ is $\mathtt{LEV}(i)$. If $b = 1$ (i.e., $B[bcipos] = 1$), then $\mathtt{FEV}(i) = i$. Otherwise, $b = 0$ and we search for the position *brpos* of the first 1 in B that is right to *bcipos*. The index corresponding to *bcipos* is $\mathtt{FEV}(i)$.

As a matter of fact, Alg. 3 also allows us to determine the *number* of values that are equal to $A[i]$ in the mount interval of i: this is $brpos - blpos + 1$, and we can access each of them in constant time!

It follows from the construction of the enhanced BPR that the values $\mathtt{PSV}(i)$ and $\mathtt{NSV}(i)$ can also be computed in constant time:

- $\mathtt{PSV}(i) = rank_{(}(enclose(select_{(}(\mathtt{FEV}(i))))$
- $\mathtt{NSV}(i) = rank_{(}(findclose(select_{(}(i))) + 1$

Furthermore, it is proved in [6] that a (general) RMQ can be computed in constant time on the BPR. However, the hidden constant is quite large, so that this operation is rather slow in practice. In other words, one should avoid its use whenever possible. Specific RMQ's (on mound-intervals) can be answered quicker as we shall see next. The proof of the following lemma is straightforward.

Lemma 1. *If $[i..j]$ is a mound-interval in array A, then $A[i-1] > A[j+1]$ if $\mathtt{NSV}(i-1) = j+1$ and $A[i-1] \leq A[j+1]$ otherwise.*

At first glance, Lemma 1 is sort of weird. However, if the array A is compressed and access to its entries takes more time than the computation of $\mathtt{NSV}(i-1)$, then a use of Lemma 1 makes perfect sense.

Lemma 2. *Let $[i..j]$ be a mound-interval in array A. Then,*

$$\texttt{RMQ}(i,j) = \begin{cases} \mathit{rank}_((\mathit{findopen}(\mathit{findclose}(\mathit{select}_((i-1))-1)), \text{ if } \texttt{NSV}(i-1)=j+1 \\ \mathit{rank}_((\mathit{findopen}(\mathit{select}_((j+1)-1)), \text{ otherwise} \end{cases}$$

Proof. Let $k = \texttt{RMQ}(i,j)$. If $\texttt{NSV}(i-1) = j+1$, then $A[i-1] > A[j+1]$ by Lemma 1. In this case, k is the right child of $i-1$ in the Super-Cartesian tree of the array A. In the BPR, $)_k$ is directly followed by $)_{i-1}$. Thus, $k = \mathit{rank}_((\mathit{findopen}(\mathit{findclose}(\mathit{select}_((i-1))-1))$. Otherwise, $A[i-1] \leq A[j+1]$. In this case, k is the left child of $j+1$ in the Super-Cartesian tree of A. In the BPR, $)_k$ is directly followed by $(_{j+1}$. Therefore, $k = \mathit{rank}_((\mathit{findopen}(\mathit{select}_((j+1)-1))$.

4 New Compressed Suffix Tree

We now show how to use the result from Sect. 3 for CSTs. Our basis is OG-CST [6], but we use the enhanced BPR for RMQ/PSV/NSV. It remains to show how to simulate the operations that access LCP in OG-CST; all other operations (including level ancestor queries, if needed) can be taken from [6, 5]. Note that an LCP-interval is a mound-interval that includes its leftmost delimiting point.

- Parent($[v_l..v_r]$) returns $\perp$ if $v = [v_l..v_r]$ is the root. If v is not the root, it returns $[\texttt{PSV}[k]..\texttt{NSV}[k]-1]$, where $k = v_l$ if $\texttt{NSV}[v_l] = v_r + 1$ (i.e., $\mathsf{LCP}[v_l] > \mathsf{LCP}[v_r+1]$ by Lemma 1) and $k = v_r + 1$ otherwise.
- FChild($[v_l..v_r]$) returns $\perp$ if $v_l = v_r$; otherwise it returns $[v_l..k-1]$, where $k = \texttt{RMQ}(v_l+1, v_r)$ is calculated according to Lemma 2.
- NSibling($[v_l..v_r]$) first determines $\texttt{NSV}(v_l)$. If $\texttt{NSV}(v_l) = v_r+1$, then $\mathsf{LCP}[v_l] > \mathsf{LCP}[v_r+1]$ by Lemma 1. In this case $\mathsf{LCP}[v_l]$ is the last LCP-index of the parent interval; so it returns $\perp$ because $[v_l..v_r]$ has no sibling. Otherwise we know from $\mathsf{LCP}[v_l] \leq \mathsf{LCP}[v_r+1]$ that $i = v_r+1$ is an LCP-index of the parent interval. There is a succeeding LCP-index j iff $)_i$ is preceded by $)^0_j$. If so, it returns $[v_r+1..j-1]$, otherwise it returns $[v_r+1..\texttt{NSV}(v_r+1)-1]$.

Moreover, the enhanced BPR provides a new (practical) approach to compress the LCP-array: Because we can access the first LCP-index of an LCP-interval $[i..j]$ in constant time (starting from $[i..j]$ or an arbitrary LCP-index) and all LCP-indices have the same LCP-value ℓ, it suffices to store ℓ solely at the first LCP-index. Combining this with the byte-aligned LCP-array [13] yields LCP-arrays of size $4n$ to $6n$ bits on texts from `pizzachili.dcc.uchile.cl` with $t_{\mathsf{LCP}} = O(1)$.

References

1. Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. J. Algorithms 39(2), 205–222 (2001)
2. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)

3. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
4. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Fully-compressed suffix trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
5. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comput. Sci. 410(51), 5354–5364 (2009)
6. Ohlebusch, E., Gog, S.: A compressed enhanced suffix array supporting fast string matching. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 51–62. Springer, Heidelberg (2009)
7. Fischer, J.: Wee LCP. Inform. Process. Lett. 110(8-9), 117–120 (2010)
8. Välimäki, N., Mäkinen, V., Gerlach, W., Dixit, K.: Engineering a compressed suffix tree implementation. ACM J. Experimental Algorithmics 4, Article no.2 (2009)
9. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 94–105. Springer, Heidelberg (2010)
10. Gog, S., Fischer, J.: Advantages of shared data structures for sequences of balanced parentheses. In: Proc. DCC, pp. 406–415. IEEE Press, Los Alamitos (2010)
11. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article No. 2 (2007)
12. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850. ACM/SIAM (2003)
13. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms 2(1), 53–86 (2004)
14. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
15. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
16. Sirén, J.: Sampled longest common prefix array. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 227–237. Springer, Heidelberg (2010)
17. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
18. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
19. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct representation of ordered trees. In: Proc. SODA, pp. 575–584. ACM/SIAM (2007)
20. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149. ACM/SIAM (2010)
21. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
22. Fischer, J., Heun, V.: Finding range minima in the middle: Approximations and applications. Mathematics in Computer Science 3(1), 17–30 (2010)
23. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. 38(6), 2162–2178 (2009)
24. Hon, W.K., Sadakane, K.: Space-economical algorithms for finding maximal unique matches. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 144–152. Springer, Heidelberg (2002)
25. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 451–462. Springer, Heidelberg (2009)