# Fully-Functional Succinct Trees

Kunihiko Sadakane* Gonzalo Navarro†

**Abstract**

We propose new succinct representations of ordinal trees, which have been studied extensively. It is known that any $n$-node static tree can be represented in $2n + o(n)$ bits and a large number of operations on the tree can be supported in constant time under the word-RAM model. However existing data structures are not satisfactory in both theory and practice because (1) the lower-order term is $\Omega(n \log\log n / \log n)$, which cannot be neglected in practice, (2) the hidden constant is also large, (3) the data structures are complicated and difficult to implement, and (4) the techniques do not extend to dynamic trees supporting insertions and deletions of nodes.

We propose a simple and flexible data structure, called the range min-max tree, that reduces the large number of relevant tree operations considered in the literature to a few primitives, which are carried out in constant time on sufficiently small trees. The result is then extended to trees of arbitrary size, achieving $2n + \mathcal{O}(n/\text{polylog}(n))$ bits of space. The redundancy is significantly lower than in any previous proposal, and the data structure is easily implemented. Furthermore, using the same framework, we derive the first fully-functional dynamic succinct trees.

## 1 Introduction

Trees are one of the most fundamental data structures, needless to say. A classical representation of a tree with $n$ nodes uses $\mathcal{O}(n)$ pointers or words. Because each pointer must distinguish all the nodes, it requires $\log n$ bits[1] in the worst case. Therefore the tree occupies $\Theta(n \log n)$ bits, which causes a space problem for manipulating large trees. Much research has been devoted to reducing the space to represent static trees [20, 26, 27, 29, 16, 17, 5, 10, 7, 8, 22, 19, 2, 18, 35, 21, 9] and dynamic trees [28, 34, 6, 1], achieving so-called *succinct data structures* for trees.

A succinct data structure stores objects using space close to the information-theoretic lower bound, while simultaneously supporting a number of primitive operations on the objects in constant time. The information-theoretic lower bound for storing an object from a universe with cardinality $L$ is $\log L$ bits because in the worst case this number of bits is necessary to distinguish any two objects. The size of the corresponding succinct data structure is typically $(1 + o(1)) \log L$ bits.

In this paper we are interested in *ordinal trees*, in which the children of a node are ordered. The information-theoretic lower bound to store an ordinal tree with $n$ nodes is $2n - \Theta(\log n)$ bits because there exist $\binom{2n-1}{n-1}/(2n-1) = 2^{2n}/\Theta(n^{\frac{3}{2}})$ such trees [26]. We assume that the computation model is the word RAM with word length $\Theta(\log n)$ in which arithmetic and logical operations on $\Theta(\log n)$-bit integers and $\Theta(\log n)$-bit memory accesses can be done in constant time. Under this model, there exist many succinct representations of ordinal trees achieving $2n + o(n)$ bits of space.

Basically there exist three types of such tree representations: the balanced parentheses sequence (BP) [20, 26], the level-order unary degree sequence (LOUDS) [20, 8], and the depth-first unary degree sequence (DFUDS) [5, 21]. An example of them is shown in Figure 1. LOUDS is a simple representation, but it lacks many basic operations, such as giving the subtree size of a tree node. Both BP and DFUDS build on a sequence of balanced parentheses, the former using the intuitive depth-first-search representation and the latter using a more sophisticated one. The advantage of DFUDS, when it was created, was that it supported a more complete set of operations in constant time, most notably going to the $i$-th child of a node. Later, this was also achieved using BP representation, yet requiring complicated additional data structures with nonnegligible lower-order terms in their space usage [22]. Another type of succinct ordinal trees, based on *tree covering* [17, 19, 9], has also achieved constant-time support of known operations, yet inheriting the problem of nonnegligible lower-order terms in size.

### 1.1 Our contributions

We focus on the BP representation, and achieve constant time for a large set of operations[2]. What distinguishes our proposal is its simplicity, which allows

*National Institute of Informatics (NII), 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan. sada@nii.ac.jp. Work supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

†Department of Computer Science, University of Chile. gnavarro@dcc.uchile.cl. Funded in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

[1] The base of logarithm is 2 throughout this paper.

[2] Moreover, as we manipulate a sequence of balanced parentheses, our data structure can be used to implement a DFUDS representation as well.

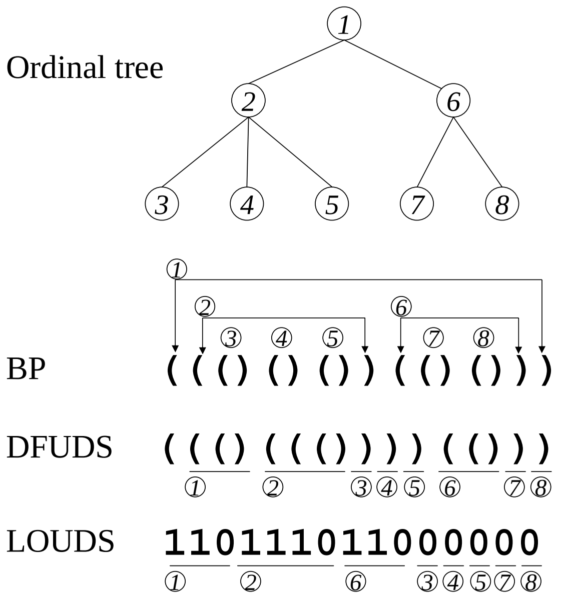


Figure 1: Succinct representations of trees.

easy implementation and derivation of dynamic variants with the same functionality; and its economy in the sublinear-space structures, which results in a considerably smaller lower-order term in the space usage.

For the static case, we obtain the following result.

Theorem 1.1. *For any ordinal tree with n nodes, all operations in Table 1 except insert and delete are carried out in constant time $\mathcal{O}(c^2)$ with a data structure using $2n + \mathcal{O}(n/\log^c n)$ bits of space on a $\Theta(\log n)$-bit word RAM, for any constant $c > 0$. The data structure can be constructed from the balanced parentheses sequence of the tree, in $\mathcal{O}(n)$ time using $\mathcal{O}(n)$ bits.*

Our data structure improves upon the lower-order term in the space complexity of previous representations. For example, formerly the extra data structure for *level-ancestor* has required $\mathcal{O}(n \log\log n/\sqrt{\log n})$ bits [29], or $\mathcal{O}(n(\log\log n)^2/\log n)$ bits[3] [21], and that for *child* has required $\mathcal{O}(n/(\log\log n)^2)$ bits [22]. The previous representation with maximum functionality [9] supports all the operations in Table 1, except *insert* and *delete*, in constant time using $2n + \mathcal{O}(n \log\log\log n/\log\log n)$-bit space. Ours requires $\mathcal{O}(n/\log^c n)$ bits for all the operations.

For the dynamic case, the following theorem summarizes our results.

[3] This data structure is for DFUDS, but the same technique can be also applied to BP.

Theorem 1.2. *On a $\Theta(\log n)$-bit word RAM, all operations on a dynamic ordinal tree with n nodes can be carried out within the worst-case complexities given in Table 1, using a data structure that requires $2n + \mathcal{O}(n \log\log n/\log n)$ bits. Alternatively, they can be carried out in $\mathcal{O}(\log n)$ time using $2n + \mathcal{O}(n/\log n)$ bits of space.*

There exist no previous dynamic data structures supporting all the operations in Table 1. The data structure of Raman and Rao [34] supports, for binary trees, *parent*, left and right *child*, and *subtree-size* of the current node in the course of traversing the tree in constant time, and updates in $\mathcal{O}((\log\log n)^{1+\epsilon})$ time. Note that this data structure assumes that all traversals start from the root. Chan et al. [6] gave a dynamic data structure using $\mathcal{O}(n)$ bits and supporting *findclose* and *enclose*, and updates, in $\mathcal{O}(\log n/\log\log n)$ time. They show this time is indeed optimal, by reduction from dynamic *rank*/*select* on bitmaps and given the lower bound of Fredman and Saks [14]. They also gave another data structure using $\mathcal{O}(n)$ bits and supporting *findclose*, *enclose*, *lca*, *leaf-rank*, *leaf-select*, and updates, in $\mathcal{O}(\log n)$ time.

The simplicity and space-efficiency of our data structures stem from the fact that any query operation in Table 1 is reduced to a few basic operations on a bit vector, which can be efficiently solved by a *range min-max tree.* This approach is different from previous studies in which each operation needs distinct auxiliary data structures. Therefore their total space is the summation over all the data structures, which enlarges the hidden constant in the lower-order term of the size. For example, the first succinct representation of BP [26] supported only *findclose*, *findopen*, and *enclose* (and other easy operations) and each operation used different data structures. Later, many further operations such as *lmost-leaf* [27], *lca* [35], *degree* [7], *child* and *child-rank* [22], and *level-ancestor* [29], were added to this representation by using other types of data structures for each. There exists another elegant data structure for BP supporting *findclose*, *findopen*, and *enclose* [16]. This reduces the size of the data structure for these basic operations, but still has to add extra auxiliary data structures for other operations.

Former static approaches use two-level data structures to reduce the size, which causes difficulties in dynamic case. Our approach using the range min-max tree, instead, is easily translated to the dynamic setting, resulting in simple and efficient dynamic data structures that support all of the operations in Table 1.

## 1.2 Organization of the paper

In Section 2 we review basic data structures used in

Table 1: Operations supported by our data structure. The time complexities are for the dynamic case; in the static case all operations are performed in constant time. The first group is composed of basic operations, used to implement the others, yet they could have other uses.

| operation | description | time complexity |
|---|---|---|
| $\mathit{inspect}(i)$ | $P[i]$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{findclose}(i)/\mathit{findopen}(i)$ | position of parenthesis matching $P[i]$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{enclose}(i)$ | position of tightest open parent. enclosing node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{rank}_((i)/\mathit{rank}_)(i)$ | number of open/close parentheses in $P[1,i]$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{select}_((i)/\mathit{select}_)(i)$ | position of $i$-th open/close parenthesis | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{rmqi}(i,j)/\mathit{RMQi}(i,j)$ | position of min/max excess value in range $[i,j]$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{pre\text{-}rank}(i)/\mathit{post\text{-}rank}(i)$ | preorder/postorder rank of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{pre\text{-}select}(i)/\mathit{post\text{-}select}(i)$ | the node with preorder/postorder $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{isleaf}(i)$ | whether $P[i]$ is a leaf | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{isancestor}(i,j)$ | whether $i$ is an ancestor of $j$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{depth}(i)$ | depth of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{parent}(i)$ | parent of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{first\text{-}child}(i)/\mathit{last\text{-}child}(i)$ | first/last child of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{next\text{-}sibling}(i)/\mathit{prev\text{-}sibling}(i)$ | next/previous sibling of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{subtree\text{-}size}(i)$ | number of nodes in the subtree of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{level\text{-}ancestor}(i,d)$ | ancestor $j$ of $i$ such that $\mathit{depth}(j)=\mathit{depth}(i)-d$ | $\mathcal{O}(\log n)$ or $\mathcal{O}(d+\log n/\log\log n)$ |
| $\mathit{level\text{-}next}(i)/\mathit{level\text{-}prev}(i)$ | next/previous node of $i$ in BFS order | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{level\text{-}lmost}(d)/\mathit{level\text{-}rmost}(d)$ | leftmost/rightmost node with depth $d$ | $\mathcal{O}(\log n)$ or $\mathcal{O}(d+\log n/\log\log n)$ |
| $\mathit{lca}(i,j)$ | the lowest common ancestor of two nodes $i,j$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{deepest\text{-}node}(i)$ | the (first) deepest node in the subtree of $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{degree}(i)$ | number of children of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{child}(i,q)$ | $q$-th child of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{child\text{-}rank}(i)$ | number of siblings to the left of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{in\text{-}rank}(i)$ | inorder of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{in\text{-}select}(i)$ | node with inorder $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{leaf\text{-}rank}(i)$ | number of leaves to the left of leaf $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{leaf\text{-}select}(i)$ | $i$-th leaf | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{lmost\text{-}leaf}(i)/\mathit{rmost\text{-}leaf}(i)$ | leftmost/rightmost leaf of node $i$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{insert}(i,j)$ | insert node given by matching parent. at $i$ and $j$ | $\mathcal{O}(\log n/\log\log n)$ |
| $\mathit{delete}(i)$ | delete node $i$ | $\mathcal{O}(\log n/\log\log n)$ |

this paper. In Section 3 we describe the main ideas for our new data structures for ordinal trees. Sections 4 and 5 describe the static construction. In Sections 6 and 7 we give two data structures for dynamic ordinal trees. In Section 8 we conclude and give future work directions.

## 2 Preliminaries

Here we describe the balanced parentheses sequence and basic data structures used in this paper.

### 2.1 Succinct data structures for *rank*/*select*

Consider a bit string $S[0,n-1]$ of length $n$. We define *rank* and *select* for $S$ as follows: $\mathit{rank}_c(S,i)$ is the number of occurrences $c\in\{0,1\}$ in $S[0,i]$, and $\mathit{select}_c(S,i)$ is the position of the $i$-th occurrence of $c$ in $S$. Note that $\mathit{rank}_c(S,\mathit{select}_c(S,i))=i$ and $\mathit{select}_c(S,\mathit{rank}_c(S,i))\le i$.

There exist many succinct data structures for *rank*/*select* [20, 25, 33]. A basic one uses $n+o(n)$ bits and supports *rank*/*select* in constant time on the word RAM with word length $\mathcal{O}(\log n)$. The space can be reduced if the number of 1's is small. For a string with $m$ 1's, there exists a data structure for constant-time *rank*/*select* using $\log\binom{n}{m}+\mathcal{O}(n\log\log n/\log n)=m\log\frac{n}{m}+\mathcal{O}(m+n\log\log n/\log n)$ bits [33]. Recently [30] the extra space has been reduced to $m\log\frac{n}{m}+\mathcal{O}(n\,t^t/\log^t n+n^{3/4})$ bits, performing *rank* and *select* in $\mathcal{O}(t)$ time. This can be built in linear worst-case time[4].

[4]They use a predecessor structure by Pătraşcu and Thorup [31], more precisely their result achieving time "$\lg\frac{\ell-\lg n}{a}$", which is a simple modification of van Emde Boas' data structure.

A crucial technique for succinct data structures is *table lookup*. For small-size problems we construct a table which stores answers for all possible sequences and queries. For example, for *rank* and *select*, we use a table storing all answers for all 0,1 patterns of length $\frac{1}{2}\log n$. Because there exist only $2^{\frac{1}{2}\log n} = \sqrt{n}$ different patterns, we can store all answers in a universal table (i.e., not depending on the bit sequence) that uses $\sqrt{n} \cdot \mathrm{polylog}(n) = o(n/\mathrm{polylog}(n))$ bits, which can be accessed in constant time on a word RAM with word length $\Theta(\log n)$.

### 2.2 Succinct tree representations

We will focus on the BP representation of trees. A rooted ordered tree $T$, or ordinal tree, with $n$ nodes is represented by a string $P[0, 2n-1]$ of balanced parentheses of length $2n$. A node $v \in T$ is represented by a pair of matching parentheses $(\ldots)$ and all subtrees rooted at the node are encoded in order between the matching parentheses (see Figure 1 for an example). Moreover, node $v$ is identified with the position $i$ of the open parenthesis $P[i]$ representing the node.

In the static setting, the tree does not change. In the dynamic setting, we consider insertion and deletion of internal nodes or leaves. More precisely, we accept inserting a new pair of matching parentheses at any legal position of $P$, as well as deleting any existing pair of matching parentheses.

## 3 Fundamental concepts

In this section we give the basic ideas of our ordinal tree representation. In the next sections we build on these to define our static and dynamic representations.

We represent a possibly non-balanced[5] parentheses sequence by a 0,1 vector $P[0, n-1]$ ($P[i] \in \{0,1\}$). Each opening/closing parenthesis is encoded by $( = 1,\ ) = 0$.

First, we remind the reader that several operations of Table 1 either are trivial in a BP representation, or are easily solved using *enclose*, *findclose*, *findopen*, *rank*, and *select* [26]. These are:

$$
\begin{aligned}
\mathit{inspect}(i) &= \mathit{rank}_1(P,i) - \mathit{rank}_1(P,i-1) \\
&\phantom{=} \text{(if accessing } P[i] \text{ is problematic)} \\
\mathit{isleaf}(i) &= [\mathit{inspect}(i+1) = 0] \\
\mathit{isancestor}(i,j) &= i \le j \text{ and} \\
&\phantom{=} \mathit{findclose}(P,j) \le \mathit{findclose}(P,i) \\
\mathit{depth}(i) &= \mathit{rank}_1(P,i) - \mathit{rank}_0(P,i) \\
\mathit{parent}(i) &= \mathit{enclose}(P,i) \\
\mathit{pre\text{-}rank}(i) &= \mathit{rank}_1(P,i) \\
\mathit{pre\text{-}select}(i) &= \mathit{select}_1(P,i) \\
\mathit{post\text{-}rank}(i) &= \mathit{rank}_0(P,i) \\
\mathit{post\text{-}select}(i) &= \mathit{select}_0(P,i) \\
\mathit{first\text{-}child}(i) &= i+1 \text{ (if } i \text{ is not a leaf)} \\
\mathit{last\text{-}child}(i) &= \mathit{findopen}(P, \mathit{findclose}(P,i)-1) \\
&\phantom{=} \text{(if } i \text{ is not a leaf)} \\
\mathit{next\text{-}sibling}(i) &= \mathit{findclose}(i)+1 \\
&\phantom{=} \text{(if } P[\mathit{findclose}(i)+1] = 0, \\
&\phantom{=} \text{ then } i \text{ is the last sibling)} \\
\mathit{prev\text{-}sibling}(i) &= \mathit{findopen}(i-1) \text{ (if } P[i-1] = 1 \\
&\phantom{=} \text{ then } i \text{ is the first sibling)} \\
\mathit{subtree\text{-}size}(i) &= (\mathit{findclose}(i) - i + 1)/2
\end{aligned}
$$

Hence the above operations will not be considered further in the paper. Let us now focus on a small set of primitives needed to implement most of the other operations. For any function $g(\cdot)$ on $\{0,1\}$, we define the following.

Definition 1. *For a 0,1 vector $P[0, n-1]$ and a function $g(\cdot)$ on $\{0,1\}$,*

$$
\begin{aligned}
\mathit{sum}(P,g,i,j) &\stackrel{\text{def}}{=} \sum_{k=i}^{j} g(P[k]) \\
\mathit{fwd\text{-}search}(P,g,i,d) &\stackrel{\text{def}}{=} \min_{j \ge i}\{j \mid \mathit{sum}(P,g,i,j) = d\} \\
\mathit{bwd\text{-}search}(P,g,i,d) &\stackrel{\text{def}}{=} \max_{j \le i}\{j \mid \mathit{sum}(P,g,j,i) = d\} \\
\mathit{rmq}(P,g,i,j) &\stackrel{\text{def}}{=} \min_{i \le k \le j}\{\mathit{sum}(P,g,i,k)\} \\
\mathit{rmqi}(P,g,i,j) &\stackrel{\text{def}}{=} \operatorname*{argmin}_{i \le k \le j}\{\mathit{sum}(P,g,i,k)\} \\
\mathit{RMQ}(P,g,i,j) &\stackrel{\text{def}}{=} \max_{i \le k \le j}\{\mathit{sum}(P,g,i,k)\} \\
\mathit{RMQi}(P,g,i,j) &\stackrel{\text{def}}{=} \operatorname*{argmax}_{i \le k \le j}\{\mathit{sum}(P,g,i,k)\}
\end{aligned}
$$

The following function is particularly important.

Definition 2. *Let $\pi$ be the function such that $\pi(1) = 1, \pi(0) = -1$. Given $P[0, n-1]$, we define the* excess array *$E[0, n-1]$ of $P$ as an integer array such that $E[i] = \mathit{sum}(P, \pi, 0, i)$.*

Note that $E[i]$ stores the difference between the number of opening and closing parentheses in $P[0, i]$. When $P[i]$ is an opening parenthesis, $E[i] = \mathit{depth}(i)$ is the depth of the corresponding node, and is the depth minus 1 for closing parentheses. We will use $E$ as a conceptual device in our discussions, it will not be

[5] Later we will use these constructions to represent arbitrary chunks of a balanced sequence.

stored. Note that, given the form of $\pi$, it holds that $|E[i+1]-E[i]|=1$ for all $i$.

The above operations are sufficient to implement the basic navigation on parentheses, as the next lemma shows. Note that the equation for *findclose* is well known, and the one for *level-ancestor* has appeared as well [29], but we give proofs for completeness.

LEMMA 3.1. *Let $P$ be a BP sequence encoded by $\{0,1\}$. Then findclose, findopen, enclose, and level-ancestor can be expressed as follows.*

$$\begin{aligned} \mathit{findclose}(i) &= \mathit{fwd\text{-}search}(P,\pi,i,0) \\ \mathit{findopen}(i) &= \mathit{bwd\text{-}search}(P,\pi,i,0) \\ \mathit{enclose}(i) &= \mathit{bwd\text{-}search}(P,\pi,i,2) \\ \mathit{level\text{-}ancestor}(i,d) &= \mathit{bwd\text{-}search}(P,\pi,i,d+1) \end{aligned}$$

*Proof.* For *findclose*, let $j>i$ be the position of the closing parenthesis matching the opening parenthesis at $P[i]$. Then $j$ is the smallest index $>i$ such that $E[j]=E[i]-1=E[i-1]$ (because of the node depths). Since by definition $E[k]=E[i-1]+\mathit{sum}(P,\pi,i,k)$ for any $k>i$, $j$ is the smallest index $>i$ such that $\mathit{sum}(P,\pi,i,j)=0$. This is, by definition, $\mathit{fwd\text{-}search}(P,\pi,i,0)$.

For *findopen*, let $j<i$ be the position of the opening parenthesis matching the closing parenthesis at $P[i]$. Then $j$ is the largest index $<i$ such that $E[j-1]=E[i]$ (again, because of the node depths)[6]. Since by definition $E[k-1]=E[i]-\mathit{sum}(P,\pi,k,i)$ for any $k<i$, $j$ is the largest index $<i$ such that $\mathit{sum}(P,\pi,j,i)=0$. This is $\mathit{bwd\text{-}search}(P,\pi,i,0)$.

For *enclose*, let $j<i$ be the position of the opening parenthesis that most tightly encloses the opening parenthesis at $P[i]$. Then $j$ is the largest index $<i$ such that $E[j-1]=E[i]-2$ (note that now $P[i]$ is an opening parenthesis). Now we reason as for *findopen* to get $\mathit{sum}(P,\pi,j,i)=2$.

Finally, the proof for *level-ancestor* is similar to that for *enclose*. Now $j$ is the largest index $<i$ such that $E[j-1]=E[i]-d-1$, which is equivalent to $\mathit{sum}(P,\pi,j,i)=d+1$. $\square$

We also have the following, easy or well-known, equalities:

$$\begin{aligned} \mathit{lca}(i,j) &= \begin{cases} i & (\text{if } \mathit{isancestor}(i,j)) \\ j & (\text{if } \mathit{isancestor}(j,i)) \\ \mathit{parent}(\mathit{rmqi}(P,\pi,i,j)+1) \end{cases} \\ \mathit{deepest\text{-}node}(i) &= \mathit{RMQi}(P,\pi,i,\mathit{findclose}(i)) \\ \mathit{level\text{-}next}(i) &= \mathit{fwd\text{-}search}(P,\pi,\mathit{findclose}(i),0) \\ \mathit{level\text{-}prev}(i) &= \mathit{findopen}(\mathit{bwd\text{-}search}(P,\pi,i,0)) \\ \mathit{level\text{-}lmost}(d) &= \mathit{fwd\text{-}search}(P,\pi,0,d) \\ \mathit{level\text{-}rmost}(d) &= \mathit{findopen}(\mathit{bwd\text{-}search}(P,\pi,n-1,-d)) \end{aligned}$$

[6]Note $E[j]-1=E[i]$ could hold at incorrect places, where $P[j]$ is a closing parenthesis.

To compute *degree*, *child*, and *child-rank*, the following lemma is important.

LEMMA 3.2. *The number of children of node $i$ is equal to the number of occurrences of the minimum value in $E[i+1,\mathit{findclose}(i)-1]$.*

*Proof.* Let $d=E[i]=\mathit{depth}(i)$ and $j=\mathit{findclose}(i)$. Then $E[j]=d-1$ and all excess values in $E[i+1,j-1]$ are $\geq d$. Therefore, as $|E[r+1]-E[r]|=1$ for all $r$, the minimum value in $E[i+1,j-1]$ is $d$. Moreover, for the range $[i_k,j_k]$ corresponding to the $k$-th child of $i$, $E[i_k]=d+1$, $E[j_k]=d$, and all the values between them are $>d$. Therefore the number of occurrences of $d$, which is the minimum value in $E[i+1,j-1]$, is equal to the number of children of $i$. $\square$

We also show that the above functions unify the algorithms for computing *rank*/*select* on 0,1 vectors and those for balanced parenthesis sequences. Namely, let $\phi,\psi$ be functions such that $\phi(0)=0,\phi(1)=1,\psi(0)=1,\psi(1)=0$. Then the following equalities hold.

LEMMA 3.3. *For a 0,1 vector $P$,*

$$\begin{aligned} \mathit{rank}_1(P,i) &= \mathit{sum}(P,\phi,0,i) \\ \mathit{select}_1(P,i) &= \mathit{fwd\text{-}search}(P,\phi,0,i) \\ \mathit{rank}_0(P,i) &= \mathit{sum}(P,\psi,0,i) \\ \mathit{select}_0(P,i) &= \mathit{fwd\text{-}search}(P,\psi,0,i) \end{aligned}$$

Therefore, in principle we must focus only on the following set of primitives: *fwd-search*, *bwd-search*, *sum*, *rmqi*, *RMQi*, *degree*, *child*, and *child-rank*. The few remaining operations will be handled later.

Our data structure for queries on a 0,1 vector $P$ is basically a search tree in which each leaf corresponds to a range of $P$, and each node stores the last, maximum, and minimum values, within its subtree, of prefix sums of $P$.

DEFINITION 3. *A* range min-max tree *for a vector $P[0,n-1]$ and a function $g(\cdot)$ is defined as follows. Let $[\ell_1..r_1],[\ell_2..r_2],\ldots,[\ell_q..r_q]$ be a partition of $[0..n-1]$ where $\ell_1=0,r_i+1=\ell_{i+1},r_q=n-1$. Then the $i$-th leftmost leaf of the tree stores the sub-vector $P[\ell_i,r_i]$, as well as $e[i]=\mathit{sum}(P,g,0,r_i)$, $m[i]=e[i-1]+\mathit{rmq}(P,g,\ell_i,r_i)$ and $M[i]=e[i-1]+\mathit{RMQ}(P,g,\ell_i,r_i)$. Each internal node $u$ stores in $e[u]/m[u]/M[u]$ the last/minimum/maximum of the $e/m/M$ values stored*

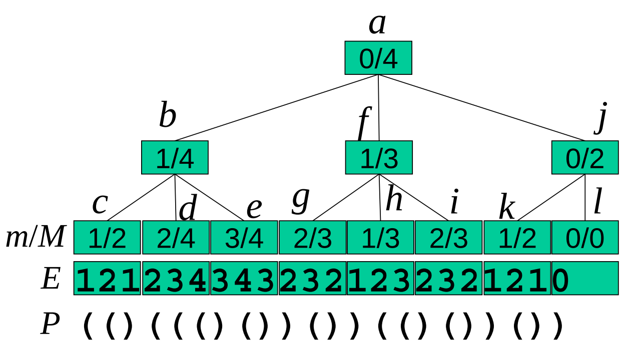


Figure 2: An example of the range min-max tree using function $\pi$, and showing the $m/M$ values.

*in its child nodes. Thus, the root node stores* $e = \mathit{sum}(P, g, 0, n-1)$, $m = \mathit{rmq}(P, g, 0, n-1)$ *and* $M = \mathit{RMQ}(P, g, 0, n-1)$.

EXAMPLE 1. *An example of range min-max tree is shown in Figure 2. Here we use* $g = \pi$, *and thus the nodes store the minimum/maximum values of array* $E$ *in the corresponding interval.*

## 4 A simple data structure for moderate-size trees

Building on the previous ideas, we give a simple data structure to compute *fwd-search*, *bwd-search*, and *sum* in constant time for arrays of moderate size. Then we will consider further operations.

Let $g(\cdot)$ be a function on $\{0, 1\}$ taking values in $\{1, 0, -1\}$. We call such a function $\pm 1$ *function*. Note that there exist only six such functions where $g(0) \neq g(1)$, which are indeed $\phi, -\phi, \psi, -\psi, \pi, -\pi$.

Let $w$ be the bit length of the machine word in the RAM model, and $c \geq 1$ any constant. We have a (not necessarily balanced) parentheses vector $P[0, n-1]$, of moderate size $n \leq N = w^c$. Assume we wish to solve the operations for an arbitrary $\pm 1$ function $g(\cdot)$, and let $G[i]$ denote $\mathit{sum}(P, g, 0, i)$, analogously to $E[i]$ for $g = \pi$.

Our data structure is a range min-max tree $T_{mM}$ for vector $P$ and function $g(\cdot)$. Let $s = \frac{1}{2}w$. We imaginarily divide vector $P$ into $\lceil n/s \rceil$ blocks of length $s$. These form the partition alluded in Definition 3: $\ell_i = s \cdot (i-1)$. Thus the values $m[i]$ and $M[i]$ correspond to minima and maxima of $G$ within each block, and $e[i] = G[r_i]$.

Furthermore, the tree will be $k$-ary and complete, for $k = \Theta(w/\log w)$. Because it is complete, the tree can be represented just by three integer arrays $e'[0, \mathcal{O}(n/s)]$, $m'[0, \mathcal{O}(n/s)]$, and $M'[0, \mathcal{O}(n/s)]$, like a heap.

Because $-w^c \leq e'[i], m'[i], M'[i] \leq w^c$ for any $i$, arrays $e'$, $m'$ and $M'$ occupy at most (i.e., for $k = 2$) $2\frac{n}{s} \cdot \lceil \log(2w^c + 1) \rceil = \mathcal{O}(nc \log w/w)$ bits each. The depth of the tree is $\lceil \log_k(n/s) \rceil = \mathcal{O}(c)$.

The following fact is well known; we reprove it for completeness.

LEMMA 4.1. *Any range* $[i..j] \subseteq [0..n-1]$ *in* $T_{mM}$ *can be represented by a disjoint union of* $\mathcal{O}(ck)$ *subranges where the leftmost and rightmost ones may be subranges of leaves of* $T_{mM}$, *and the others correspond to whole tree nodes.*

*Proof.* Let $a$ be the smallest value such that $i \leq r_a$ and $b$ be the largest such that $j \geq \ell_b$. Then the range $[i..j]$ is covered by the partition $[i..j] = [i..r_a][\ell_{a+1}..r_{a+1}] \ldots [\ell_b..j]$ (we can discard the special case $a = b$, as in this case we have already one leaf covering $[i..j]$). Then $[i..r_a]$ and $[\ell_b..j]$ are the leftmost and rightmost leaf subranges alluded in the lemma; all the others are whole tree nodes.

It remains to show that we can reexpress this partition using $\mathcal{O}(ck)$ tree nodes. If all the $k$ children of a node are in the range, we replace the $k$ children by the parent node, and continue recursively level by level. Note that if two parent nodes are created in a given level, then all the other intermediate nodes of the same level must be created as well, because the original/created nodes form a range at any level. At the end, there cannot be more than $2k-2$ nodes at any level, because otherwise $k$ of them would share a single parent and would have been replaced. As there are $c$ levels, the obtained set of nodes covering $[i..j]$ is of size $\mathcal{O}(ck)$. □

EXAMPLE 2. *In Figure 2 (where* $s = k = 3$*), the range* $[3..18]$ *is covered by* $[3..5], [6..8], [9..17], [18..18]$. *They correspond to nodes* $d$, $e$, $f$, *and a part of leaf* $k$, *respectively.*

Computing $\mathit{fwd\text{-}search}(P, g, i, d)$ is done as follows ($\mathit{bwd\text{-}search}$ is symmetric). First we check if the block of $i$, $[\ell_k, r_k]$ for $k = \lfloor i/s \rfloor$, contains $\mathit{fwd\text{-}search}(P, g, i, d)$ with table lookup using vector $P$, by precomputing a simple universal table of $2^s \cdot 2s^2 \cdot \log s = \mathcal{O}(\sqrt{2^w} w^2 \log w)$ bits. If so, we are done. Else, we compute the global target value we seek, $d' = G[i-1] + d = e[k] - \mathit{sum}(P, g, i, r_k) + d$ (again, the sum inside the block is done in constant time using table lookup). Now we divide the range $[r_k + 1, n-1]$ into subranges $I_1, I_2, \ldots$ represented by range min-max tree nodes $u_1, u_2, \ldots$ as in Lemma 4.1. Then, for each $I_j$, we check if the target value $d'$ is between $m[u_j]$ and $M[u_j]$, the minimum and maximum values of subrange $I_j$. Let $I_k$ be the first $j$ such that $m[u_j] \leq d' \leq M[u_j]$, then $\mathit{fwd\text{-}search}(P, g, i, d)$ lies within $I_k$. If $I_k$ corresponds to an internal tree node, we iteratively find the leftmost child of the node whose range contains $d'$, until we reach a leaf. Finally, we find the target in the block corresponding to the leaf by table lookup, using $P$ again.

EXAMPLE 3. *In Figure 2, where $G = E$ and $g = \pi$, computing findclose$(3) =$ fwd-search$(P, \pi, 3, 0) = 12$ can be done as follows. Note this is equivalent to finding the first $j > 3$ such that $E[i] = E[3-1] + 0 = 1$. First examine the node $\lfloor 3/s \rfloor = 1$ (labeled $d$ in the figure). We see that the target 1 does not exist within $d$ after position 3. Next we examine node $e$. Since $m[e] = 3$ and $M[e] = 4$, $e$ does not contain the answer either. Next we examine the node $f$. Because $m[f] = 1$ and $M[f] = 3$, the answer must exist in its subtree. Therefore we scan the children of $f$ from left to right, and find the leftmost one with $m[\cdot] \le 1$, which is node $h$. Because node $h$ is already a leaf, we scan the segment corresponding to it, and find the answer 12.*

The sequence of subranges arising in this search corresponds to a leaf-to-leaf path in the range min-max tree, and it contains $\mathcal{O}(ck)$ ranges according to Lemma 4.1. We show now how to carry out this search in time $\mathcal{O}(c^2)$ rather than $\mathcal{O}(ck)$.

According to Lemma 4.1, the $\mathcal{O}(ck)$ nodes can be partitioned into $\mathcal{O}(c)$ sequences of sibling nodes. We will manage to carry out the search within each such sequence in $\mathcal{O}(c)$ time. Assume we have to find the first $j \ge i$ such that $m[u_j] \le d' \le M[u_j]$, where $u_1, u_2, \ldots, u_k$ are sibling nodes in $T_{mM}$. We first check if $m[u_i] \le d' \le M[u_i]$. If so, the answer is $u_i$. Otherwise, if $d' < m[u_i]$, the answer is the first $j > i$ such that $m[u_j] \le d'$, and if $d' > M[u_i]$, the answer is the first $j > i$ such that $M[u_j] \ge d'$.

LEMMA 4.2. *Let $u_1, u_2, \ldots$ a sequence of $T_{mM}$ nodes containing consecutive intervals of $P$. If $g(\cdot)$ is a $\pm 1$ function and $d < m[u_1]$, then the first $j$ such that $d \in [m[u_j], M[u_j]]$ is the first $j > 1$ such that $d \ge m[u_j]$. Similarly, if $d > M[u_1]$, then it is the first $j > 1$ such that $d \le M[u_j]$.*

*Proof.* Since $g(\cdot)$ is a $\pm 1$ function and the intervals are consecutive, $M[u_j] \ge m[u_{j-1}] - 1$ and $m[u_j] \le M[u_{j-1}] + 1$. Therefore, if $d \ge m[u_j]$ and $d < m[u_{j-1}]$, then $d < M[u_j] + 1$, thus $d \in [m[u_j], M[u_j]]$; and of course $d \notin [m[u_k], M[u_k]]$ for any $k < j$ as $j$ is the first index such that $d \ge m[u_j]$. The other case is symmetric. □

Thus the problem is reduced to finding the first $j > i$ such that $m[j] \le d'$, among (at most) $k$ sibling nodes (the case $M[j] \ge d'$ is symmetric). We build a universal table with all the possible sequences of $k/c$ $m[\cdot]$ values and all possible $-w^c \le d' \le w^c$ values, and for each such sequence and $d'$ we store the first $j$ in the sequence such that $m[j] \le d'$ (or we store a mark telling that there is no such node in the sequence). Thus the table has $(2w^c + 1)^{(k/c)+1}$ entries, and $\log(1 + k/c)$ bits per entry. By choosing the constant of $k = \Theta(w/\log w)$ so that $k \le \frac{cw}{2\log(2w+1)} - c$, the total space is $\mathcal{O}(\sqrt{2^w}\log w)$ (and the arguments for the table fit in a machine word). With the table, each search for the first node in a sequence of siblings can be done by chunks of $k/c$ nodes, which takes $\mathcal{O}(k/(k/c)) = \mathcal{O}(c)$ rather than $\mathcal{O}(k)$ time, and hence the overall time is $\mathcal{O}(c^2)$ rather than $\mathcal{O}(ck)$. Note that $k/c$ values to input to the table are stored in contiguous memory, as we store the $m'[\cdot]$ values in heap order. Thus we can access any $k/c$ consecutive children values in constant time. We use an analogous table for $M[\cdot]$.

Finally, the process to solve *sum*$(P, g, i, j)$ in $\mathcal{O}(c^2)$ time is simple. We descend in the tree up to the leaf $[\ell_k, r_k]$ containing $j$. In the process we easily obtain *sum*$(P, g, 0, \ell_k - 1)$, and compute the rest, *sum*$(P, g, \ell_k, j)$, in constant time using a universal table we have already introduced. We repeat the process for *sum*$(P, g, 0, i-1)$ and then subtract both results.

We have proved the following lemma.

LEMMA 4.3. *In the RAM model with $w$-bit word size, for any constant $c \ge 1$ and a 0,1 vector $P$ of length $n < w^c$, and a $\pm 1$ function $g(\cdot)$, fwd-search$(P, g, i, j)$, bwd-search$(P, g, i, j)$, and sum$(P, g, i, j)$ can be computed in $\mathcal{O}(c^2)$ time using the range min-max tree and universal lookup tables that require $\mathcal{O}(\sqrt{2^w} w^2 \log w)$ bits.*

### 4.1 Supporting range minimum queries

Next we consider how to compute *rmqi*$(P, g, i, j)$ and *RMQi*$(P, g, i, j)$.

LEMMA 4.4. *In the RAM model with $w$-bit word size, for any constant $c \ge 1$ and a 0,1 vector $P$ of length $n < w^c$, and a $\pm 1$ function $g(\cdot)$, rmqi$(P, g, i, j)$ and RMQi$(P, g, i, j)$ can be computed in $\mathcal{O}(c^2)$ time using the range min-max tree and universal lookup tables that require $\mathcal{O}(\sqrt{2^w} w^2 \log w)$ bits.*

*Proof.* Because the algorithm for *RMQi* is analogous to that for *rmqi*, we consider only the latter. From Lemma 4.1, the range $[i, j]$ is expressed by a disjoint partition of $\mathcal{O}(ck)$ subranges, each corresponding to some node of the range min-max tree. Let $\mu_1, \mu_2, \ldots$ be the minimum values of the subranges. Then the minimum value in $[i, j]$ is the minimum of them. The minimum values in each subrange are stored in array $m'$, except for at most two subranges corresponding to leaves of the range min-max tree. The minimum values of such leaf subranges are found by table lookups using $P$, by precomputing a universal table of $\mathcal{O}(\sqrt{2^w}w^2 \log w)$ bits. The minimum value of a subsequence $\mu_\ell, \ldots, \mu_r$ which shares the same parent in the range min-max tree can be also found by table lookups. There are at most $k$ such values, and for consecutive $k/c$ values we use a universal table to find their minimum, and repeat this $c$ times, as before. The size of the table is $\mathcal{O}(\sqrt{2^w}(k/c)\log(k/c)) = \mathcal{O}(\sqrt{2^w}w)$ bits (the $k/c$ factor is to account for queries that span less than $k/c$ blocks, so we can compute the minimum up to any value in the sequence).

Let $\mu$ be the minimum value we find in $\mu_1, \mu_2, \ldots, \mu_m$. If there is a tie, we choose the leftmost one. If $\mu$ corresponds to an internal node of the range min-max tree, we traverse the tree from the node to a leaf having the leftmost minimum value. At each step, we find the leftmost child of the current node having the minimum, in $\mathcal{O}(c)$ time using our precomputed table. We repeat the process from the resulting child, until reaching a leaf. Finally, we find the index of the minimum value in the leaf, in constant time by a lookup on our other universal table. The overall time complexity is $\mathcal{O}(c^2)$. □

### 4.2 Other operations

The previous development on *fwd-search*, *bwd-search*, *rmqi*, and *RMQi*, has been general, for any $g(\cdot)$. Applied to $g = \pi$, they solve a large number of operations, as shown in Section 3. For the remaining ones we focus directly on the case $g = \pi$.

It is obvious how to compute $degree(i)$, $child(i, q)$ and $child\text{-}rank(i)$ in time proportional to the degree of the node. To compute them in constant time, we use Lemma 3.2, that is, the number of children of node $i$ is equal to the number of minimum values in the excess array for $i$. We add another array $n'[\cdot]$ to the data structure. In the range min-max tree, each node stores the minimum value of its subrange. In addition to this, we also store in $n'[\cdot]$ the number of the minimum values of the subrange of each node in the tree.

Now we can compute $degree(i)$ in constant time. Let $d = depth(i)$ and $j = findclose(i)$. We partition the range $E[i+1, j-1]$ into $\mathcal{O}(ck)$ subranges, each of which corresponds to a node of the range min-max tree. Then for each subrange whose minimum value is $d$, we sum up the number of occurrences of the minimum value ($n'[\cdot]$). The number of occurrences of the minimum value in leaf subranges can be computed by table lookup on $P$, with a universal table using $\mathcal{O}(\sqrt{2^w}w^2 \log w)$ bits. The time complexity is $\mathcal{O}(c^2)$ if we use universal tables that let us process chunks of $k/c$ children at once, that is, the one used for *rmqi* plus another telling the number of times the minimum appears in the sequence. This table also requires $\mathcal{O}(\sqrt{2^w}w)$ bits.

Operation $child\text{-}rank(i)$ can be computed similarly, by counting the number of minima in $E[parent(i), i-1]$. Operation $child(i, q)$ follows the same idea of $degree(i)$, except that, in the node where the sum of $n'[\cdot]$ exceeds $q$, we must descend until the range min-max leaf that contains the opening parenthesis of the $q$-th child. This search is also guided by the $n'[\cdot]$ values of each node, and is done also in $\mathcal{O}(c^2)$ time by using another universal table of $O(\sqrt{2^w} \log w)$ bits (that tells us where the $n'[\cdot]$ exceed some threshold in a sequence of $k/c$ values).

For operations *leaf-rank*, *leaf-select*, *lmost-leaf* and *rmost-leaf*, we define a bit-vector $P_1[0, n-1]$ such that $P_1[i] = 1 \iff P[i] = 1 \wedge P[i+1] = 0$. Then $leaf\text{-}rank(i) = rank_1(P_1, i)$ and $leaf\text{-}select(i) = select_1(P_1, i)$ hold. The other operations are computed by $lmost\text{-}leaf(i) = select_1(P_1, rank_1(P_1, i-1)+1)$ and $rmost\text{-}leaf(i) = select_1(P_1, rank_1(P_1, findclose(i)))$.

We recall the definition of *inorder* of nodes, which is essential for compressed suffix trees.

Definition 4. ([35]) *The inorder rank of an internal node $v$ is defined as the number of visited internal nodes, including $v$, in the left-to-right depth-first traversal, when $v$ is visited from a child of it and another child of it will be visited next.*

Note that an internal node with $q$ children has $q-1$ inorders, so leaves and unary nodes have no inorder. We define $in\text{-}rank(i)$ as the smallest inorder value of internal node $i$.

To compute *in-rank* and *in-select*, we use another bit-vector $P_2[0, n-1]$ such that $P_2[i] = 1 \iff P[i] = 0 \wedge P[i+1] = 1$. The following lemma gives an algorithm to compute the inorder of an internal node.

Lemma 4.5. ([35]) *Let $i$ be an internal node, and let $j = in\text{-}rank(i)$, so $i = in\text{-}select(j)$. Then*

$$\begin{aligned} in\text{-}rank(i) &= rank_1(P_2, findclose(P, i+1)) \\ in\text{-}select(j) &= enclose(P, select_1(P_2, j)+1) \end{aligned}$$

*Note that $in\text{-}select(j)$ will return the same node $i$ for any its $degree(i) - 1$ inorder values.*

Note that we need not store $P_1$ and $P_2$ explicitly; they can be computed from $P$ when needed. We only need the extra data structures for constant-time *rank* and *select*, which can be reduced to the corresponding *sum* and *fwd-search* operations on the virtual $P_1$ and $P_2$ vectors.

### 4.3 Reducing extra space

Apart from vector $P[0, n-1]$, we need to store vectors $e'$, $m'$, $M'$, and $n'$. In addition, to implement *rank* and *select* using *sum* and *fwd-search*, we would need to store vectors $e'_\phi$, $e'_\psi$, $m'_\phi$, $m'_\psi$, $M'_\phi$, and $M'_\psi$ which maintain the corresponding values for functions $\phi$ and $\psi$. However, note that $sum(P, \phi, 0, i)$ and $sum(P, \psi, 0, i)$ are nondecreasing, thus the minimum/maximum within the block is just the value of the sum at the beginning/end of the block. Moreover, as $sum(P, \pi, 0, i) = sum(P, \phi, 0, i) - sum(P, \psi, 0, i)$ and $sum(P, \phi, 0, i) + sum(P, \psi, 0, i) = i$, it turns out that both $e_\phi[i] = (r_i + e[i])/2$ and $e_\psi[i] = (r_i - e[i])/2$ are redundant; analogous formulas hold for $M$ and $m$ and for internal nodes. Moreover, any sequence of $k/c$ consecutive such values can be obtained, via table lookup, from the sequence of $k/c$ consecutive values of $e[\cdot]$, because the $r_i$ values increase regularly at any node. Hence we do not store any extra information to support $\phi$ and $\psi$.

If we store vectors $e'$, $m'$, $M'$, and $n'$ naively, we require $\mathcal{O}(nc\log(w)/w)$ bits of extra space on top of the $n$ bits for $P$.

The space can be largely reduced by using a recent technique by Pătraşcu [30]. They define an *aB-tree* over an array $A[0, n-1]$, for $n$ a power of $B$, as a complete tree of arity $B$, storing $B$ consecutive elements of $A$ in each leaf. Additionally, a value $\varphi \in \Phi$ is stored at each node. This must be a function of the corresponding elements of $A$ for the leaves, and a function of the $\varphi$ values of the children, and of the subtree size, for internal nodes. The construction is able to decode the $B$ values of $\varphi$ for the children of any node in constant time, and to decode the $B$ values of $A$ for the leaves in constant time, if they can be packed in a machine word.

In our case, $A = P$ is the vector, $B = k = s$ is our arity, and our trees will be of size $N = B^c$, which is slightly smaller than the $w^c$ we have been assuming. Our values are tuples $\varphi \in \langle -B^c, -B^c, 0, -B^c\rangle \ldots \langle B^c, B^c, B^c, B^c\rangle$ encoding the $m$, $M$, $n$, and $e$ values at the nodes, respectively. We give next their result, adapted to our case.

LEMMA 4.6. (ADAPTED FROM THM. 8 IN [30]) *Let $\Phi = (2B+1)^{4c}$, and $B$ be such that $(B+1)\log(2B+1) \le \frac{w}{8c}$ (thus $B = \Theta(\frac{w}{c\log w})$). An aB-tree of size $N = B^c$ with values in $\Phi$ can be stored using $N+2$ bits, plus universal lookup tables of $\mathcal{O}(\sqrt{2^w})$ bits. It can obtain the $m$, $M$, $n$ or $e$ values of the children of any node, and descend to any of those children, in constant time. The structure can be built in $\mathcal{O}(N + w^{3/2})$ time, plus $\mathcal{O}(\sqrt{2^w}\text{poly}(w))$ for the universal tables.*

The "$+w^{3/2}$" construction time comes from a fusion tree [15] that is used internally on $\mathcal{O}(w)$ values. It could be reduced to $w^\epsilon$ time for any constant $\epsilon > 0$ and navigation time $\mathcal{O}(1/\epsilon)$, but we prefer to set $c > 3/2$ to make it irrelevant.

These parameters still allow us to represent our range min-max trees while yielding the complexities we had found, as $k = \Theta(w/\log w)$ and $N \le w^c$. Our accesses to the range min-max tree are either $(i)$ partitioning intervals $[i, j]$ into $\mathcal{O}(ck)$ subranges, which are easily identified by navigating from the root in $\mathcal{O}(c)$ time (as the $k$ children are obtained together in constant time); or $(ii)$ navigating from the root while looking for some leaf based on the intermediate $m$, $M$, $n$, or $e$ values.

Thus we retain all of our time complexities. The space, instead, is reduced to $N + 2 + \mathcal{O}(\sqrt{2^w})$, where the latter part comes from universal tables (ours also shrink due to the reduced $k$ and $s$). Note that our vector $P$ must be exactly of length $N$; padding is necessary otherwise. Both the padding and the universal tables will lose relevance for larger trees, as seen in the next section.

The next theorem summarizes our results in this section. We are able of handling trees of $\Theta((\frac{w}{c\log w})^c)$ nodes, for any $c > 3/2$.

THEOREM 4.1. *On a $w$-bit word RAM, for any constant $c > 3/2$, we can represent a sequence $P$ of $N = B^c$ parentheses, with sufficiently small $B = \Theta(\frac{w}{c\log w})$, computing all operations of Table 1 in $\mathcal{O}(c^2)$ time, with a data structure depending on $P$ that uses $N+2$ bits, and universal lookup tables (i.e., not depending on $P$) that use $\mathcal{O}(\sqrt{2^w})$ bits. The preprocessing time is $\mathcal{O}(N + \sqrt{2^w}\text{poly}(w))$ (the latter being needed only once for universal tables) and the working space is $\mathcal{O}(N)$ bits.*

In case we need to solve the operations that build on $P_1$ and $P_2$, we need to represent their corresponding $\phi$ functions (as $\psi$ is redundant). This can still be done with Lemma 4.6 using $\Phi = (2B+1)^{6c}$ and $(B+1)\log(2B+1) \le \frac{w}{12c}$. Theorem 4.1 applies verbatim.

## 5 A data structure for large trees

In practice, one can use the solution of the previous section for trees of any size, achieving $\mathcal{O}(\frac{k\log n}{w}\log_k n) = \mathcal{O}(\frac{\log n}{\log w - \log\log n}) = \mathcal{O}(\log n)$ time (using $k = w/\log n$) for all operations with an extremely simple and elegant data structure (especially if we choose to store arrays $m'$, etc. in simple form). In this section we show how to achieve constant time on trees of arbitrary size.

For simplicity, let us assume in this section that we handle trees of size $w^c$ in Section 4. We comment at the end the difference with the actual size $B^c$ handled.

For large trees with $n > w^c$ nodes, we divide the parentheses sequence into *blocks* of length $w^c$. Each block (containing a possibly non-balanced sequence of parentheses) is handled with the range min-max tree of Section 4.

Let $m_1, m_2, \ldots, m_\tau$; $M_1, M_2, \ldots, M_\tau$; and $e_1, e_2, \ldots, e_\tau$; be the minima, maxima, and excess of the $\tau = \lceil 2n/w^c \rceil$ blocks, respectively. These values are stored at the root nodes of each $T_{mM}$ tree and can be obtained in constant time.

### 5.1 Forward and backward searches on $\pi$

We consider extending *fwd-search*$(P, \pi, i, d)$ and *bwd-search*$(P, \pi, i, d)$ to trees of arbitrary size. We focus on *fwd-search*, as *bwd-search* is symmetric.

We first try to solve *fwd-search*$(P, \pi, i, d)$ within the block $j = \lfloor i/w^c \rfloor$ of $i$. If the answer is within block $j$, we are done. Otherwise, we must look for the first excess $d' = e_{j-1} + sum(P, \pi, 0, i - 1 - w^c \cdot (j-1)) + d$ in the following blocks (where the *sum* is local to block $j$). Then the answer lies in the first block $r > j$ such that $m_r \leq d' \leq M_r$. Thus, we can apply again Lemma 4.2, starting at $[m_{j+1}, M_{j+1}]$: If $d' \notin [m_{j+1}, M_{j+1}]$, we must either find the first $r > j+1$ such that $m_r \leq j$, or such that $M_r \geq j$. Once we find such block, we complete the operation with a local *fwd-search*$(P, \pi, 0, d' - e_{r-1})$ query inside it.

The problem is how to achieve constant-time search, for any $j$, in a sequence of length $\tau$. Let us focus on left-to-right minima, as the others are similar.

Definition 5. *Let $m_1, m_2, \ldots, m_\tau$ be a sequence of integers. We define for each $1 \leq j \leq \tau$ the* left-to-right minima *starting at $j$ as $lrm(j) = \langle j_0, j_1, j_2, \ldots \rangle$, where $j_0 = j$, $j_r < j_{r+1}$, $m_{j_{r+1}} < m_{j_r}$, and $m_{j_r+1} \ldots m_{j_{r+1}-1} \geq m_{j_r}$.*

The following lemmas are immediate.

Lemma 5.1. *The first element $\leq x$ after position $j$ in a sequence of integers $m_1, m_2, \ldots, m_\tau$ is $m_{j_r}$ for some $r > 0$, where $j_r \in lrm(j)$.*

Lemma 5.2. *Let $lrm(j)[p_j] = lrm(j')[p_{j'}]$. Then $lrm(j)[p_j + i] = lrm(j')[p_{j'} + i]$ for all $i > 0$.*

That is, once the $lrm$ sequences starting at two positions coincide in a position, they coincide thereafter. Lemma 5.2 is essential to store all the $\tau$ sequences $lrm(j)$ for each block $j$, in compact form. We form a tree $T_{lrm}$, which is essentially a trie composed of the reversed $lrm(j)$ sequences. The tree has $\tau$ nodes, one per block. Block $j$ is a child of block $j_1 = lrm(j)[1]$ (note $lrm(j)[0] = j_0 = j$), that is, $j$ is a child of the first block $j_1 > j$ such that $m_{j_1} < m_j$. Thus each $j$-to-root path spells out $lrm(j)$, by Lemma 5.2. We add a fictitious root to convert the forest into a tree. Note this structure is called 2d-Min-Heap by Fischer [11], who shows how to build it in linear time.

Example 4. *Figure 3 illustrates the tree built from the sequence $\langle m_1..m_9 \rangle = \langle 6, 4, 9, 7, 4, 4, 1, 8, 5 \rangle$. Then $lrm(1) = \langle 1, 2, 7 \rangle$, $lrm(2) = \langle 2, 7 \rangle$, $lrm(3) = \langle 3, 4, 5, 7 \rangle$, and so on.*

If we now assign weight $m_j - m_{j_1}$ to the edge between $j$ and its parent $j_1$, the original problem of finding the first $j_r > j$ such that $m_{j_r} \leq d'$ reduces to finding the first ancestor $j_r$ of node $j$ such that the sum of the weights between $j$ and $j_r$ exceeds $d'' = m_j - d'$. Thus we need to compute *weighted level ancestors* in $T_{lrm}$. Note that the weight of an edge in $T_{lrm}$ is at most $w^c$.

Lemma 5.3. *For a tree with $\tau$ nodes where each edge has an integer weight in $[1, W]$, after $\mathcal{O}(\tau \log^{1+\epsilon} \tau)$ time preprocessing, a weighted level-ancestor query is solved in $\mathcal{O}(t + 1/\epsilon)$ time on a $\Omega(\log(\tau W))$-bit word RAM. The size of the data structure is $\mathcal{O}(\tau \log \tau \log(\tau W) + \frac{\tau W t^t}{\log^t(\tau W)} + (\tau W)^{3/4})$ bits.*

*Proof.* We use a variant of Bender and Farach's $\langle \mathcal{O}(\tau \log \tau), \mathcal{O}(1) \rangle$ algorithm [4]. Let us ignore weights for a while. We extract a longest root-to-leaf path of the tree, which disconnects the tree into several subtrees. Then we repeat the process recursively for each subtree, until we have a set of paths. Each such path, say of length $\ell$, is extended upwards, adding other $\ell$ nodes towards the root (or less if the root is reached). The extended path is called a *ladder*, and its is stored as an array so that level-ancestor queries within a ladder are trivial. This guarantees that a node of height $h$ has also height $h$ in its path, and thus at least its first $h$ ancestors in its ladder. Moreover the union of all ladders has at most $2\tau$ nodes and thus requires $\mathcal{O}(\tau \log \tau)$ bits. For each tree node $v$, an array of its (at most) $\log \tau$ ancestors at depths $depth(v) - 2^i$, $i \geq 0$, is stored (hence the $\mathcal{O}(\tau \log \tau)$-number space and construction time). To solve the query *level-ancestor*$(v, d)$, where $d' = depth(v) - d$, the ancestor $v'$ at distance $d'' = 2^{\lfloor \log d' \rfloor}$ from $v$ is computed. Since $v'$ has height at least $d''$, it has at least its first $d''$ ancestors in its ladder. But from $v'$ we need only the ancestor at distance $d' - d'' < d''$, so the answer is in the ladder.

To include the weights, we must be able to find the node $v'$ and the answer considering the weights, instead of the number of nodes. We store for each ladder of length $\ell$ a sparse bitmap of length at most $\ell W$, where

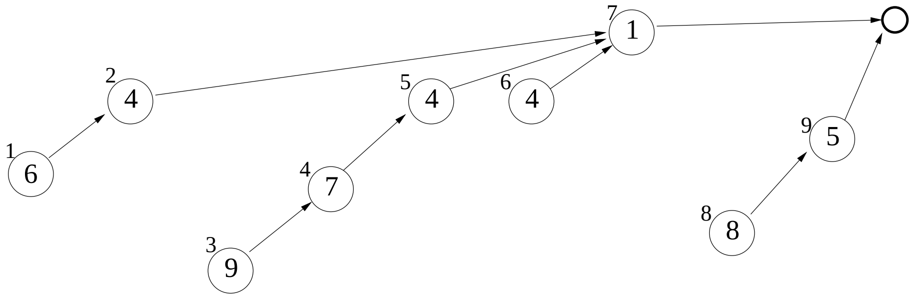


Figure 3: A tree representing the $lrm(j)$ sequences of values $m_1 \dots m_9$.

the $i$-th 1 left-to-right represents the $i$-th node upwards in the ladder, and the distance between two 1s, the weight of the edge between them. All the bitmaps are concatenated into one (so each ladder is represented by a couple of integers indicating the extremes of its bitmap). This long bitmap contains at most $2\tau$ 1s, and because weights do not exceed $W$, at most $2\tau W$ 0s. Using Pătraşcu's sparse bitmaps [30], it can be represented using $\mathcal{O}(\tau \log W + \frac{\tau W t^t}{\log^t(\tau W)} + (\tau W)^{3/4})$ bits and do *rank*/*select* in $\mathcal{O}(t)$ time.

In addition, we store for each node the $\log \tau$ accumulated weights towards ancestors at distances $2^i$, using fusion trees [15]. These can store $z$ keys of $\ell$ bits in $\mathcal{O}(z\ell)$ bits and, using $\mathcal{O}(z^{5/6}(z^{1/6})^4) = \mathcal{O}(z^{1.5})$ preprocessing time, answer predecessor queries in $\mathcal{O}(\log_\ell z)$ time (via an $\ell^{1/6}$-ary tree). The $1/6$ can be reduced to achieve $\mathcal{O}(z^{1+\epsilon})$ preprocessing time and $\mathcal{O}(1/\epsilon)$ query time for any desired constant $0 < \epsilon \leq 1/2$.

In our case this means $\mathcal{O}(\tau \log \tau \log(\tau W))$ bits of space, $\mathcal{O}(\tau \log^{1+\epsilon} \tau)$ construction time, and $\mathcal{O}(1/\epsilon)$ access time. Thus we can find in constant time, from each node $v$, the corresponding weighted ancestor $v'$ using a predecessor query. If this corresponds to distance $2^i$, then the true ancestor is at distance $< 2^{i+1}$, and thus it is within the ladder of $v'$, where it is found using *rank*/*select* on the bitmap of ladders (each node $v$ has a pointer to its 1 in the ladder corresponding to the path it belongs to). □

To apply this lemma for our problem of computing *fwd-search* outside blocks, we have $W = w^c$ and $\tau = \frac{n}{w^c}$. Then the size of the data structure becomes $\mathcal{O}(\frac{n \log^2 n}{w^c} + \frac{n\, t^t}{\log^t n} + n^{3/4})$. By choosing $\epsilon = \min(1/2, 1/c^2)$, the query time is $\mathcal{O}(c^2 + t)$ and the preprocessing time is $\mathcal{O}(n)$ for $c \geq 1.47$.

### 5.2 Other operations

For computing *rmqi* and *RMQi*, we use a simple data structure [3] on the $m_r$ and $M_r$ values, later improved to require only $\mathcal{O}(\tau)$ bits on top of the sequence of values [35, 12]. The extra space is thus $\mathcal{O}(n/w^c)$ bits, and it solves any query up to the block granularity. For solving a general query $[i, j]$ we should compare the minimum/maximum obtained with the result of running queries *rmqi* and *RMQi* within the blocks at the two extremes of the boundary $[i, j]$.

For the remaining operations, we define *pioneers* [20]. We divide the parentheses sequence $P[0, 2n-1]$ into blocks of length $w^c$. Then we extract pairs $(i, j)$ of matching parentheses ($j = \mathit{findclose}(i)$) such that $i$ and $j$ belong to different blocks. If we consider a graph whose vertex set consists of the blocks and whose edge set consists of the pairs of parentheses, the graph is outer-planar. To remove multiple edges, we choose the tightest pair of parentheses for each pair of vertices. These parentheses are called pioneers. Because pioneers correspond to the edges (without multiplicity) of an outer-planar graph, their number is $\mathcal{O}(n/w^c)$. Furthermore, they form another balanced parentheses sequence $P'$ representing an ordinal tree with $\mathcal{O}(n/w^c)$ nodes.

To encode $P'$ we use a compressed bit vector $C[0, 2n-1]$ such that $C[i] = 1$ indicates that parenthesis $P[i]$ is a pioneer. Using again Pătraşcu's result [30], vector $C$ can be represented in at most $\frac{n}{w^c} \log(w^c) + \mathcal{O}(\frac{n\, t^t}{\log^t n} + n^{3/4})$ bits, so that operations *rank* and *select* can be computed in $\mathcal{O}(t)$ time.

For computing *child* and *child-rank*, it is enough to consider only nodes which completely include a block (otherwise the query is solved in constant time by considering just two adjacent blocks). Furthermore, among them, it is enough to consider pioneers because if pair $(i, j)$, with $i$ and $j$ in different blocks, is not a pioneer, then it must contain a pioneer matching pair $(i', j')$, with $i'$ in the same block of $i$ and $j'$ in the same block of $j$. Thus $i'$ is a descendant of $i$ and all the children of $i$ start within $[i+1, i']$ or within $[j'+1, j-1]$, thus all are contained in two blocks. Hence computing *child*$(i, q)$ and *child-rank* for a child of $i$ can be done in constant time by considering just these two blocks.

Thus we only need to care about pioneer nodes containing at least one block; let us call *marked* these nodes, of which there are only $\mathcal{O}(n/w^c)$. We focus on the children of marked nodes placed at the blocks fully contained in them, as the others lie in at most the two extreme blocks and can be dealt with in constant time.

For each marked node $v$ we store a list formed by the blocks fully contained in $v$, and that contain (starting positions of) children of $v$. Since each block contains children of at most one marked node fully containing the block, each block belongs to at most one list, and it stores its position in the list it belongs to. All this data occupies $\mathcal{O}(\frac{n \log n}{w^c})$ bits. In addition, the contained blocks store the number of children of $v$ that start within them. The sequence of number of children formed for each marked node $v$ is stored as gaps between consecutive 1s in a bitmap $C_v$. All these lists together contain at most $n$ 0s and $\mathcal{O}(n/w^c)$ 1s, and thus can be stored within the same space bounds of the other bitmaps in this section.

Using this bitmap *child* and *child-rank* can easily be solved using *rank* and *select*. For $child(v, q)$ on a marked node $v$ we start using $p = rank_1(C_v, select_0(C_v, q))$. This tells the position in the list of blocks of $v$ where the $q$-th child of $v$ lies. Then the answer corresponds to the $q'$-th minimum within that block, for $q' = q - rank_0(select_1(C_v, p))$. For $child\text{-}rank(u)$, where $v = parent(u)$ is marked, we start with $z = rank_0(C_v, select_1(C_v, p_u))$, where $p_u$ is the position of the block of $u$ within the list of $v$. Then we add to $z$ the number of minima in the block of $u$ until $u - 1$.

For *degree*, similar arguments show that we only need to consider marked nodes, for which we simply store all the answers within $\mathcal{O}(\frac{n \log n}{w^c})$ bits of space.

Finally, the remaining operations require just *rank* and *select* on $P$, or the virtual bit vectors $P_1$ and $P_2$. We can make up a sequence with the accumulated number of 1s in each of the $\tau$ blocks. The numbers add up to $\mathcal{O}(n)$ and thus can be represented as gaps of 0s between consecutive 1s in a bitmap, which can be stored within the previous space bounds. Performing *rank* and *select* on this bitmap, in time $\mathcal{O}(t)$, lets us know in which block must we finish the query, using its range min-max tree.

### 5.3 The final result

Recalling Theorem 4.1, we have $\mathcal{O}(n/B^c)$ blocks, for $B = \mathcal{O}(\frac{w}{c \log w})$. The sum of the space for all the blocks is $2n + \mathcal{O}(n/B^c)$, plus shared universal tables that add up to $\mathcal{O}(\sqrt{2^w})$ bits. Padding the last block to size exactly $B^c$ adds up another negligible extra space.

On the other hand, in this section we have extended the results to larger trees of $n$ nodes, adding time $\mathcal{O}(t)$ to the operations. By properly adjusting $w$ to $B$ in the results, the overall extra space added is $\mathcal{O}(\frac{n(c \log B + \log^2 n)}{B^c} + \frac{n\, t^t}{\log^t n} + \sqrt{2^B} + n^{3/4})$ bits. Assuming pessimistically $w = \log n$, setting $t = c^2$, and replacing $B$, we get that the time for any operation is $\mathcal{O}(c^2)$, and the total space simplifies to $2n + \mathcal{O}(\frac{n \log^c \log n}{\log^{c-2} n})$.

Construction time is $\mathcal{O}(n)$. We now analyze the working space for constructing the data structure. We first convert the input balanced parentheses sequence $P$ into a set of aB-trees, each of which represents a part of the input of length $B^c$. The working space is $\mathcal{O}(B^c)$ from Theorem 4.1. Next we compute pioneers: We scan $P$ from left to right, and if $P[i]$ is an opening parenthesis, we push $i$ in a stack, and if it is closing, we pop an entry from the stack. Because $P$ is nested, the values in the stack are monotone. Therefore we can store a new value as the difference from the previous one using unary code. Thus the values in the stack can be stored in $\mathcal{O}(n)$ bits. Encoding and decoding the stack values takes $\mathcal{O}(n)$ time in total. It is easy to compute pioneers from the stack. Once the pioneers are identified, Pătraşcu's compressed representation [30] of bit vector $C$ is built in $\mathcal{O}(n)$ space too, as it also cuts the bitmap into polylog-sized aB-trees and then computes some directories over just $\mathcal{O}(n/\text{polylog}(n))$ values.

The remaining data structures, such as the *lrm* sequences and tree, the lists of the marked nodes, and the $C_v$ bitmaps, are all built on $\mathcal{O}(n/B^c)$ elements, thus they need at most $\mathcal{O}(n)$ bits of space for construction.

By rewriting $c-2-\delta$ as $c$, for any constant $\delta > 0$, we get our main result on static ordinal trees, Theorem 1.1.

## 6 A simple data structure for dynamic trees

In this section we give a simple data structure for dynamic ordinal trees. In addition to the previous query operations, we add now insertion and deletion of internal nodes and leaves. We then consider a more sophisticated representation giving sublogarithmic time for almost all of the operations.

### 6.1 Memory management

We store a 0,1 vector $P[0, 2n-1]$ using a dynamic min-max tree. Each leaf of the min-max tree stores a segment of $P$ in verbatim form. The length $\ell$ of each segment is restricted to $L \le \ell \le 2L$ for some parameter $L > 0$.

If insertions or deletions occur, the length of a segment will change. We use a standard technique for dynamic maintenance of memory cells [24]. We regard the memory as an array of cells of length $2L$ each, hence allocation is easily handled in constant time. We use $L+1$ linked lists $s_L, \ldots, s_{2L}$ where $s_i$ stores all the segments of length $i$. All the segments with equal length

$i$ are packed consecutively, without wasting any extra space, in the cells of linked list $s_i$. Therefore a cell (of length $2L$) stores (parts of) at most three segments, and a segment spans at most two cells. Tree leaves store pointers to the cell and offset where its segment is stored. If the length of a segment changes from $i$ to $j$, it is moved from $s_i$ to $s_j$. The space generated by the removal is filled with the head segment in $s_i$, and the removed segment is stored at the head of $s_j$.

With this scheme, scanning any segment takes $\mathcal{O}(L/\log n)$ time, by processing it by chunks of $\Theta(\log n)$ bits. This is also the time to compute operations *fwd-search*, *bwd-search*, *rmqi*, etc. on the segment, using proper universal tables. Migrating a node to another list is also done in $\mathcal{O}(L/\log n)$ time.

If a migration of a segment occurs, pointers to the segment from a leaf of the tree must change. For this sake we store back-pointers from each segment to its leaf. Each cell stores also a pointer to the next cell of its list. Finally, an array of pointers for the heads of $s_L, \ldots, s_{2L}$ is necessary. Overall, the space for storing a 0,1 vector of length $2n$ is $2n + \mathcal{O}(\frac{n \log n}{L})$ bits.

The rest of the dynamic tree will use sublinear space, and thus we allocate fixed-size memory cells for the internal nodes, as they will waste at most a constant fraction of the allocated space.

### 6.2 A dynamic tree

We give a simple dynamic data structure representing an ordinal tree with $n$ nodes using $2n + \mathcal{O}(n/\log n)$ bits, and supporting all query and update operations in $\mathcal{O}(\log n)$ worst-case time.

We divide the 0,1 vector $P[0, 2n-1]$ into segments of length from $L$ to $2L$, for $L = \log^2 n$. We use a balanced binary tree for representing the range min-max tree. If a node of the tree corresponds to a vector $P[i, j]$, the node stores $i$ and $j$, as well as $e = \mathit{sum}(P, \pi, i, j)$, $m = \mathit{rmq}(P, \pi, i, j)$, $M = \mathit{RMQ}(P, \pi, i, j)$, and $n$, the number of minimum values in $P[i, j]$ regarding $\pi$. (Data on $\phi$ for the virtual vectors $P_1$ and $P_2$ is handled analogously.)

It is clear that *fwd-search*, *bwd-search*, *rmqi*, *RMQi*, *rank*, *select*, *degree*, *child* and *child-rank* can be computed in $\mathcal{O}(\log n)$ time, by using the same algorithms developed for small trees in Section 4. These operations cover all the functionality of Table 1. Note the values we store are local to the subtree (so that they are easy to update), but global values are easily derived in a top-down traversal. For example, to solve *fwd-search*$(P, \pi, i, d)$ starting at the min-max tree root $v$ with children $v_l$ and $v_r$, we first compute the desired global excess $d' = E[i-1] + d$, where $E[i-1]$ is found in a top-down traversal towards position $i-1$, adding up $e(v_l)$ each time we descend to $v_r$. Once we obtain $d'$, we start again at the root and see if $j(v_l) \geq i$, in which case try first on $v_l$. If the answer is not there or $j(v_l) < i$, we try on $v_r$, now seeking excess $d' - e(v_l)$.

Because each node uses $\mathcal{O}(\log n)$ bits, and the number of nodes is $\mathcal{O}(n/L)$, the total space is $2n + \mathcal{O}(n/\log n)$ bits. This includes the extra $\mathcal{O}(\frac{n \log n}{L})$ term for the leaf data. Note that we need to maintain several universal tables that handle chunks of $\frac{1}{2}\log n$ bits. These require just $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}(n))$ extra bits.

If insertion/deletion occurs, we update a segment, and the stored values in the leaf for the segment. If the length of the segment exceeds $2L$, we split it into two and add a new node. If the length becomes shorter than $L$, we find the adjacent segment to the right. If its length is $L$, we concatenate them; otherwise move the leftmost bit of the right segment to the left one. In this manner we can keep the invariant that all segments have length $L$ to $2L$. Then we update all the values in the ancestors of the modified leaves. If a balancing operation occurs, we also update the values in nodes. All these updates are easily carried out in constant time per involved node, as the values to update are minima, maxima, and sum over the two children values. Thus the update time is also $\mathcal{O}(\log n)$.

When $\lceil \log n \rceil$ changes, we must update the allowed values for $L$, recompute universal tables, change the width of the stored values, etc. Mäkinen and Navarro [23] have shown how to do this for a very similar case (dynamic *rank*/*select* on a bitmap). Their solution of splitting the bitmap into 5 parts and moving border bits across parts to deamortize the work applies verbatim to our case, thus we can handle changes in $\lceil \log n \rceil$ without altering the space nor the time complexity (except for $\mathcal{O}(w)$ extra bits in the space due to a constant number of system-wide pointers, a technicism we ignore). This applies to the next solution too, where we will omit the issue.

## 7 A faster dynamic data structure

Instead of the balanced binary tree, we use a B-tree with branching factor $\Theta(\sqrt{\log n})$, as in previous work [6]. Then the depth of the tree is $\mathcal{O}(\log n/\log\log n)$. The lengths of segments is $L$ to $2L$ for $L = \log^2 n/\log\log n$. The required space for the range min-max tree and the vector is now $2n + \mathcal{O}(n \log\log n/\log n)$ bits (the internal nodes use $\mathcal{O}(\log^{3/2} n)$ bits but there are only $\mathcal{O}(\frac{n}{L\sqrt{\log n}})$ internal nodes). Now each leaf can be processed in time $\mathcal{O}(\log n/\log\log n)$.

Each internal node $v$ of the range min-max tree has $k$ children, for $\sqrt{\log n} \leq k \leq 2\sqrt{\log n}$. Let $c_1, c_2, \ldots, c_k$ be the children of $v$, and $[\ell_1..r_1], \ldots, [\ell_k..r_k]$ be their corresponding subranges. We store (*i*) the children boundaries $\ell_i$, (*ii*) $s_\phi[1,k]$ and $s_\psi[1,k]$ storing $s_{\phi/\psi}[i] = sum(P, \phi/\psi, \ell_1, r_i)$, (*iii*) $e[1,k]$ storing $e[i] = sum(P, \pi, \ell_1, r_i)$, (*iv*) $m[1,k]$ storing $m[i] = e[i-1] + rmq(P, \pi, \ell_i, r_i)$, and $M[1,k]$ storing $M[i] = e[i-1] + RMQ(P, \pi, \ell_i, r_i)$. Note that the values stored are local to the subtree (as in the simpler balanced binary tree version) but cumulative with respect to previous siblings. Note also that storing $s_\phi$, $s_\psi$ and $e$ is redundant, as noted in Section 4.3, but we need them in explicit form to achieve constant-time searching into their values.

Apart from simple accesses, we need to support the following operations within a node:

- $p(i)$: the largest $j$ such that $\ell_{j-1} \le i$ (or $j = 1$).
- $w_{\phi/\psi}(i)$: the largest $j$ such that $s_{\phi/\psi}[j-1] \le i$ (or $j = 1$).
- $f(i,d)$: the smallest $j \ge i$ such that $m[j] \le d \le M[j]$.
- $b(i,d)$: the largest $j \le i$ such that $m[j] \le d \le M[j]$.
- $r(i,j,t)$: the $t$-th $x$ such that $m[x]$ is minimum in $m[i,j]$.
- $R(i,j,t)$: the $t$-th $x$ such that $M[x]$ is maximum in $M[i,j]$.
- $n(i,j)$: the number of times the minimum occurs in $m[i,j]$.
- *update*: updates the data structure upon $\pm 1$ changes in some child.

Operations *fwd-search*/*bwd-search* can then be carried out via $\mathcal{O}(\log n/\log\log n)$ applications of $f(i,d)/b(i,d)$. Recalling Lemma 4.1, the interval of interest is partitioned into $\mathcal{O}(\sqrt{\log n} \cdot \log n/\log\log n)$ nodes of the B-tree, but these can be grouped into $\mathcal{O}(\log n/\log\log n)$ sequences of siblings. Within each such sequence a single $f(i,d)/b(i,d)$ operation is sufficient. Once the answer of interest $j$ is finally found within some internal node, we descend to its $j$-th child and repeat the search until finding the correct leaf, again in $\mathcal{O}(\log n/\log\log n)$ applications of $f(i,d)/b(i,d)$. Operations *rmqi* and *RMQi* are solved in very similar fashion, using $\mathcal{O}(\log n/\log\log n)$ applications of $r(i,j,1)/R(i,j,1)$. Also, operations *rank* and *select* on $P$ are carried out in obvious manner with $\mathcal{O}(\log n/\log\log n)$ applications of $p(i)$ and $w_{\phi/\psi}(i)$. Handling $\phi$ for $P_1$ and $P_2$ is immediate; we omit it.

For *degree* we partition the interval as for *rmqi* and then use $m[r(i,j,1)]$ in each node to identify those holding the global minimum. For each node holding the minimum, $n(i,j)$ gives the number of occurrences of the minimum in the node. Thus we apply $r(i,j,1)$ and $n(i,j)$ $\mathcal{O}(\log n/\log\log n)$ times. Operation *child-rank* is very similar, by changing the right end of the interval of interest, as before. Finally, solving *child* is also similar, except that when we exceed the desired rank in the sum (i.e., in some node $n(i,j) \ge t$, where $t$ is the local rank of the child we are looking for), we find the desired min-max tree branch with $r(i,j,t)$, and continue until finding the proper leaf with one $r(i,j,t)$ operation per level.

By using the dynamic partial sums data structure [32] and the Super-Cartesian tree [13], we obtain:

Lemma 7.1. *For a 0,1 vector of length $2n$, there exists a data structure using $2n + \mathcal{O}(n\log\log n/\log n)$ bits supporting fwd-search and bwd-search in $\mathcal{O}(\log n)$ time, and all other operations (including update) in $\mathcal{O}(\log n/\log\log n)$ time.*

In many operations to support, we carry out *fwd-search*$(P, \pi, i, d)$ or *bwd-search*$(P, \pi, i, d)$ for a small constant $d$. Those particular cases can be made more efficient.

Lemma 7.2. *For a 0,1 vector $P$, fwd-search$(P, \pi, i, d)$ and bwd-search$(P, \pi, i, d)$ can be computed in $\mathcal{O}(d + \log n/\log\log n)$ time.*

The proofs will be given in the full paper.

This completes our main result in this section, Theorem 1.2.

## 8 Concluding remarks

In this paper we have proposed flexible and powerful data structures for the succinct representation of ordinal trees. For the static case, all the known operations are done in constant time using $2n + \mathcal{O}(n/\mathrm{polylog}(n))$ bits of space, for a tree of $n$ nodes. This largely improves the redundancy of previous representations, by building on a recent result [30]. The core of the idea is the range min-max tree, which has independent interest. This simple data structure reduces all of the operations to a handful of primitives, which run in constant time on polylog-sized subtrees. It can be used in standalone form to obtain a simple and practical implementation that achieves $\mathcal{O}(\log n)$ time for all the operations. We then achieve constant time by using the range min-max tree as a building block for handling larger trees.

For the dynamic case, there have been no data structures supporting several of the usual tree operations. The data structures of this paper support all of the operations, including node insertion and deletion, in $\mathcal{O}(\log n)$ time, and a variant supports most of them in $\mathcal{O}(\log n/\log\log n)$ time. They are based on dynamic range min-max trees, and especially the former is extremely simple and can be easily implemented.

Future work includes reducing the time complexities for all of the operations in the dynamic case to $\mathcal{O}(\log n/\log\log n)$, as well as trying to improve the redundancy (this is $\mathcal{O}(n/\log n)$ for the simpler structure and $\mathcal{O}(n\log\log n/\log n)$ for the more complex one).

**Acknowledgments**

We thank Mihai Pătraşcu for confirming us the construction cost of his aB-tree and rank/select data structure [30].

## References

[1] D. Arroyuelo. An improved succinct representation for dynamic $k$-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 277–289, 2008.

[2] J. Barbay, J. I. Munro, M. He, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[3] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 1776, pages 88–94, 2000.

[4] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[5] 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.

[6] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.

[7] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.

[8] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 134–145. LNCS 4007, 2006.

[9] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, LNCS 5124, pages 173–184, 2008.

[10] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[11] J. Fischer. Optimal succinctness for range minimum queries. *CoRR*, abs/0812.2775, 2008.

[12] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, LNCS 4614, pages 459–470, 2007.

[13] J. Fischer and V. Heun. Range median of minima queries, super-cartesian trees, and text indexing. In *Proc. 19th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 239–252, 2008.

[14] M. Fredman and M. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[15] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and Systems Science*, 47(3):424–436, 1993.

[16] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 159–172, 2004.

[17] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[18] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. 15th Annual European Symposium on Algorithms (ESA)*, pages 371–382. LNCS 4698, 2007.

[19] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4596, pages 509–520, 2007.

[20] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[21] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[22] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms (TALG)*, 4(3):article 28, 2008.

[23] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008.

[24] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time. *Journal of Computer System Sciences*, 33(1):66–74, 1986.

[25] J. I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

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

[27] J. I. Munro, V. Raman, and S. S. Rao. Space efficient

suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[28] J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536, 2001.

[29] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. 31th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 3142, pages 1006–1015, 2004.

[30] M. Pătraşcu. Succincter. In *Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[31] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[32] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proc. 7th Annual Workshop on Algorithms and Data Structures (WADS)*, LNCS 2125, pages 426–437, 2001.

[33] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[34] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 357–368, 2003.

[35] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.