Fully-Functional Static and Dynamic Succinct Trees

GONZALO NAVARRO, University of Chile, Chile
KUNIHIKO SADAKANE, National Institute of Informatics, Japan

We propose new succinct representations of ordinal trees, and match various space/time lower bounds. It is known that any n-node static tree can be represented in $2n + o(n)$ bits so that a number of operations on the tree can be supported in constant time under the word-RAM model. However, the data structures are complicated and difficult to dynamize. 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 that are carried out in constant time on polylog-sized trees. The result is extended to trees of arbitrary size, retaining constant time and reaching $2n + \mathcal{O}(n/\text{polylog}(n))$ bits of space. This space is optimal for a core subset of the operations supported, and significantly lower than in any previous proposal.

For the dynamic case, where insertion/deletion (indels) of nodes is allowed, the existing data structures support a very limited set of operations. Our data structure builds on the range min-max tree to achieve $2n+\mathcal{O}(n/\log n)$ bits of space and $\mathcal{O}(\log n)$ time for all the operations supported in the static scenario, plus indels. We also propose an improved data structure using $2n + \mathcal{O}(n \log\log n/\log n)$ bits and improving the time to the optimal $\mathcal{O}(\log n/\log\log n)$ for most operations. We extend our support to forests, where whole subtrees can be attached to or detached from others, in time $\mathcal{O}(\log^{1+\epsilon} n)$ for any $\epsilon > 0$. Such operations had not been considered before.

Our techniques are of independent interest. An immediate derivation yields an improved solution to range minimum/maximum queries where consecutive elements differ by ± 1, achieving $n + \mathcal{O}(n/\text{polylog}(n))$ bits of space. A second one stores an array of numbers supporting operations *sum* and *search* and limited updates, in optimal time $\mathcal{O}(\log n/\log\log n)$. A third one allows representing dynamic bitmaps and sequences over alphabets of size σ, supporting rank/select and indels, within zero-order entropy bounds and time $\mathcal{O}(\log n \log \sigma/(\log\log n)^2)$ for all operations. This time is the optimal $\mathcal{O}(\log n/\log\log n)$ on bitmaps and polylog-sized alphabets. This improves upon the best existing bounds for entropy-bounded storage of dynamic sequences, compressed full-text self-indexes, and compressed-space construction of the Burrows-Wheeler transform.

Categories and Subject Descriptors: E.1 [**Data structures**]; E.2 [**Data storage representations**]

General Terms: Algorithms

Additional Key Words and Phrases: Succinct tree representations, compressed sequence representations, compressed text databases

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

Work funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile (first author), and a Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan (second author).
Authors' address: G. Navarro, Department of Computer Science, University of Chile, Santiago, Chile. gnavarro@dcc.uchile.cl. Kunihiko Sadakane, Principles of Informatics Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan. sada@nii.ac.jp.
A preliminary version of this paper appeared in *Proc. SODA 2010*, pp. 134–149.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

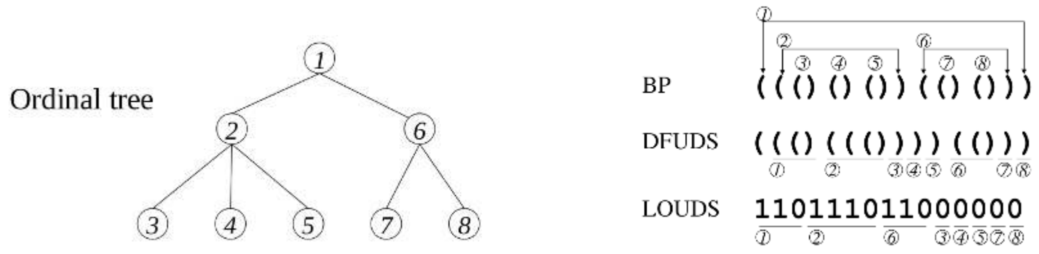

Fig. 1. Succinct representations of trees.

distinguish all the nodes, it requires $\log n$ bits[1] in the worst case. Therefore the tree occupies $\Theta(n \log n)$ bits. This causes a space problem for storing a large set of items in a tree. Much research has been devoted to reducing the space to represent static trees [Jacobson 1989; Munro and Raman 2001; Munro et al. 2001a; Munro and Rao 2004; Benoit et al. 2005; Chiang et al. 2005; Geary et al. 2006a; Geary et al. 2006b; Delpratt et al. 2006; He et al. 2007; Golynski et al. 2007; Raman et al. 2007; Sadakane 2007a; Lu and Yeh 2008; Farzan and Munro 2008; Ferragina et al. 2009; Farzan et al. 2009; Barbay et al. 2011b; Jansson et al. 2012] and dynamic trees [Munro et al. 2001b; Raman and Rao 2003; Chan et al. 2007; Arroyuelo 2008; Farzan and Munro 2011; Davoodi and Rao 2011; Joannou and Raman 2012], achieving so-called *succinct data structures* for trees.

A succinct data structure stores objects using space asymptotically equal to the information-theoretic lower bound, while simultaneously supporting a number of primitive operations on the objects in constant time. Here 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 a succinct data structure storing an object from the universe is thus $(1 + o(1)) \log L$ bits.

In this paper we are interested in *ordinal trees*, in which the children of a node are ordered. The information-theoretic lower bound for representing 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 [Munro and Raman 2001]. 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 $\mathcal{O}(1)$ time.

Basically there exist three types of succinct representations of ordinal trees: the balanced parentheses sequence (BP) [Jacobson 1989; Munro and Raman 2001], the level-order unary degree sequence (LOUDS) [Jacobson 1989; Delpratt et al. 2006], and the depth-first unary degree sequence (DFUDS) [Benoit et al. 2005; Jansson et al. 2012]. An example of them is shown in Figure 1. LOUDS is a simple representation, but it lacks many basic operations, such as the subtree size of a given 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 is that it supports a more complete set of operations by simple primitives, most notably going to the i-th child of a node in constant time. In this paper we focus on the BP representation, and achieve constant time for a large set of operations, including all those handled with DFUDS. Moreover, as we manipulate a sequence of balanced parentheses, our data structure can be used to implement a DFUDS representation as well.

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

1.1. Our contributions

We propose new succinct data structures for ordinal trees encoded with balanced parentheses, in both static and dynamic scenarios.

Static succinct trees. For the static case we obtain the following result.

THEOREM 1.1. *For any ordinal tree with n nodes, all operations in Table I except insert and delete are carried out in constant time $\mathcal{O}(c)$ 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 of space.*

The space complexity of our data structures significantly improves upon the lower-order term achieved in previous representations. For example, the extra data structure for *level_anc* requires $\mathcal{O}(n \log\log n/\sqrt{\log n})$ bits [Munro and Rao 2004], or $\mathcal{O}(n(\log\log n)^2/\log n)$ bits[2] [Jansson et al. 2012], and that for *child* requires $\mathcal{O}(n/(\log\log n)^2)$ bits [Lu and Yeh 2008]. Ours requires $\mathcal{O}(n/\log^c n)$ bits for all of the operations. We show in the Conclusions that this redundancy is optimal if a core subset of the operations is to be supported.

The simplicity and space-efficiency of our data structures stem from the fact that any query operation in Table I 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 sum over all the data structures. For example, the first succinct representation of BP [Munro and Raman 2001] 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* [Munro et al. 2001a], *lca* [Sadakane 2007a], *degree* [Chiang et al. 2005], *child* and *child_rank* [Lu and Yeh 2008], *level_anc* [Munro and Rao 2004], 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* [Geary et al. 2006a]. This reduces the size of the data structure for these basic operations, but still needs extra auxiliary data structures for other operations.

Dynamic succinct trees. Our approach is suitable for the dynamic maintenance of trees. Former approaches in the static case use two-level data structures to reduce the size, which causes difficulties in the dynamic case. On the other hand, our approach using the range min-max tree is easily applied in this scenario, resulting in simple and efficient dynamic data structures. This is illustrated by the fact that all the operations are supported. The following theorem summarizes our results.

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 I, using a data structure that requires $2n + \mathcal{O}(n \log\log n/\log n)$ bits. Alternatively, all the operations of the table can be carried out in $\mathcal{O}(\log n)$ time using $2n + \mathcal{O}(n/\log n)$ bits of space.*

Note that Table I offers two variants. In variant 1, we achieve $\mathcal{O}(\log n/\log\log n)$ for most operations, including *insert* and *delete*, but we solve *degree*, *child*, and *child_rank* naively. In variant 2, we achieve $\mathcal{O}(\log n)$ complexity for these, yet also for *insert* and *delete*.[3] The theorem also offers an alternative where we solve all the operations in time $\mathcal{O}(\log n)$, with

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

[3]In the conference version of this paper [Sadakane and Navarro 2010] we erroneously affirm we can obtain $\mathcal{O}(\log n/\log\log n)$ for all these operations, as well as *level_anc*, *level_next*/*level_prev*, and *level_lmost*/*level_rmost*, for which we can actually obtain only $\mathcal{O}(\log n)$.

Table I. Operations supported by our data structure. The time complexities are for the dynamic case; in the static case all operations take $\mathcal{O}(1)$ time. The first group, of basic operations, is used to implement the others, but could have other uses.

operation	description	time complexity	
		variant 1	variant 2
$inspect(i)$	$P[i]$	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$findclose(i)$ / $findopen(i)$	position of parenthesis matching $P[i]$	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$enclose(i)$	position of tightest open parent. enclosing i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$rank_{\boxed{(}}(i)$ / $rank_{\boxed{)}}(i)$	number of open/close parentheses in $P[0,i]$	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$select_{\boxed{(}}(i)$ / $select_{\boxed{)}}(i)$	position of i-th open/close parenthesis	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$rmqi(i,j)$ / $RMQi(i,j)$	position of min/max excess value in range $[i,j]$	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$pre\_rank(i)$ / $post\_rank(i)$	preorder/postorder rank of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$pre\_select(i)$ / $post\_select(i)$	the node with preorder/postorder i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$isleaf(i)$	whether $P[i]$ is a leaf	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$isancestor(i,j)$	whether i is an ancestor of j	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$depth(i)$	depth of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$parent(i)$	parent of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$first\_child(i)$ / $last\_child(i)$	first/last child of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$next\_sibling(i)$ / $prev\_sibling(i)$	next/previous sibling of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$subtree\_size(i)$	number of nodes in the subtree of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$level\_anc(i,d)$	ancestor j of i s.t. $depth(j) = depth(i) - d$	$\mathcal{O}(\log n)$	
$level\_next(i)$ / $level\_prev(i)$	next/previous node of i in BFS order	$\mathcal{O}(\log n)$	
$level\_lmost(d)$ / $level\_rmost(d)$	leftmost/rightmost node with depth d	$\mathcal{O}(\log n)$	
$lca(i,j)$	the lowest common ancestor of two nodes i,j	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$deepest\_node(i)$	the (first) deepest node in the subtree of i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$height(i)$	the height of i (distance to its deepest node)	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$degree(i)$	q = number of children of node i	$\mathcal{O}(\frac{q\log n}{\log\log n})$	$\mathcal{O}(\log n)$
$child(i,q)$	q-th child of node i	$\mathcal{O}(\frac{q\log n}{\log\log n})$	$\mathcal{O}(\log n)$
$child\_rank(i)$	q = number of siblings to the left of node i	$\mathcal{O}(\frac{q\log n}{\log\log n})$	$\mathcal{O}(\log n)$
$in\_rank(i)$	inorder of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$in\_select(i)$	node with inorder i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$leaf\_rank(i)$	number of leaves to the left of leaf i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$leaf\_select(i)$	i-th leaf	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$lmost\_leaf(i)$ / $rmost\_leaf(i)$	leftmost/rightmost leaf of node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	
$insert(i,j)$	insert node given by matching parent. at i and j	$\mathcal{O}(\frac{\log n}{\log\log n})$	$\mathcal{O}(\log n)$
$delete(i)$	delete node i	$\mathcal{O}(\frac{\log n}{\log\log n})$	$\mathcal{O}(\log n)$

the reward of reducing the extra space to $\mathcal{O}(n/\log n)$. We give yet another tradeoff later, in Lemma 7.4.

The time complexity $\mathcal{O}(\log n/\log\log n)$ is optimal: Chan et al. [2007, Thm. 5.2] showed that just supporting the most basic operations of Table I (*findopen*, *findclose*, and *enclose*, as we will see) plus *insert* and *delete*, requires this time even in the amortized sense, by a reduction from Fredman and Saks [1989] lower bounds on *rank* queries.

Moreover, we are able to attach and detach whole subtrees, in time $\mathcal{O}(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$ (see Section 2.3 for the precise details). These operations had never been considered before in succinct tree representations, and may find various applications.

Byproducts. Our techniques are of more general interest. A subset of our data structure is able to solve the well-known "range minimum query" problem [Bender and Farach-Colton 2000]. In the important case where consecutive elements differ by ± 1, we improve upon the best current space redundancy of $\mathcal{O}(n\log\log n/\log n)$ bits [Fischer 2010].

COROLLARY 1.3. *Let $E[0, n-1]$ be an array of numbers with the property that $E[i] - E[i-1] \in \{-1, +1\}$ for $0 < i < n$, encoded as a bit vector $P[0, n-1]$ such that $P[i] = 1$ if $E[i] - E[i-1] = +1$ and $P[i] = 0$ otherwise. Then, in a RAM machine we can preprocess P in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ bits such that range maximum/minimum queries are answered in constant $\mathcal{O}(c)$ time and $\mathcal{O}(n/\log^c n)$ extra bits on top of P.*

Another direct application, to the representation of a dynamic array of numbers, yields an improvement to the best current alternative [Mäkinen and Navarro 2008] by a $\Theta(\log\log n)$ time factor. If the updates are limited, further operations *sum* (that gives the sum of the numbers up to some position) and *search* (that finds the position where a given sum is exceeded) can be supported, and our complexity matches the lower bounds for *searchable partial sums* by Pătraşcu and Demaine [2006] (if the updates are not limited one can still use previous results [Mäkinen and Navarro 2008], which are optimal in that general case). We present our result in a slightly more general form.

LEMMA 1.4. *A sequence of n variable-length constant-time self-delimiting*[4] *bit codes $x_1 \ldots x_n$, where $|x_i| = \mathcal{O}(\log n)$, can be stored within $(\sum |x_i|)(1 + o(1))$ bits of space, so that we can (1) compute any sequence of codes $x_i, \ldots, x_j$, (2) update any code $x_i \leftarrow y$, (3) insert a new code z between any pair of codes, and (4) delete any code x_d from the sequence, all in $\mathcal{O}(\log n/\log\log n)$ time (plus $j - i$ for (1)). Moreover, let $f(x_i)$ be a nonnegative integer function computable in constant time from the codes. If the updates and indels are such that $|f(y) - f(x_i)|$, $f(z)$, and $f(x_d)$ are all $\mathcal{O}(\log n)$, then we can also support operations sum$(i) = \sum_{j=1}^{i} f(x_i)$ and search$(s) = \max\{i, \mathit{sum}(i) \le s\}$ within the same time.*

For example we can store n numbers $0 \le a_i < 2^k$ within $kn + o(kn)$ bits, by using their k-bit binary representation $[a_i]_2$ as the code, and their numeric value as $f([a_i]_2) = a_i$, so that we support *sum* and *search* on the sequence of numbers. If the numbers are very different in magnitude we can δ-encode them to achieve $(\sum \log a_i)(1 + o(1)) + \mathcal{O}(n)$ bits of space. We can also store bits, seen as 1-bit codes, in $n + o(n)$ bits and and carry out *sum* = *rank* and *search* = *select*, insertions and deletions, in $\mathcal{O}(\log n/\log\log n)$ time.

A further application of our results to the compressed representation of sequences achieves a result summarized in the next theorem.

THEOREM 1.5. *A sequence $S[0, n-1]$ over alphabet $[1, \sigma]$ can be stored in $nH_0(S) + \mathcal{O}(n\log\sigma/\log^\epsilon n + \sigma\log^\epsilon n)$ bits of space, for any constant $0 < \epsilon < 1$, and support the operations rank, select, insert, and delete, all in time $\mathcal{O}\left(\frac{\log n}{\log\log n}\left(1 + \frac{\log\sigma}{\log\log n}\right)\right)$.*

[4]This means that one can distinguish the first code x_i from a bit stream $x_i\alpha$ in constant time.

For polylogarithmic-sized alphabets, this is the optimal $\mathcal{O}(\log n/\log\log n)$*; otherwise it is* $O\left(\frac{\log n \log \sigma}{(\log\log n)^2}\right)$.

This time complexity slashes the the best current result [González and Navarro 2008] by a $\Theta(\log\log n)$ factor (the relation with results appeared after our conference publication [He and Munro 2010; Navarro and Nekrich 2012] will be discussed later). The optimality of the polylogarithmic case stems again from Fredman and Saks [1989] lower bound on *rank* on dynamic bitmaps. This result has immediate applications to building compressed indexes for text, building the Burrows-Wheeler transform within compressed space, and so on. The structure also allows us to attach and detach substrings, which is novel and may find various interesting applications.

1.2. Organization of the paper

In Section 2 we review basic data structures used in this paper. Section 3 describes 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 derive our new results on compressed sequences and applications. In Section 9 we conclude and give future work directions.

2. PRELIMINARIES

Here we describe balanced parenthesis sequences 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 of $c \in \{0, 1\}$ 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* [Jacobson 1989; Munro 1996; Raman et al. 2007; Pătraşcu 2008]. A basic one uses $n + o(n)$ bits and supports *rank*/*select* in constant time on the word RAM with word length $\Theta(\log n)$. 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 $nH_0(S) + \mathcal{O}(n \log\log n/\log n)$ bits, where $H_0(S) = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m} = m\log\frac{n}{m} + \mathcal{O}(m)$ is called the *empirical zero-order entropy* of the sequence. The space overhead on top of the entropy has been recently reduced [Pătraşcu 2008] to $\mathcal{O}(n\,t^t/\log^t n + n^{3/4})$ bits for any integer $t > 0$, while supporting *rank* and *select* in $\mathcal{O}(t)$ time. This can be built in linear worst-case time[5].

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 $\Theta(\log n)$.

The definition of *rank* and *select* on bitmaps generalizes to arbitrary sequences over an integer alphabet $[1, \sigma]$, as well as the definition of zero-order empirical entropy of sequences, to $H_0(S) = \sum_{1\le c\le\sigma}\frac{n_c}{n}\log\frac{n}{n_c}$, where c occurs n_c times in S. A compressed representation of general sequences that supports *rank*/*select* is achieved through a structure called a *wavelet tree* [Grossi et al. 2003; Navarro 2012]. This is a complete binary tree that partitions the

[5]They use a predecessor structure by Pătraşcu and Thorup [2006], more precisely their result achieving time "$\lg\frac{\ell-\lg n}{a}$", which is a simple modification of van Emde Boas' data structure.

alphabet $[1, \sigma]$ into contiguous halves at each node. The node then stores a bitmap telling which branch did each letter go. The tree has height $\lceil \log \sigma \rceil$, and it reduces *rank* and *select* operations to analogous operations on its bitmaps in a root-to-leaf or leaf-to-root traversal. If the bitmaps are represented within their zero-order entropy, the total space adds up to $nH_0(S)+o(n \log \sigma)$ and the operations are supported in $\mathcal{O}(\log \sigma)$ time. This can be improved to $\mathcal{O}(\lceil \frac{\log \sigma}{\log \log n} \rceil)$, while maintaining the same asymptotic space, by using a multiary wavelet tree of arity $\Theta(\sqrt{\log n})$, and replacing the bitmaps by sequences over small alphabets, which still can answer *rank*/*select* in constant time [Ferragina et al. 2007].

2.2. Succinct tree representations

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

There exist many succinct data structures for ordinal trees. Among them, the ones with maximum functionality [Farzan and Munro 2008] support all the operations in Table I, except *insert* and *delete*, in constant time using $2n + \mathcal{O}(n \log \log \log n / \log \log n)$-bit space. Our static data structure supports the same operations and reduces the space to $2n + \mathcal{O}(n/\text{polylog}(n))$ bits.

2.3. Dynamic succinct trees

We consider insertion and deletion of internal nodes or leaves in ordinal trees. In this setting, there exist no data structures supporting all the operations in Table I. The data structure of Raman and Rao [2003] 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. Farzan and Munro [2011] obtain constant time for simple updates and basic traversal operations (*parent*, *first_child*, *last_child*, *next_sibling*, *prev_sibling*) and show that obtaining the same for more sophisticated operations requires limiting the update patterns. Chan et al. [2007] gave a dynamic data structure using $\mathcal{O}(n)$ bits and supporting *findopen*, *findclose*, *enclose*, and updates, in $\mathcal{O}(\log n / \log \log n)$ time. They also gave another data structure using $\mathcal{O}(n)$ bits and supporting *findopen*, *findclose*, *enclose*, *lca*, *leaf_rank*, *leaf_select*, and updates, in $\mathcal{O}(\log n)$ time.

We consider a flexible node updating protocol (consistent with that of Chan et al. [2007]) that supports inserting a new leaf, a new tree root, or an internal node that will enclose a range of the consecutive children of some existing node. It also supports removing a leaf, the tree root (only if it has zero or one child), and an internal node, whose children will become children of its parent. On a BP representation, it is enough to allow inserting any pair of matching parentheses in the tree. On LOUDS and DFUDS, supporting this model may be more complex as it may require abrupt changes in node arities.

Furthermore, we consider the more sophisticated operation (which is simple on classical trees) of attaching a new subtree as a child of a node, instead of just a leaf. The model is that this new subtree is already represented with our data structures. Both trees are thereafter blended and become a single tree. Similarly, we can detach any subtree from a given tree so that it becomes an independent entity represented with our data structure. This allows for extremely flexible support of algorithms handling dynamic trees, far away from the limited operations allowed in previous work. This time we have to consider a maximum possible value for $\log n$ (say, w, the width of the system-wide pointers). Then we require $2n + \mathcal{O}(n \log w / w + \sqrt{2^w})$ bits of space and carry out the queries in time $\mathcal{O}(w / \log w)$ or $\mathcal{O}(w)$, depending on the tree structure we use. Insertions and deletions take $\mathcal{O}(w^{1+\epsilon})$ time for any constant $\epsilon > 0$ if we wish to allow attachment and detachment of subtrees, which then can also be carried out in time $\mathcal{O}(w^{1+\epsilon})$.

2.4. Dynamic compressed bitmaps and sequences

Let $B[0, n-1]$ be a bitmap. We want to support operations *rank* and *select* on B, as well as operations $\mathit{insert}(B, i, b)$, which inserts bit b between $B[i]$ and $B[i+1]$, and $\mathit{delete}(B, i)$, which deletes position $B[i]$ from B. Chan et al. [2007] handle all these operations in $\mathcal{O}(\log n / \log\log n)$ time (which is optimal [Fredman and Saks 1989]) using $\mathcal{O}(n)$ bits of space (actually, by reducing the problem to a particular dynamic tree). Mäkinen and Navarro [2008] achieve $\mathcal{O}(\log n)$ time and $nH_0(B) + \mathcal{O}(n \log\log n / \sqrt{\log n})$ bits of space. The results can be generalized to sequences. González and Navarro [2008] achieve $nH_0 + \mathcal{O}(n \log\sigma / \sqrt{\log n})$ bits of space and $\mathcal{O}(\log n (1 + \frac{\log\sigma}{\log\log n}))$ time to handle all the operations on a sequence over alphabet $[1, \sigma]$. They give several applications to managing dynamic text collections, construction of static compressed indexes within compressed space, and construction of the Burrows-Wheeler transform [Burrows and Wheeler 1994] within compressed space. We improve all these results in this paper, achieving the optimal $\mathcal{O}(\log n / \log\log n)$ on polylog-sized alphabets and reducing the lower-order term in the compressed space by a $\Theta(\log\log n)$ factor.

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[6] parentheses sequence by a 0,1 vector $P[0, n-1]$ ($P[i] \in \{0, 1\}$). Each opening/closing parenthesis is encoded by $\boxed{(} = 1$ and $\boxed{)} = 0$.

First, we remind that several operations of Table I either are trivial in a BP representation, or are easily solved using *enclose*, *findclose*, *findopen*, *rank*, and *select* [Munro and Raman 2001]. These are:

$$\begin{aligned}
\mathit{inspect}(i) &= P[i] \text{ (or } \mathit{rank}_1(P, i) - \mathit{rank}_1(P, i-1) \text{ if there is no access to } P[i] \\
\mathit{isleaf}(i) &= [P[i+1] = 0] \\
\mathit{isancestor}(i, j) &= i \le j \le \mathit{findclose}(P, i) \\
\mathit{depth}(i) &= \mathit{rank}_1(P, i) - \mathit{rank}_0(P, i) \\
\mathit{parent}(i) &= \mathit{enclose}(P, i) \\
\mathit{pre\_rank}(i) &= \mathit{rank}_1(P, i) \\
\mathit{pre\_select}(i) &= \mathit{select}_1(P, i) \\
\mathit{post\_rank}(i) &= \mathit{rank}_0(P, \mathit{findclose}(P, i)) \\
\mathit{post\_select}(i) &= \mathit{findopen}(\mathit{select}_0(P, i)) \\
\mathit{first\_child}(i) &= i + 1 \text{ (if } P[i+1] = 1\text{, else } i \text{ is a leaf)} \\
\mathit{last\_child}(i) &= \mathit{findopen}(P, \mathit{findclose}(P, i) - 1) \text{ (if } P[i+1] = 1\text{, else } i \text{ is a leaf)} \\
\mathit{next\_sibling}(i) &= \mathit{findclose}(i) + 1 \text{ (if } P[\mathit{findclose}(i) + 1] = 1\text{, else } i \text{ is last sibling)} \\
\mathit{prev\_sibling}(i) &= \mathit{findopen}(i - 1) \text{ (if } P[i-1] = 0\text{, else } i \text{ is the first sibling)} \\
\mathit{subtree\_size}(i) &= (\mathit{findclose}(i) - i + 1)/2
\end{aligned}$$

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.

[6]As later we will use these constructions to represent arbitrary segments of a balanced sequence.

Definition 3.1. For a 0,1 vector $P[0, n-1]$ and a function $g(\cdot)$ on $\{0,1\}$,

$$\begin{aligned}
\mathit{sum}(P,g,i,j) &\stackrel{\text{def}}{=} \sum_{k=i}^{j} g(P[k]) \\
\mathit{fwd\_search}(P,g,i,d) &\stackrel{\text{def}}{=} \min\{j \geq i \mid \mathit{sum}(P,g,i,j) = d\} \\
\mathit{bwd\_search}(P,g,i,d) &\stackrel{\text{def}}{=} \max\{j \leq i \mid \mathit{sum}(P,g,j,i) = d\} \\
\mathit{rmq}(P,g,i,j) &\stackrel{\text{def}}{=} \min\{\mathit{sum}(P,g,i,k) \mid i \leq k \leq j\} \\
\mathit{rmqi}(P,g,i,j) &\stackrel{\text{def}}{=} \operatorname*{argmin}_{i \leq k \leq j}\{\mathit{sum}(P,g,i,k)\} \\
\mathit{RMQ}(P,g,i,j) &\stackrel{\text{def}}{=} \max\{\mathit{sum}(P,g,i,k) \mid i \leq k \leq j\} \\
\mathit{RMQi}(P,g,i,j) &\stackrel{\text{def}}{=} \operatorname*{argmax}_{i \leq k \leq j}\{\mathit{sum}(P,g,i,k)\}
\end{aligned}$$

The following function is particularly important.

Definition 3.2. Let π 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 stored. Note that, given the form of π, 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_anc* has appeared as well [Munro and Rao 2004], but we give proofs for completeness.

LEMMA 3.3. *Let P be a BP sequence encoded by $\{0,1\}$. Then findclose, findopen, enclose, and level_anc can be expressed as follows.*

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

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, $\mathit{fwd\_search}(P, \pi, i, 0)$.

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)[7]. 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 $\mathit{bwd\_search}(P, \pi, i, 0)$.

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 $\mathit{sum}(P, \pi, j, i) = 2$.

[7]Note $E[j] - 1 = E[i]$ could hold at other places as well, where $P[j]$ is a closing parenthesis.

Finally, the proof for *level_anc* 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:

$$\begin{aligned}
\mathit{lca}(i,j) &= \max(i,j), \text{ if } \mathit{isancestor}(i,j) \text{ or } \mathit{isancestor}(j,i) \\
& \mathit{parent}(\mathit{rmqi}(P,\pi,i,j)+1), \text{ otherwise [Sadakane 2002]} \\
\mathit{deepest\_node}(i) &= \mathit{RMQi}(P,\pi,i,\mathit{findclose}(i)) \\
\mathit{height}(i) &= \mathit{depth}(\mathit{deepest\_node}(i)) - \mathit{depth}(i) \\
\mathit{level\_next}(i) &= \mathit{fwd\_search}(P,\pi,\mathit{findclose}(i),0) \\
\mathit{level\_prev}(i) &= \mathit{findopen}(\mathit{bwd\_search}(P,\pi,i,0)) \\
\mathit{level\_lmost}(d) &= \mathit{fwd\_search}(P,\pi,0,d) \\
\mathit{level\_rmost}(d) &= \mathit{findopen}(\mathit{bwd\_search}(P,\pi,n-1,-d))
\end{aligned}$$

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 ϕ, ψ be functions such that $\phi(0) = 0, \phi(1) = 1, \psi(0) = 1, \psi(1) = 0$. Then the following equalities hold.

LEMMA 3.4. *For a 0,1 vector P,*

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

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*, for the rest of the paper.

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 of prefix sums for the concatenation of all the ranges up to the subtree rooted at that node.

Definition 3.5. A *range min-max tree* for a bit 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 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 3.6. 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 POLYLOGARITHMIC-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 (not necessarily balanced) bit vectors of polylogarithmic size. This will be used to handle chunks of larger trees in later sections. Then we consider the particular case of balanced parentheses and complete all the operations on polylogarithmic-sized trees.

The general idea is that we will use a range min-max tree of constant height and sublogarithmic arity k. This arity will allow us encoding k accumulators (like e, m, and M) within a number of bits small enough to maintain a universal table with all the possible k-tuples

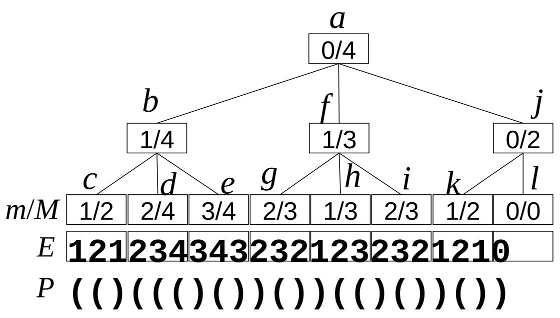

Fig. 2. An example of the range min-max tree using function π, and showing the m/M values.

of values. Hence any desired function on the nodes can be precomputed in constant time using universal tables.

Let $g(\cdot)$ be a function on $\{0,1\}$ taking values in $\{1,0,-1\}$. We call such a function ± 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 bit vector $P[0, n-1]$, of moderate size $n \leq N = w^c$. Assume we wish to solve the operations for an arbitrary ± 1 function $g(\cdot)$, and let $G[i]$ denote $sum(P, g, 0, i)$, 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 $\lceil n/s \rceil$ *chunks* of length s. These form the partition alluded in Definition 3.5: $\ell_i = s \cdot (i-1)$. Thus the values $m[i]$ and $M[i]$ correspond to minima and maxima of G within each chunk, and $e[i] = G[r_i]$.

Furthermore, the tree will be k-ary and complete, for $k = \Theta(w/(c \log w))$. Thus the leaves store all the elements of arrays m and M. 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 $\frac{k}{k-1} \cdot \frac{n}{s} \cdot \lceil \log(2w^c + 1) \rceil = \mathcal{O}(nc \log w/w)$ 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} is covered 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 nodes of T_{mM}.*

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 disjoint union $[i,j] = [i, r_a][\ell_{a+1}, r_{a+1}] \ldots [\ell_{b-1}, r_{b-1}][\ell_b, j]$ (we can discard the case $a = b$, as then 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 disjoint union 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 $\mathcal{O}(ck)$. $\square$

Example 4.2. 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.

4.1. Supporting forward and backward searches

Computing *fwd_search*(P, g, i, d) is done as follows (*bwd_search* is symmetric). First we check if the chunk of i, $[\ell_k, r_k]$ for $k = \lfloor i/s \rfloor$, contains *fwd_search*(P, g, i, d) with a table lookup using vector P, by precomputing a simple universal table of $2^s \log s = \mathcal{O}(\sqrt{2^w} \log w)$ bits[8]. If so, we are done. Else, we compute the global target value we seek, $d' = G[i-1] + d = e[k] - sum(P, g, i, r_k) + d$ (the sum inside the chunk is also computed 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 (note these are simply all the right siblings of the parent, all the right siblings of the grandparent, and so on, of the range min-max tree node holding the chunk $[\ell_k, r_k]$). 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 $m[u_j] \le d' \le M[u_j]$, then *fwd_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 within the chunk corresponding to the leaf by table lookups, using P again.

Example 4.3. 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 $E[j] = E[3-1] + 0 = 1$. 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 $\mathcal{O}(c)$ rather than $\mathcal{O}(ck)$.

According to Lemma 4.1, the $\mathcal{O}(ck)$ nodes can be partitioned into $\mathcal{O}(c)$ sequences of consecutive sibling nodes. We will manage to carry out the search within each such sequence in $\mathcal{O}(1)$ 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 consecutive 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 $M[u_j] \ge d'$. The next lemma proves it.

LEMMA 4.4. *Let $u_1, u_2, \ldots$ be a sequence of T_{mM} nodes containing consecutive intervals of P. If $g(\cdot)$ is a ± 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 ± 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 $d < M[u_j] + 1$,

[8]Using integer division and remainder, a segment within a chunk can be isolated and padded in constant time. If this is not allowed, the table must be slightly larger, $2^s s^2 \log s = \mathcal{O}(\sqrt{2^w} w^2 \log w)$ bits, which would not change our final results.

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 $d \geq m[u_j]$. The other case is symmetric. □

Thus the problem is reduced to finding the first $j > i$ such that $m[j] \leq d'$, among (at most) k consecutive sibling nodes (the case $M[j] \geq d'$ is symmetric). We build a universal table with all the possible sequences of k values $m[\cdot]$ and all possible $-w^c \leq d' \leq w^c$ values, and for each such sequence and d' we store the first j in the sequence such that $m[j] \leq d'$ (or we store a mark telling that there is no such position in the sequence). Thus the table has $(2w^c+1)^{k+1}$ entries, and $\log(k+1)$ bits per entry. By choosing the constant of $k = \Theta(w/(c \log w))$ so that $k \leq \frac{w}{2\log(2w^c+1)} - 1$, 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 consecutive siblings can be done in constant rather than $\mathcal{O}(k)$ time, and hence the overall time is $\mathcal{O}(c)$ rather than $\mathcal{O}(ck)$. Note that we store the $m'[\cdot]$ values in heap order, and therefore the k consecutive sibling values to input to the table are stored in contiguous memory, thus they can be accessed in constant time. We use an analogous universal table for $M[\cdot]$.

Finally, the process to solve $sum(P, g, i, j)$ in $\mathcal{O}(c)$ time is simple. We descend in the tree up to the leaf $[\ell_k, r_k]$ containing j. We obtain $sum(P, g, 0, \ell_k - 1) = e[k-1]$ 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 4.5. *In the RAM model with w-bit word size, for any constant $c \geq 1$ and a 0,1 vector P of length $n < w^c$, and a ± 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 $\mathcal{O}(c)$ time using the range min-max tree and universal lookup tables that require $\mathcal{O}(\sqrt{2^w}\log w)$ bits.*

4.2. Supporting range minimum queries

Next we consider how to compute $rmqi(P, g, i, j)$ and $RMQi(P, g, i, j)$. 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 covered by a disjoint union 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 $\mathcal{O}(\sqrt{2^w}\log w)$ bits. The minimum value of a subsequence $\mu_\ell, \ldots, \mu_r$ which shares the same parent in the range min-max tree can also be found by table lookups. The size of such universal table is $\mathcal{O}((2w^c+1)^k \log k) = \mathcal{O}(\sqrt{2^w})$ bits[9]. Hence we find the node containing the minimum value μ among $\mu_1, \mu_2, \ldots$, in $\mathcal{O}(c)$ time. If there is a tie, we choose the leftmost one.

If μ 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 constant 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 universal table for leaves. The overall time complexity is $\mathcal{O}(c)$. We have proved the following lemma.

LEMMA 4.6. *In the RAM model with w-bit word size, for any constant $c \geq 1$ and a 0,1 vector P of length $n < w^c$, and a ± 1 function $g(\cdot)$, rmqi(P, g, i, j) and RMQi(P, g, i, j) can*

[9]Again, we can isolate subranges using integer division and remainder; otherwise we need $\mathcal{O}((2w^c+1)^k k^2 \log k) = \mathcal{O}(\sqrt{2^w} w^2 \log w)$ bits, which would not affect our final result.

be computed in $\mathcal{O}(c)$ time using the range min-max tree and universal lookup tables that require $\mathcal{O}(\sqrt{2^w})$ bits.

4.3. 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 $g = \pi$, and assume P is the BP representation of a tree.

It is obvious how to compute *degree*(i), *child*(i, q) and *child_rank*(i) in time proportional to the degree of the node. To compute them in constant time, we add another array $n'[\cdot]$ to the data structure. In the range min-max tree, each node stores (in array $m'[\cdot]$) the minimum value of the subrange of the node. Now we also store, in $n'[\cdot]$, the multiplicity of the minimum value of each subrange.

LEMMA 4.7. *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 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. $\square$

Now we can compute *degree*(i) in constant time. Let $d = \mathit{depth}(i)$ and $j = \mathit{findclose}(i)$. 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 $\mathcal{O}(\sqrt{2^w} \log w)$ bits. The time complexity is $\mathcal{O}(c)$ if we use universal tables that let us process sequences of (up to) k consecutive children at once, that is, telling the minimum $m[\cdot]$ value within the sequence and the number of times it appears (adding up $n[\cdot]$ values). As the result depends on $m[\cdot]$ and $n[\cdot]$ values, this table requires $\mathcal{O}((2w^c + 1)^{2k} \cdot 2 \log(2w^c + 1))$ bits. This is $\mathcal{O}(\sqrt{2^w} \log w)$ if we limit $k = \frac{w}{4 \log(2w^c+1)} - 1$ (indeed the final "-1" is needed for operation *child* in the next paragraph).

Operation *child_rank*(i) can be computed similarly, by counting the number of minima in $E[\mathit{parent}(i), i - 1]$. 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 $\mathcal{O}(c)$ time. Here we need another universal table that tells at which position the number of occurrences of the minimum value exceeds some threshold, which requires $\mathcal{O}((2w^c + 1)^{2k+1} \log k) = \mathcal{O}(\sqrt{2^w} \log w)$ bits.

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_rank*$(i) = \mathit{rank}_1(P_1, i)$ and *leaf_select*$(i) = \mathit{select}_1(P_1, i)$ hold. The other operations are computed by *lmost_leaf*$(i) = \mathit{select}_1(P_1, \mathit{rank}_1(P_1, i - 1) + 1)$ and *rmost_leaf*$(i) = \mathit{select}_1(P_1, \mathit{rank}_1(P_1, \mathit{findclose}(i)))$. Note that we need not store P_1 explicitly; it 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 vector.

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

Definition 4.8. (From Sadakane [2007a].) The inorder rank of an internal node v is defined as the number of visited internal nodes, including v, in a 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 $q-1$ inorder ranks, so leaves and unary nodes have no inorder rank. We define $\mathit{in\_rank}(i)$ as the smallest inorder rank of internal node i, and $\mathit{in\_select}(j)$ as the (internal) the tree node whose (smallest) inorder rank is j.

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 rank of an internal node.

LEMMA 4.9. *(From Sadakane [2007a].) Let i be an internal node, and let $j = \mathit{in\_rank}(i)$, so $i = \mathit{in\_select}(j)$. Then*

$$\begin{aligned}\mathit{in\_rank}(i) &= \mathit{rank}_1(P_2, \mathit{findclose}(P, i+1)) \\ \mathit{in\_select}(j) &= \mathit{enclose}(P, \mathit{select}_1(P_2, j) + 1)\end{aligned}$$

Note that $\mathit{in\_select}(j)$ will return the same node i for any of its $\mathit{degree}(i) - 1$ inorder ranks.

Once again, we need not store P_2 explicitly.

4.4. 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'_ϕ, e'_ψ, m'_ϕ, m'_ψ, M'_ϕ, and M'_ψ which maintain the corresponding values for functions ϕ and ψ (recall Lemma 3.4). However, note that $\mathit{sum}(P, \phi, 0, i)$ and $\mathit{sum}(P, \psi, 0, i)$ are nondecreasing, thus the minimum/maximum within the chunk is just the value of the sum at the beginning/end of the chunk. Moreover, as $\mathit{sum}(P, \pi, 0, i) = \mathit{sum}(P, \phi, 0, i) - \mathit{sum}(P, \psi, 0, i)$ and $\mathit{sum}(P, \phi, 0, i) + \mathit{sum}(P, \psi, 0, i) = i$, 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 internal nodes. Moreover, any sequence of k consecutive such values can be obtained, via table lookup, from the sequence of k 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 ϕ and ψ.

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 [2008]. 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 φ values of the children and of the subtree size, for internal nodes. The construction is able to decode the B values of φ 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. Next we adapt their result to our case.

LEMMA 4.10. *(Adapted from Pătraşcu [2008, Thm. 8].) Let $|\Phi| = (2B+1)^{4c}$, and B be such that $(B+1)\log(2B+1) \leq \frac{w}{8c}$ (thus $B = \Theta(\frac{w}{c \log w})$). An aB-tree over bit vector $P[0, N-1]$, where $N = B^c$, with values in Φ, 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 $\mathcal{O}(\sqrt{2^w}\mathrm{poly}(w))$ for the universal tables.*

The "$+w^{3/2}$" construction time comes from a fusion tree [Fredman and Willard 1993] that is used internally on $\mathcal{O}(w)$ values. It could be reduced to w^ϵ time for any constant $\epsilon > 0$ and navigation time $\mathcal{O}(1/\epsilon)$, but we prefer to set $c > 3/2$ so that $N = B^c$ dominates it.

These parameters still allow us to represent our range min-max trees while yielding the complexities we had found, as $k = \Theta(w/(c \log w))$ 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 our universal tables and those of Lemma 4.10 (our universal tables become smaller with the reduction from w and s to B). 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.

THEOREM 4.11. *On a w-bit word RAM, for any constant $c > 3/2$, we can represent a sequence P of $N = B^c$ parentheses describing a tree in BP form, for sufficiently small $B = \Theta(\frac{w}{c \log w})$, computing all operations of Table I in $\mathcal{O}(c)$ time, with a data structure depending on P that uses $N + 2$ bits, and universal tables (i.e., not depending on P) that use $\mathcal{O}(\sqrt{2^w})$ bits. The preprocessing time is $\mathcal{O}(N + \sqrt{2^w}\mathrm{poly}(w))$ (the second term is needed only once for universal tables) and its working space is $\mathcal{O}(N)$ bits.*

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. The main idea is that queries that cannot be solved inside a block can make use of more classical data structures, as long as these are built on $\mathcal{O}(n/w^c)$ elements, so their space will be negligible. Some of those structures are known, yet some new ones are introduced.

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 π

We consider extending *fwd_search*(P, π, i, d) and *bwd_search*(P, π, 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, π, 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 $d' = e_{j-1} + \mathit{sum}(P, \pi, 0, i - 1 - w^c \cdot (j-1)) + d$ (where the *sum* is local to block j) in the following blocks. Then the answer lies in the first block $r > j$ such that $m_r \le d' \le M_r$. Thus, we can

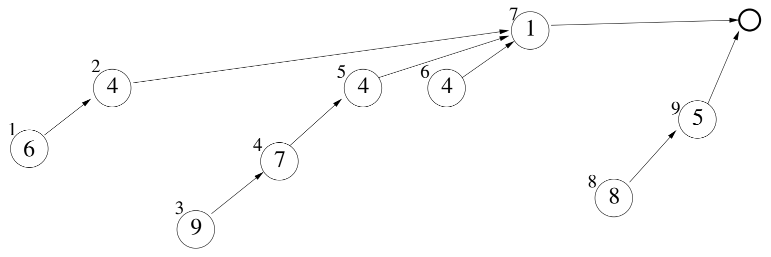

Fig. 3. A tree representing the $lrm(j)$ sequences of values $m_1 \dots m_9$.

apply again Lemma 4.4, 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 $m_r \leq d'$, or such that $M_r \geq d'$. 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 τ. Let us focus on left-to-right minima, as the others are similar.

Definition 5.1. Let $m_1, m_2, \dots, m_\tau$ be a sequence of integers. We define for each $1 \leq j \leq \tau$ the *left-to-right minima starting at* j as $lrm(j) = \langle j_0, j_1, j_2, \dots \rangle$, where $j_0 = j$, $j_r < j_{r+1}$, $m_{j_{r+1}} < m_{j_r}$, and $m_{j_r+1}, \dots, m_{j_{r+1}-1} \geq m_{j_r}$.

The following lemmas are immediate.

LEMMA 5.2. *The first element $\leq x$ after position j in a sequence of integers $m_1, m_2, \dots, m_\tau$, where $lrm(j) = \langle j_0, j_1, j_2, \dots \rangle$, is m_{j_r} for some $r > 0$.*

LEMMA 5.3. *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 5.3 is essential to store all the τ 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 τ 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 5.3. We add a fictitious root to convert the forest into a tree. Note that this structure was independently discovered[10] by Fischer [2010], who called it “2d-Min-Heap” and showed how to build it in linear time.

Example 5.4. Figure 3 illustrates the tree built from the sequence $\langle m_1 \dots m_9 \rangle = \langle 6, 4, 9, 7, 4, 4, 1, 8, 5 \rangle$. 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 5.5. *For a tree with τ 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 an $\Omega(\log(\tau W))$-bit word RAM, for any $\epsilon > 0$ and integer $t > 0$. 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.*

[10]Published in the same year of the conference version of our paper [Sadakane and Navarro 2010].

PROOF. We use a variant of Bender and Farach-Colton [2004] $\langle \mathcal{O}(\tau \log \tau), \mathcal{O}(1) \rangle$ algorithm. 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 ℓ, is extended upwards, adding other ℓ 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 partitioning guarantees that a node of height h has also height h in its path, and thus at least its first h ancestors are in its ladder. Moreover the union of all ladders has at most 2τ 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 $\mathcal{O}(\tau \log \tau)$-words space and time). To solve the query $level\_anc(v, d)$, the ancestor v' at distance $d' = 2^{\lfloor \log d \rfloor}$ from v is obtained. 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 ℓ a sparse bitmap of length at most ℓW, where 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 2τ 1s, and because weights do not exceed W, at most $2\tau W$ 0s. Using the sparse bitmaps of Pătraşcu [2008], 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 support *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 [Fredman and Willard 1993]. These can store z keys of ℓ 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 (unweighted) 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 in $\mathcal{O}(t)$ further time 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 $\epsilon = \min(1/2, 1/c)$, the query time is $\mathcal{O}(c + t)$ and the preprocessing time is $\mathcal{O}(n)$ for $c > 3/2$.

5.2. Other operations

For computing *rmqi* and *RMQi*, we use a simple data structure [Bender and Farach-Colton 2000] on the m_r and M_r values, later improved to require only $\mathcal{O}(\tau)$ bits on top of the sequence of values [Sadakane 2002; Fischer and Heun 2007]. 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 covering the two extremes of the boundary $[i, j]$.

For computing *child*, *child_rank*, and *degree*, only some parenthesis pairs must be cared about. We consider all pairs (i, j) of matching parentheses ($j = findclose(i)$) such that i and j belong to different blocks. If we define a graph whose vertices are blocks and the edges are the pairs of parentheses considered, the graph is outer-planar since the parenthesis pairs nest [Jacobson 1989], yet there are multiple edges among nodes. To remove these, we choose the tightest pair of parentheses for each pair of vertices. These parenthesis pairs are called *pioneers*. Since they correspond to edges of a planar graph, the number of pioneers is $\mathcal{O}(n/w^c)$.

For the three queries, it is enough to consider only nodes that completely include a block (otherwise the query is solved in constant time by considering at most two adjacent blocks; we can easily identify such nodes using *findclose*). Furthermore, among them, it is enough to consider pioneers: Assume the parentheses pair (i, i') contains a whole block but it is not a pioneer. Then there exists a pioneer pair (j, j') contained in (i, i') where j is in the same block of i and j' is in the same block of i'. Thus the block contained in (i, i') contains no children of (i, i') as all its parentheses descend from (j, j'). Moreover, all the children of (i, i') start either in the block of i or in the block of i', since (j, j') or an ancestor of it is a child of (i, i'). So again the operations are solved in constant time by considering two blocks. Such cases can be identified by doing *findclose* on the last child of i starting in its block and seeing if that child closes in the block of i'.

Let us call *marked* the nodes to consider (that is, pioneers that contain a whole block). There are $\mathcal{O}(n/w^c)$ marked nodes, thus for *degree* we can simply store the degrees of marked nodes using $\mathcal{O}(\frac{n \log n}{w^c})$ bits of space, and the others are computed in constant time as explained.

For *child* and *child_rank*, we set up a bitmap $C[0, 2n-1]$ where marked nodes v are indicated with $C[v] = 1$, and preprocess C for *rank* queries so that satellite information can be associated to marked nodes. Using again the result of Pătraşcu [2008], vector C can be represented in at most $\frac{2n}{w^c}\log(w^c) + \mathcal{O}(\frac{n\, t^t}{\log^t n} + n^{3/4})$ bits, so that any bit $C[i]$ can be read, and any *rank* on C can be computed, in $\mathcal{O}(t)$ time.

We will focus on children of marked nodes placed at the blocks fully contained in those nodes, as the others are inside the two extreme blocks and can be dealt with in constant time. Note that a block fully contained in some marked nodes can contain children of at most one of them (i.e., the innermost).

For each marked node v we store a list formed by the blocks fully contained in v, and the marked nodes children of v, in left-to-right order of P. The blocks store the number of children of v that start within them, and the children marked nodes store simply a 1 (indicating they contain 1 child of v). Both store also their position inside the list. The length of all the sequences adds up to $\mathcal{O}(n/w^c)$ because each block and marked node appears in at most one list. Their total sum of children is at most n, for the same reason. Thus, it is easy to store all the number of children as gaps between consecutive 1s in a bitmap, which can be stored within the same space bounds of the other bitmaps in this section ($\mathcal{O}(n)$ bits, $\mathcal{O}(n/w^c)$ 1s).

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 $p = rank_1(C_v, select_0(C_v, q))$ on the bitmap C_v of v. This tells the position in the list of blocks and marked nodes of v where the q-th child of v lies. If it is a marked node, then that node is the child. If instead it is a block v', then the answer corresponds to the q'-th minimum within that block, where $q' = q - rank_0(select_1(C_v, p))$. (Recall that we first have to see if $child(v, q)$ lies in the block of v or in that of $findclose(v)$, using a within-block query in those cases, and otherwise subtracting from q the children that start in the block of v.)

For $child\_rank(u)$, we can directly store the answers for marked blocks u. Else, it might be that $v = parent(u)$ starts in the same block of u or that $findclose(v)$ is in the same block of $findclose(u)$, in which case we solve $child\_rank(u)$ with an in-block query and the help of $degree(v)$. Otherwise, the block where u belongs must be in the list of v, say at position p_u. Then the answer is $rank_0(C_v, select_1(C_v, p_u))$ plus the number of minima in the block of u until $u - 1$.

Finally, the remaining operations require just *rank* and *select* on P, or the virtual bit vectors P_1 and P_2. For *rank* it is enough to store the answers at the end of blocks, and finish the query within a single block. For $\mathit{select}_1(P, i)$ (and similarly for select_0 and for P_1 and P_2), we make up a sequence with the accumulated number of 1s in each of the τ blocks. The numbers add up to $2n$ and thus can be represented as gaps of 0s between consecutive 1s in a bitmap $S[0, 2n-1]$, which can be stored within the previous space bounds. Computing $x = \mathit{rank}_1(S, \mathit{select}_0(S, i))$, in time $\mathcal{O}(t)$, lets us know that we must finish the query in block x, querying its range min-max tree with the value $i' = \mathit{select}_0(S, i) - \mathit{select}_1(S, x)$.

5.3. The final result

Recall from Theorem 4.11 that we actually use blocks of size B^c, not w^c, for $B = \mathcal{O}(\frac{w}{c \log w})$. The sum of the space for all the block 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 these results, the overall extra space added is $\mathcal{O}(\frac{n(c \log B + \log^2 n)}{B^c} + \frac{n\, t^t}{\log^t n} + \sqrt{2^w} + n^{3/4})$ bits. Using a computer word of $w = \log n$ bits, setting $t = c$, and expanding $B = \mathcal{O}(\frac{\log n}{c \log\log n})$, we get that the time for any operation is $\mathcal{O}(c)$ and the total space simplifies to $2n + \mathcal{O}(\frac{n(c \log\log n)^c}{\log^{c-2} n})$.

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 4.11. Next we compute marked nodes: 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. At this point it is very easy to spot marked nodes. 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. Once the marked nodes are identified, the compressed representation of [Pătraşcu 2008] for 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 allow inserting or deleting a pair of matching parentheses, which supports the node update operations described in Section 2.3.

The main idea in this section is to use one single range min-max tree for the whole parentheses sequence, and implement it as a dynamic balanced binary tree. We first deal with some memory allocation technicalities to avoid wasting $\Theta(n)$ bits at the leaf buckets, and then go on to describe the dynamic range min-max tree.

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 ℓ 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 [Munro 1986]. 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 each s_i uses an exclusive set of cells to store all the segments of length i. Those segments are packed consecutively, without wasting any extra space in the cells of linked list s_i (except possibly at the head cell of each list). 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.

If a segment is moved, 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 $2n + \mathcal{O}(\frac{n \log n}{L} + L^2)$ 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.

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 universal tables. Moving a segment to another list is also done in $\mathcal{O}(L/\log n)$ time.

6.2. A dynamic tree

We give a simple dynamic data structure that represents an ordinal tree with n nodes using $2n + \mathcal{O}(n/\log n)$ bits, and supports all the query and update operations of Table I 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 v of the tree corresponds to a range $P[i, j]$, the node stores $i(v) = i$ and $j(v) = j$, as well as $e(v) = sum(P, \pi, i, j)$, $m(v) = rmq(P, \pi, i, j)$, $M(v) = RMQ(P, \pi, i, j)$, and $n(v)$, the number of times $m(v)$ occurs in $P[i, j]$.

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 I. Note that the values we store are now 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 compute *depth*(i) starting at the min-max tree root v with children v_l and v_r, we first see if $j(v_l) \geq i$, in which case we continue at v_l, otherwise we obtain the result from v_r and add $e(v_l)$ to it. As another example, to solve *fwd_search*(P, π, i, d), we first convert d into the global value $d' \leftarrow depth(i) + d$, and then 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, looking for global depth $d' - e(v_l)$. This will only traverse $\mathcal{O}(\log n)$ nodes, as seen in Section 4.

Since 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 $\mathcal{O}(\frac{n \log n}{L} + L^2)$ 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 $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}(n))$ extra bits, which is negligible.

If an insertion/deletion occurs, we update a segment, and the stored values in the leaf for the segment. From the leaf we step back to the root, updating the values as follows:

$$i(v), j(v) = i(v_l), j(v_r)$$

$$\begin{aligned} e(v) &= e(v_l) + e(v_r) \\ m(v) &= \min(m(v_l), e(v_l) + m(v_r)) \\ M(v) &= \max(M(v_l), e(v_l) + M(v_r)) \\ n(v) &= n(v_l), \text{ if } m(v_l) < e(v_l) + m(v_r), \\ & \quad n(v_r), \text{ if } m(v_l) > e(v_l) + m(v_r), \\ & \quad n(v_l) + n(v_r), \text{ otherwise.} \end{aligned}$$

If the length of the segment exceeds $2L$, we split it into two and add a new node. If, instead, 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, as explained. If a balancing operation (i.e., a rotation) occurs, we also update the values in the involved nodes. All these updates are carried out in constant time per involved node, as their values are recomputed using the formulas above. 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 [2008] have shown how to do this for a very similar case (dynamic *rank*/*select* on a bitmap). Their solution of splitting the bitmap into three parts and moving border bits across parts to deamortize the work applies verbatim to our sequence of parentheses, 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 technicality we ignore). We have one range min-max tree for each of the three parts and adapt all the algorithms in the obvious way[11].

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 done by Chan et al. [2007]. Then the depth of the tree is $\mathcal{O}(\log n / \log \log n)$, and we can expect to reduce the time complexities to this order. The challenge is to support the operations within nodes, that handle $\Theta(\sqrt{\log n})$ values, in constant time. We will achieve it for some operations, while others will stay $\mathcal{O}(\log n)$ time.

To reduce the time complexities, we must slightly decrease the lengths of the segments, which range from L to $2L$, setting $L = \log^2 n / \log \log n$. Now each leaf can be processed in time $\mathcal{O}(\log n / \log \log n)$ using universal tables. This reduction affects the required space for the range min-max tree and the parenthesis vector, which 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 $\mathcal{O}(\frac{n}{L\sqrt{\log n}})$ of them).

Each internal node v of the range min-max tree has k children, for $\sqrt{\log n} \le k \le 2\sqrt{\log n}$ in principle (we will relax the constants later). 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 ℓ_i, (ii) $s_\phi[1,k]$ and $s_\psi[1,k]$ storing $s_{\phi/\psi}[i] = \mathit{sum}(P, \phi/\psi, \ell_1, r_i)$, (iii) $e[1,k]$ storing $e[i] = \mathit{sum}(P, \pi, \ell_1, r_i)$, (iv) $m[1,k]$ storing $m[i] = e[i-1] + \mathit{rmq}(P, \pi, \ell_i, r_i)$, $M[1,k]$ storing $M[i] = e[i-1] + \mathit{RMQ}(P, \pi, \ell_i, r_i)$, and (v) $n[1,k]$ storing in $n[i]$ the number of times $m[i]$ occurs within the subtree of the ith child. Note that the values stored are local to the subtree (as in the simpler balanced binary tree version, Section 6) but cumulative with respect to previous siblings. Note also that storing s_ϕ, s_ψ and e is redundant, as noted in Section 4.4,

[11]One can act as if one had a single range min-max tree where the first two levels were used to split the three parts (these first nodes would be special in the sense that their handling of insertions/deletions would reflect the actions on moving bits between the three parts).

but now we need $s_{\phi/\psi}$ in explicit form to achieve constant-time searching into their values, as it will be clear soon.

Apart from simple accesses to the stored values, we need to support the following operations within any node:

— $p(i)$: the largest $j \geq 2$ such that $\ell_{j-1} \leq i$ (or 1 of there is no such j).
— $w_{\phi/\psi}(i)$: the largest $j \geq 2$ such that $s_{\phi/\psi}[j-1] \leq i$ (or 1 if there is no such j).
— $f(i,d)$: the smallest $j \geq i$ such that $m[j] \leq d \leq M[j]$.
— $b(i,d)$: the largest $j \leq i$ such that $m[j] \leq d \leq M[j]$.
— $r(i,j)$: the smallest x such that $m[x]$ is minimum in $m[i,j]$.
— $R(i,j)$: the smallest x such that $M[x]$ is maximum in $M[i,j]$.
— $n(i,j)$: the number of times the minimum within the subtrees of children i to j occurs within that range.
— $r(i,j,t)$: the x such that the t-th minimum within the subtrees of children i to j occurs within the x-th child.
— *update*: updates the data structure upon ± 1 changes in some child.

Simple operations involving *rank* and *select* on P are carried out easily with $\mathcal{O}(\log n/\log\log n)$ applications of $p(i)$ and $w_{\phi/\psi}(i)$. For example *depth*(i) is computed, starting from the root node, by finding the child $j = p(i)$ to descend, then recursively computing *depth*$(i-\ell_j)$ on the j-th child, and finally adding $e[j-1]$ to the result. Handling ϕ for P_1 and P_2 is immediate; we omit it.

Operations *fwd_search*/*bwd_search* can 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 consecutive siblings. Within each such sequence a single $f(i,d)/b(i,d)$ operation is sufficient. For example, for *fwd_search*(i,d), let us assume d is a global excess to find (i.e., start with $d \leftarrow d + \mathit{depth}(i) - 1$). We start at the root v of the range min-max tree, and compute $j = p(i)$, so the search starts at the j-th child, with the recursive query *fwd_search*$(i-\ell_j, d-e[j-1])$. If the answer is not found in that child, query $j' = f(j+1,d)$ tells that it is within child j'. We then enter recursively into the j'-th child of the node with *fwd_search*$(i-\ell_{j'}, d-e[j'-1])$, where the answer is sure to be found.

Operations *rmqi* and *RMQi* are solved similarly, using $\mathcal{O}(\log n/\log\log n)$ applications of $r(i,j)/R(i,j)$. For example, to compute *rmq*(i,i') (the extension to *rmqi* is obvious) we start with $j = p(i)$ and $j' = p(i')$. If $j = j'$ we answer with $e[j-1] + \mathit{rmq}(i-\ell_j, i'-\ell_j)$ on the j-th child of the current node. Otherwise we recursively compute $e[j-1]+\mathit{rmq}(i-\ell_j, \ell_{j+1}-\ell_j-1)$, $e[j'-1] + \mathit{rmq}(0, i'-\ell_{j'})$ and, if $j+1 < j'$, $m[r(j+1, j'-1)]$, and return the minimum of the two or three values.

For *degree* we partition the interval as for *rmqi* and then use $m[r(i,j)]$ 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 range. Thus we apply $r(i,j)$ and $n(i,j)$ $\mathcal{O}(\log n/\log\log n)$ times. Operation *child_rank* is very similar, by redefining 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 it holds $n(i,j) \geq t$, 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 on the child with $t \leftarrow t - n(i, r(i,j,t)-1)$, using one $r(i,j,t)$ operation per level.

7.1. Dynamic partial sums

Let us now face the problem of implementing the basic operations. Our first tool is a result by Raman et al. [2001], which solves several subproblems of the same type.

LEMMA 7.1. *(From Raman et al. [2001].) Under the RAM model with word size* $\Theta(\log n)$, *it is possible to maintain a sequence of* $\log^{\epsilon} n$ *nonnegative integers* $x_1, x_2, \ldots$ *of* $\log n$ *bits* *each, for any constant $0 \leq \epsilon < 1$, such that the data structure requires $\mathcal{O}(\log^{1+\epsilon} n)$ bits and carries out the following operations in constant time: sum$(i) = \sum_{j=1}^{i} x_j$, search$(s) = \max\{i,\ \text{sum}(i) \leq s\}$, and update$(i, \delta)$, which sets $x_i \leftarrow x_i + \delta$, for $-\log n \leq \delta \leq \log n$. The data structure also uses a precomputed universal table of size $\mathcal{O}(n^{\epsilon'})$ bits for any fixed $\epsilon' > 0$. The structure can be built in $\mathcal{O}(\log^{\epsilon} n)$ time except the table.*

Then we can store ℓ, s_ϕ, and s_ψ in differential form, and obtain their values via *sum*. Operations p and $w_{\phi/\psi}$ are then solved via *search* on ℓ and $s_{\phi/\psi}$, respectively This also eliminates the problem of a change in one subtree propagating to the accumulators of its right siblings, and thus we can handle ± 1 changes in the subtrees in constant time. The same can be done with e, provided we fix the fact that it can contain negative values by storing $e[i] + 2^{\lceil \log n \rceil}$ (this works for constant-time *sum*, yet not for *search*). In addition, we can store $M[i] - e[i-1] + 1 \geq 0$ and $e[i-1] - m[i] + 1 \geq 0$, which depend only on the subtree, and reconstruct any $M[i]$ or $m[i]$ in constant time using *sum* on e, which eliminates the problem of propagating changes in $e[i]$ to $m[i+1, k]$ and $M[i+1, k]$. Local changes to $m[i]$ or $M[i]$ can be applied directly.

7.2. Cartesian trees

Our second tool is the Cartesian tree [Vuillemin 1980; Sadakane 2007b]. A Cartesian tree for an array $B[1, k]$ is a binary tree in which the root node stores the minimum value $B[\mu]$, and the left and the right subtrees are Cartesian trees for $B[1, \mu-1]$ and $B[\mu+1, k]$, respectively. If there exist more than one minimum value position, then μ is the leftmost. Thus the tree shape has enough information to determine the position of the leftmost minimum in any range $[i, j]$. As it is a binary tree of k nodes, a Cartesian tree can be represented within $2k$ bits using parentheses and the bijection with general trees. It can be built in $\mathcal{O}(k)$ time.

We build Cartesian trees for $m[1, k]$ and for $M[1, k]$ (this one taking maxima). Since $2k = \mathcal{O}(\sqrt{\log n})$, universal tables let us answer in constant time any query of the form $r(i, j)$ and $R(i, j)$, as these depend only on the tree shape, as explained. All the universal tables we will use on Cartesian trees take $\mathcal{O}(2^{\mathcal{O}(\sqrt{\log n})} \cdot \text{polylog}(n)) = o(n^{\alpha})$ bits for any constant $0 < \alpha < 1$.

We also use Cartesian trees to solve operations $f(i, d)$ and $b(i, d)$. However, these do not depend only on the tree shape, but on the actual values $m[i, k]$. We focus on $f(i, d)$ since $b(i, d)$ is symmetric. Following Lemma 4.4, we first check whether $m[i] \leq d \leq M[i]$, in which case the answer is i. Otherwise, the answer is either the next j such that $m[j] \leq d$ (if $d < m[i]$), or $M[j] \geq d$ (if $d > M[i]$). Let us focus on the case $d < m[i]$, as the other is symmetric. By Lemma 5.2, the answer belongs to $lrm(i)$, where the sequence is $m[1, k]$.

The next lemma shows that any lrm sequence can be deduced from the topology of the Cartesian tree[12].

LEMMA 7.2. *Let C be the Cartesian tree for $m[1, k]$. Then $lrm(i)$ is the sequence of nodes of C in the upward path from i to the root, which are reached from the left child.*

PROOF. The left and right children of node i contain values not smaller than i. All the nodes in the upward path are equal to or smaller than i. Those reached from the right must be at the left of position i. Their left children are also to the left of i. Ancestors j reached from the left are strictly smaller than i and appear to the right of i. Furthermore, they are smaller than any other element in their right subtree, which includes all positions from i to j, thus they belong to $lrm(i)$. Finally, the right descendants of those ancestors j are not in $lrm(i)$ because they appear after j and equal to or larger than $m[j]$. □

[12]This is not surprising if we consider that the lrm and the Cartesian trees are related by the usual isomorphism to convert general into binary trees [Davoodi et al. 2012].

The Cartesian tree can have precomputed $lrm(i)$ for each i, as this depends only on the tree shape, and thus are stored in universal tables. This is the sequence of positions in $m[1,k]$ that must be considered. We can then binary search this sequence, using the technique described to retrieve any desired $m[j]$, to compute $f(i,d)$ in $\mathcal{O}(\log k) = \mathcal{O}(\log\log n)$ time.

7.3. Complete trees

We arrange a complete binary tree on top of the $n[1,k]$ values, so that each node of the tree records (i) one leaf where the subtree minimum is attained, and (ii) the number of times the minimum arises in its subtree. This tree is arranged in heap order and requires $\mathcal{O}(\log^{3/2} n)$ bits of space.

A query $n(i,j)$ is answered essentially as in Section 6: We find the $\mathcal{O}(\log k)$ nodes that cover $[i,j]$, find the minimum $m[\cdot]$ value among the leaves stored in (i) for each covering node (recall we have constant-time access to m), and add up the number of times (field (ii)) the minimum of $m[i,j]$ occurs. This takes overall $\mathcal{O}(\log k)$ time.

A query $r(i,j,t)$ is answered similarly, stopping at the node where the left-to-right sum of the fields (ii) reaches t, and then going down to the leaf x where t is reached. Then the t-th occurrence of the minimum in subtrees i to j occurs within the x-th subtree.

When an $m[i]$ or $n[i]$ value changes, we must update the upward path towards the root of the complete tree, using the update formula for $n(v)$ given in Section 6. This is also sufficient when $e[i]$ changes: Although this implicitly changes all the $m[i+1,k]$ values, the local subtree data outside the ancestors of i are unaffected. Finally, the root $n(v)$ value will become an $n[i']$ value at the parent of the current range min-max tree node (just as the minimum of $m[1,k]$, maximum of $M[1,k]$, excess $e[k]$, etc., which can be computed in constant time as we have seen).

Since these operations take time $\mathcal{O}(\log k) = \mathcal{O}(\log\log n)$ time, the time complexity of *degree*, *child*, and *child_rank* is $\mathcal{O}(\log n)$. Update operations (*insert* and *delete*) also require $\mathcal{O}(\log n)$ time, as we may need to update $n[\cdot]$ for one node per tree level. However, as we see later, it is possible to achieve time complexity $\mathcal{O}(\log n/\log\log n)$ for *insert* and *delete* for all the other operations. Therefore, we might choose not to support operations $n(i,j)$ and $r(i,j,t)$ to retain the lower update complexity. In this case, operations *degree*, *child*, and *child_rank* can only be implemented naively using *first_child*, *next_sibling*, and *parent*.

7.4. Updating Cartesian trees

We have already solved some simple aspects of updating, but not yet how to maintain Cartesian trees. When a value $m[i]$ or $M[i]$ changes (by ± 1), the Cartesian trees might change their shape. Similarly, a ± 1 change in $e[i]$ induces a change in the effective value of $m[i+1,k]$ and $M[i+1,k]$. We store m and M in a way independent of previous e values, but the Cartesian trees are built upon the actual values of m and M. Let us focus on m, as M is similar. If $m[i]$ decreases by 1, we need to determine if i should go higher in the Cartesian tree. We compare i with its Cartesian tree parent $j = Cparent(i)$ and, if (a) $i < j$ and $m[i] - m[j] = 0$, or if (b) $i > j$ and $m[i] - m[j] = -1$, we must carry out a rotation with i and j. Figure 4 shows the two cases. As it can be noticed, case (b) may propagate the rotations towards the new parent of i, as $m[i]$ is smaller than the previous root $m[j]$.

In order to carry out those propagations in constant time, we store an array $d[1,k]$, so that $d[i] = m[i] - m[Cparent(i)]$ if this is $\leq k+1$, and $k+2$ otherwise. Since $d[1,k]$ requires $\mathcal{O}(k\log k) = \mathcal{O}(\sqrt{\log n}\,\log\log n) = o(\log n)$ bits of space, it can be manipulated in constant time using universal tables: With $d[1,k]$ and the current Cartesian tree as input, a universal table can precompute the outcome of the changes in $d[\cdot]$ and the corresponding sequence of rotations triggered by the decrease of $m[i]$ for any i, so we can obtain in constant time the new Cartesian tree and the new table $d[1,k]$. The limitation of values up to $k+2$ is

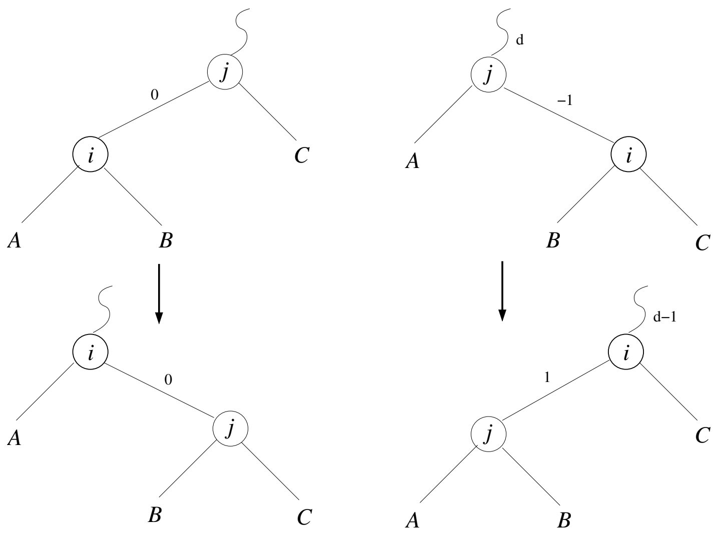

Fig. 4. Rotations between i and its parent j when $m[i]$ decreases by 1. The edges between any x and its parent are labeled with $d[x] = m[x] - m[Cparent(x)]$, if these change during the rotation. The $d[\cdot]$ values have already been updated. On the left, when $i < j$, on the right, when $i > j$.

necessary for the table fitting in a machine word, and its consequences will be discussed soon.

Similarly, if $m[i]$ increases by 1, we must compare i with its two children: (a) the difference with its left child must not fall below 1 and (b) the difference with its right child must not fall below 0. Otherwise we must carry out rotations as well, depicted in Figure 5. While it might seem that case (b) can propagate rotations upwards (due to the $d-1$ at the root), this is not the case because d had just been increased as $m[i]$ grew by 1. In case both (a) and (b) arise simultaneously, we must apply the rotation corresponding to (b) and then that of (a). No further propagation occurs. Again, universal tables can precompute all these updates.

For changes in $e[i]$, the universal tables have precomputed the effect of carrying out all the changes in $m[i+1,k]$, updating all the necessary $d[1,k]$ values and the Cartesian tree. This is equivalent to precomputing the effect of a sequence of $k-i$ successive changes in $m[\cdot]$.

Our array $d[1,k]$ distinguishes values between 0 and $k+2$. As the changes to the structure of the Cartesian tree only depend on whether $d[i]$ is 0, 1, or larger than 1, and all the updates to $d[i]$ are by ± 1 per operation, we have sufficient information in $d[\cdot]$ to correctly predict any change in the Cartesian tree shape for the next k updates. We refresh table $d[\cdot]$ fast enough to ensure that no value of $d[\cdot]$ is used for more than k updates without recomputing it, as then its imprecision could cause a flaw. We simply recompute cyclically the cells of $d[\cdot]$, one per update. That is, at the i-th update arriving at the node, we recompute the cell $i' = 1 + (i \bmod k)$, setting again $d[i'] = \min(k+2, m[i'] - m[Cparent(i')])$; note that $Cparent(i')$ is computed from the Cartesian tree shape in constant time via table lookup. Note the values of $m[\cdot]$ are always up to date because we do not keep them in explicit form but with $e[i-1]$ subtracted (and in turn e is not maintained explicitly but via partial sums).

7.5. Handling splits and merges

In case of splits or merges of segments or internal range min-max tree nodes, we must insert or delete children in a node. To maintain the range min-max tree dynamically, we use a data structure by Fleischer [1996]. This is an $(a, 2b)$-tree (for $a \le 2b$) storing n numeric keys in

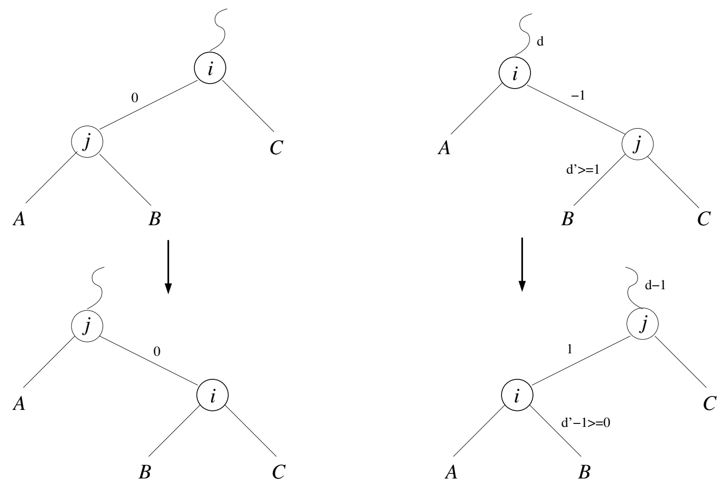

Fig. 5. Rotations between i and its children when $m[i]$ increases by 1. The edges between any x and its parent are labeled with $d[x] = m[x] - m[Cparent(x)]$, if these change during the rotation. The $d[\cdot]$ values have already been updated. On the left (right), when the edge to the left (right) child becomes invalid after the change in $d[\cdot]$.

the leaves, and each leaf is a bucket storing at most $2 \log_a n$ keys. It supports constant-time insertion and deletion of a key once its location in a leaf is known.

Each leaf owns a *cursor*, which is a pointer to a tree node. This cursor traverses the tree upwards, looking for nodes that should be split, moving one step per insertion received at the leaf. When the cursor reaches the root, the leaf has received at most $\log_a n$ insertions and thus it is split into two new leaves. These leaves are born with their cursor at their common parent. In addition some edges must be marked. Marks are considered when splitting nodes (see Fleischer [1996] for details). The insertion steps are as follows:

(1) Insert the new key into the leaf B. Let v be the current node where the cursor of B points.
(2) If v has more than b children, split it into v_1 and v_2, and unmark all edges leaving from those nodes. If the parent of v has more than b children, mark the edges to v_1 and v_2.
(3) If v is not the root, set the cursor to the parent of v. Otherwise, split B into two halves, and let the cursor of both new buckets point to their common parent.

To apply this to our data structure, let $a = \sqrt{\log n}, b = 2\sqrt{\log n}$. Then the arity is in the range $1 \le k \le 4\sqrt{\log n}$, the height of the tree is $\mathcal{O}(\log n / \log\log n)$, and each leaf should store $\Theta(\log n / \log\log n)$ keys. Instead our structure stores $\Theta(\log^2 n / \log\log n)$ bits in each leaf. If Fleischer's structure handles $\mathcal{O}(\log n)$-bit numbers, it turns out that the leaf size is the same in both cases. The difference is that our insertions are bitwise, whereas Fleischer's insertions are number-wise (that is, in packets of $\mathcal{O}(\log n)$ bits). Therefore we can use the same structure, yet the cursor will return to the leaf $\Theta(\log n)$ times more frequently than necessary. Thus we only split a leaf when the cursor returns to it *and* it has actually exceeded size $2L$. This means leaves can actually reach size $L' = 2L + \mathcal{O}(\log n / \log\log n) = 2L(1 + \mathcal{O}(1/\log n))$, which is not a problem. Marking/unmarking of children edges is easily handled in constant time by storing a bit-vector of length $2b$ in each node.

Fleischer's update time is constant. Ours is $\mathcal{O}(\sqrt{\log n})$ because, if we split a node into two, we fully reconstruct all the values in those two nodes and their parent. This can be done in $\mathcal{O}(k) = \mathcal{O}(\sqrt{\log n})$ time, as the structure of Lemma 7.1, the Cartesian trees, and the complete trees can be built in linear time. Nevertheless this time is dominated by the $\mathcal{O}(\log n/\log\log n)$ cost of inserting a bit at the leaf.

Deletion of nodes may make the parent node arity fall below a. This is handled as in Fleischer's structure, by deamortized global rebuilding. This increases only the sublinear size of the range min-max tree; the leaves are not affected.

Deletions at leaves, however, are handled as before, ensuring that they have always between L and L' bits. This may cause some abrupt growth in their length. The most extreme case arises when merging an underflowing leaf of $L-1$ bits with its sibling of length L. In this case the result is of size $2L-1$, close to overflowing, it cannot be split, and the cursor may be far from the root. This is not a problem, however, as we have still sufficient time before the merged leaf reaches size L'.

7.6. The final result

We have obtained the following result.

LEMMA 7.3. *For a 0,1 vector of length $2n$ representing a tree in BP form, 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 updates and all other queries except degree, child, and child_rank, in $\mathcal{O}(\log n/\log\log n)$ time. Alternatively, degree, child, child_rank, and updates can be handled in $\mathcal{O}(\log n)$ time.*

The complexity of *fwd_search* and *bwd_search* is not completely satisfactory, as we have reduced many operators to those. To achieve better complexities, we note that most operators that reduce to *fwd_search* and *bwd_search* actually reduce to the less general operations *findclose*, *findopen*, and *enclose* on parentheses. Those three operations can be supported in time $\mathcal{O}(\log n/\log\log n)$ by adapting the technique of Chan et al. [2007]. They use a tree of similar layout as ours: leaves storing $\Theta(\log^2 n/\log\log n)$ parentheses and internal nodes of arity $k = \Theta(\sqrt{\log n})$, where Lemma 7.1 is used to store seven arrays of numbers recording information on matched and unmatched parentheses on the children. Those are updated in constant time upon parenthesis insertions and deletions, and are sufficient to support the three operations. They report $\mathcal{O}(n)$ bits of space because they do not use a mechanism like the one we describe in Section 6.1 for the leaves; otherwise their space would be $2n + \mathcal{O}(n\log\log n/\log n)$ as well. Note, on the other hand, that they do not achieve the times we offer for the important *lca* and related operations.

This completes the main result of this section, Theorem 1.2.

7.7. Updating whole subtrees

We consider now attaching and detaching whole subtrees. Now we assume $\log n$ is fixed to some sufficiently large value, for example $\log n = w$, the width of the systemwide pointers. Hence, no matter the size of the trees, they use segments of the same length, and the times are a function of w and not of the actual tree size.

Now we cannot use Fleischer [1996] data structure, because a detached subtree could have dangling cursors pointing to the larger tree it belonged. As a result, the time complexity for insert or delete changes to $\mathcal{O}(\sqrt{\log n}\cdot\log n/\log\log n) = \mathcal{O}(\log^{3/2} n/\log\log n)$. To improve it, we change the degree of the nodes in the range min-max tree from $\Theta(\sqrt{\log n})$ to $\Theta(\log^{\epsilon} n)$ for any fixed constant $\epsilon > 0$. This makes the complexity of *insert* and *delete* $\mathcal{O}(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n) = \mathcal{O}(\log^{1+\epsilon} n)$, and multiplies all query time complexities by the constant $\mathcal{O}(1/\epsilon)$.

First we consider attaching a tree T^1 to another tree T^2, that is, the root of T^1 becomes a child of a node of T^2, which can be a leaf or already have other children. Let $P^1[0, 2n_1-1]$ and $P^2[0, 2n_2-1]$ be the BP sequences of T^1 and T^2, respectively. Then this attaching operation corresponds to creating a new BP sequence $P' = P^2[0, p]P^1[0, 2n_1-1]P^2[p+1, 2n_2-1]$ for some position p (so p is the position of the new parent of T^1 if it will become a first child, or otherwise it is the position of the closing parenthesis of the new previous sibling of T^1).

If p and $p+1$ belong to the same segment, we cut the segment into two, say $P_l = P^2[l,p]$ and $P_r = P^2[p+1,r]$. If the length of P_l (P_r) is less than L, we concatenate it to the left (right) segment of it. If its length exceeds $2L$, we split it into two. We also update the upward paths from P_l and P_r to the root of the range min-max tree for T^2 to reflect the changes done at the leaves. Those changes are not by ± 1, so accumulators must be rebuilt from scracth, in time $O(\frac{1}{\epsilon}\log^{\epsilon} n)$.

Now we merge the range min-max trees for T^1 and T^2 as follows. Let h_1 be the height of the range min-max tree of T^1, and h_2 be the height of the *lca*, say v, between P_l and P_r in the range min-max tree of T^2. If $h_2 > h_1$ then can simply concatenate the root of T^1 at the right of the ancestor of P_l of height h_1, then split the node if it has overflowed, and finish.

If $h_2 \leq h_1$, we divide v into v_l and v_r, so that the rightmost child of v_l is an ancestor of P_l and the leftmost child of v_r is an ancestor of P_r. We do not yet care about v_l or v_r being too small. We repeat the process on the parent of v until reaching the height $h_2 = h_1 + 1$. Let us call u the ancestor where this height is reached (we leave for later the case where we split the root of T^2 without reaching the height $h_1 + 1$).

Now we add T^1 as a child of u, between the child of u that is an ancestor of P_l and the child of u that is an ancestor of P_r. All the leaves have the same depth, but the ancestors of P_l and of P_r at heights h_2 to h_1 might be underfull as we have cut them arbitrarily. We glue the ancestor of height h of P_l with the leftmost node of height h of T^1, and that of P_r with the rightmost node of T^1, for all $h_2 \leq h \leq h_1$. Now there are no underfull nodes, but they can have overflowed. We verify the node sizes in both paths, from height $h = h_2$ to $h_1 + 1$, splitting them as necessary. At height h_2 the node can be split into two, adding another child to its parent, which can thus be split into three, adding in turn two children to its parent, but from there on nodes can only be split into three and add two more children to their parent. Hence the overall process of fixing arities takes time $\mathcal{O}(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n)$.

If node u does not exist, then T^1 is not shorter than T^2. In this case we have divided T^2 into a left and right parts, of heights h_l and h_r, respectively. We attach the left part of T^2 to the leftmost node of height h_l in T^1, and the right part of T^2 to the rightmost node of height h_r in T^1. Then we fix arities in both paths, similarly as before.

Detaching is analogous. After splitting the leftmost and rightmost leaves of the area to be detached, let P_l and P_r be the leaves of T preceding and following the leaves that will be detached. We split the ancestors of P_l and P_r until reaching their *lca*, let it be v. Then we can form a new tree with the detached part and remove it from the original tree T. Again, the paths from P_l and P_r to v may contain underfull nodes. But now P_l and P_r are consecutive leaves, so we can merge their ancestor paths up to v and then split as necessary.

Similarly, the leftmost and rightmost path of the detached tree may contain underfull nodes. We merge each node of the leftmost (rightmost) path with its right (left) sibling, and then split if necessary. The root may contain as few as two children. Overall the process takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

7.8. Using DFUDS representation

The structure described in this section can be used to maintain any balanced parentheses sequence, supporting operations *rank*, *select*, *findopen*, *findclose*, *enclose*, *insert* and *delete* in time $\mathcal{O}(\log n/\log\log n)$ and using $2n + \mathcal{O}(n\log\log n/\log n)$ bits. If we use it to encode the DFUDS representation [Benoit et al. 2005] of a tree, a number of the operations of Table I can be translated into those basic operations [Jansson et al. 2012]. In particular, operations *degree*, *child*, and *child_rank* can be supported within this time complexity. On the other hand, operations related to depth are not directly supported.

Insertion of a new leaf as the qth child of a node i is carried out by inserting a new $\boxed{)}$ at position *child*(i,q) and then a $\boxed{(}$ at position i. Insertion of a new internal node encompassing the qth to q'th children of i requires inserting $q'-q+1$ symbols $\boxed{(}$ followed by a $\boxed{)}$ at position *child*(i,q), and then removing $q'-q$ symbols $\boxed{(}$ at position i. This can be done in time $\mathcal{O}(\log n/\log\log n)$ if $q'-q = \mathcal{O}(\log^2 n/\log\log n)$. Deletion is analogous. We then have the following result.

LEMMA 7.4. *On a $\Theta(\log n)$-bit word RAM, all the operations on a dynamic ordinal tree with n nodes given in Table I, except depth, level_anc, level_next, level_prev, level_lmost, level_rmost, deepest_node, post_rank, post_select and height, can be carried out within time $\mathcal{O}(\log n/\log\log n)$, using a data structure that requires $2n + \mathcal{O}(n\log\log n/\log n)$ bits. Insertions and deletions are limited to nodes of arity $\mathcal{O}(\log^2 n/\log\log n)$.*

8. IMPROVING DYNAMIC COMPRESSED SEQUENCES

The techniques we have developed along the paper are of independent interest. We illustrate this point by improving the best current results on sequences of numbers with *sum* and *search* operations, dynamic compressed bitmaps, and their many byproducts.

8.1. Codes, numbers, and partial sums

We prove now Lemma 1.4 on sequences of codes and partial sums, this way improving previous results by Mäkinen and Navarro [2008] and matching lower bounds [Pătraşcu and Demaine 2006]. The idea is to concatenate all the codes x_i, and store information in the internal nodes of the tree on cumulative code lengths, to locate any desired x_i, and cumulative $f(\cdot)$ values, to support *sum* and *search*.

Section 7 shows how to maintain a dynamic bitmap P supporting various operations in time $\mathcal{O}(\log n/\log\log n)$, including insertion and deletion of bits (parentheses in P). This bitmap P will now be the concatenation of the (possibly variable-length) codes x_i. We will ensure that each leaf contains a sequence of whole codes (no code is split at a leaf boundary). As these are of $\mathcal{O}(\log n)$ bits, we only need to slightly adjust the lower limit L to enforce this: After splitting a leaf of length $2L$, one of the two new leaves might be of size $L - \mathcal{O}(\log n)$.

We process a leaf by chunks of $b = \frac{1}{2}\log n$ bits: A universal table (easily computable in $\mathcal{O}(\sqrt{n}\,\mathrm{polylog}(n))$ time and space) can tell us how many whole codes are there in the next b bits, how much their $f(\cdot)$ values add up to, and where the last complete code ends (assuming we start reading at a code boundary). Note that the first code could be longer than b, in which case the table lets us advance zero positions. In this case we decode the next code directly. Thus in constant time (at most two table accesses plus one direct decoding) we advance in the traversal by at least b bits. If we surpass the desired position with the table we reprocess the last $\mathcal{O}(\log n)$ codes using a second table that advances by chunks of $\mathcal{O}(\sqrt{\log n})$ bits, and finally process the last $\mathcal{O}(\sqrt{\log n})$ codes directly. Thus in time $\mathcal{O}(\log n/\log\log n)$ we can access a given code in a leaf (and subsequent ones in constant time each), sum the $f(\cdot)$ values up to some position, and find the position where a given sum s is exceeded. We can also easily modify a code or insert/delete codes, by shifting all the other codes of the leaf in time $\mathcal{O}(\log n/\log\log n)$.

In internal nodes of the range min-max tree we use the structure of Lemma 7.1 to maintain the number of codes stored below the subtree of each child of the node. This allows determining in constant time the child to follow when looking for any code x_i, thus access to any codes $x_i \ldots x_j$ is supported in time $\mathcal{O}(\log n/\log\log n + j - i)$.

When a code is inserted/deleted at a leaf, we must increment/decrement the number of codes in the subtree of the ancestors up to the root; this is supported in constant time by Lemma 7.1. Splits and merges can be caused by indels and by updates. They force the recomputation of their whole parent node, and Fleischer's technique is used to ensure a constant number of splits/merges per update. Note that we are inserting not individual bits but whole codes of $\mathcal{O}(\log n)$ bits. This can easily be done, but now $\mathcal{O}(\log n/\log\log n)$ insertions/updates can double the size of a leaf, and thus we must consider splitting the leaf every time the cursor returns to it (as in the original Fleischer's proposal, not every $\log n$ times as when inserting parentheses), and we must advance the cursor upon insertions *and* updates.

Also, we must allow leaves of sizes between L and $L' = 3L$ (but still split them as soon as they exceed $2L$ bits). In this way, after a merge produces a leaf of size $2L-1$, we still have time to carry out $L = \mathcal{O}(\log n/\log\log n)$ further insertions before the cursor reaches the root and splits the leaf. (Recall that if the merge produces a leaf larger than $2L$ we can immediately split it, so $2L$ is the worst case we must handle.)

For supporting *sum* and *search* we also maintain at each node the sum of the $f(\cdot)$ values of the codes stored in the subtree of each child. Then we can determine in constant time the child to follow for *search*, and the sum of previous subtrees for *sum*. However, insertions, deletions and updates must alter the upward sums only by $\mathcal{O}(\log n)$ units, so that the change can be supported by Lemma 7.1 within the internal nodes in constant time.

8.2. Dynamic bitmaps

Apart from its general interest, handling a dynamic bitmap in compressed form is useful for maintaining satellite data for a sample of the tree nodes. A dynamic bitmap B could mark which nodes are sampled, so if the sampling is sparse enough we would like B to be compressed. A *rank* on this bitmap would give the position in a dynamic array where the satellite information for the sampled nodes is stored. This bitmap would be accessed by preorder (*pre_rank*) on the dynamic tree. That is, node v is sampled iff $B[\mathit{pre\_rank}(v)] = 1$, and if so, its data is at position $\mathit{rank}_1(B, \mathit{pre\_rank}(v))$ in the dynamic array of satellite data. When a tree node is inserted or deleted, we need to insert/delete its corresponding bit in B.

In the following we prove the next lemma, which improves and indeed simplifies previous results [Chan et al. 2007; Mäkinen and Navarro 2008]; then we explore several byproducts[13].

LEMMA 8.1. *We can store a bitmap $B[0, n-1]$ in $nH_0(B) + \mathcal{O}(n\log\log n/\log n)$ bits of space, while supporting the operations rank, select, insert, and delete, all in time $\mathcal{O}(\log n/\log\log n)$. We can also support attachment and detachment of contiguous bitmaps within time $\mathcal{O}(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$, yet now $\log n$ is a maximum fixed value across all the operations.*

To achieve zero-order entropy space, we use the (c, o) encoding of Raman et al. [2007]: The bits are grouped into small chunks of $b = \frac{\log n}{2}$ bits, and each chunk is represented by two components: the *class* c_i, which is the number of bits set, and the *offset* o_i, which is an identifier of that chunk within those of the same class. While the $|c_i|$ lengths add up to $\mathcal{O}(n\log\log n/\log n)$ extra bits, the $|o_i| = \left\lceil \log \binom{b}{c_i} \right\rceil$ components add up to $nH_0(B) + \mathcal{O}(n/\log n)$ bits [Pagh 2001].

We plan to store whole chunks in leaves of the range min-max tree. A problem is that the insertion or even deletion of a single bit in Raman et al.'s representation can up to double the size of the compressed representation of the segment, because it can change all the alignments. This occurs for example when moving from $0^b\ 1^b\ 0^b\ 1^b \ldots$ to $10^{b-1}\ 01^{b-1}\ 10^{b-1}\ 01^{b-1} \ldots$, where we switch from all $c_i = 0$ or b, and $|o_i| = 0$, to all

[13]Very recently, He and Munro [2010] achieved a similar result (excluding attachment and detachment of sequences), independently and with a different technique. Their space redundancy, however, is $\mathcal{O}(n\log\log n/\sqrt{\log n})$, that is, $\Theta(\sqrt{\log n})$ times larger than ours.

$c_i = 1$ or $b-1$, and $|o_i| = \lceil \log b \rceil$. This problem can be dealt with (laboriously) on binary trees [Mäkinen and Navarro 2008; González and Navarro 2008], but not on our k-ary tree, because Fleischer's scheme does not allow leaves being partitioned often enough.

We propose a different solution that ensures that an insertion cannot make the leaf's physical size grow by more than $\mathcal{O}(\log n)$ bits. Instead of using the same b value for all the chunks, we allow any $1 \le b_i \le b$. Thus each chunk is represented by a triple (b_i, c_i, o_i), where o_i is the offset of this chunk among those of length b_i having c_i bits set. To ensure $\mathcal{O}(n \log\log n / \log n)$ space overhead over the entropy, we state the invariant that any two consecutive chunks i, $i+1$ must satisfy $b_i + b_{i+1} > b$. Thus there are $\mathcal{O}(n/b)$ chunks and the overhead of the b_i and c_i components, representing each with $\lceil \log(b+1) \rceil$ bits, is $\mathcal{O}(n \log b / b)$. It is also easy to see that the inequality [Pagh 2001] $\sum |o_i| = \sum \lceil \log \binom{b_i}{c_i} \rceil = \log \Pi \binom{b_i}{c_i} + \mathcal{O}(n/\log n) \le \log \binom{n}{m} + \mathcal{O}(n/\log n) = nH_0(B) + \mathcal{O}(n/\log n)$ holds, where m is the number of 1s in the bitmap.

To maintain the invariant, the insertion of a bit is processed as follows. We first identify the chunk (b_i, c_i, o_i) where the bit must be inserted, and compute its new description (b'_i, c'_i, o'_i). If $b'_i > b$, we split the chunk into two, (b_l, c_l, o_l) and (b_r, c_r, o_r), for $b_l, b_r = b'_i/2 \pm 1$. Now we check left and right neighbors $(b_{i-1}, c_{i-1}, o_{i-1})$ and $(b_{i+1}, c_{i+1}, o_{i+1})$ to ensure the invariant on consecutive chunks holds. If $b_{i-1} + b_l \le b$ we merge these two chunks, and if $b_r + b_{i+1} \le b$ we merge these two as well. Merging is done in constant time by obtaining the plain bitmaps, concatenating them, and reencoding them, using universal tables (which we must have for all $1 \le b_i \le b$). Deletion of a bit is analogous; we remove the bit and then consider the conditions $b_{i-1} + b'_i \le b$ and $b'_i + b_{i+1} \le b$. It is easy to see that no insertion/deletion can increase the encoding by more than $\mathcal{O}(\log n)$ bits.

Now let us consider *codes* $x_i = (b_i, c_i, o_i)$. These are clearly constant-time self-delimiting and $|x_i| = \mathcal{O}(\log n)$, so we can directly use Lemma 1.4 to store them in a range min-max tree within $n' + \mathcal{O}(n' \log\log n' / \log n')$ bits, where $n' = nH_0(B) + \mathcal{O}(n \log\log n / \log n)$ is the size of our compressed representation. Since $n' \le n + \mathcal{O}(n \log\log n / \log n)$, we have $\mathcal{O}(n' \log\log n' / \log n') = \mathcal{O}(n \log\log n / \log n)$ and the overall space is as promised in the lemma. We must only take care of checking the invariant on consecutive chunks when merging leaves, which takes constant time.

Now we use the *sum*/*search* capabilities of Lemma 1.4. Let $f_b(b_i, c_i, o_i) = b_i$ and $f_c(b_i, c_i, o_i) = c_i$. As both are always $\mathcal{O}(\log n)$, we can have *sum*/*search* support on them. With *search* on f_b we can reach the code containing the jth bit of the original sequence, which is key for accessing an arbitrary bit. For supporting *rank* we need to descend using *search* on f_b, and accumulate the *sum* on the f_c values of the left siblings as we descend. For supporting *select* we descend using *search* on f_c, and accumulate the *sum* on the f_b values. Finally, for insertions and deletions of bits we first access the proper position, and then implement the operation via a constant number of updates, insertions, and deletions of codes (for updating, splitting, and merging our triplets). Thus we implement all the operations within time $\mathcal{O}(\log n / \log\log n)$.

We can also support attachment and detachment of contiguous bitmaps, by applying essentially the same techniques developed in Section 7.7. We can have a bitmap $B'[0, n'-1]$ and insert it between $B[i]$ and $B[i+1]$, or we can detach any $B[i, j]$ from B and convert it into a separate bitmap that can be handled independently. The complications that arise when cutting the compressed segments at arbitrary positions are easily handled by splitting codes. Zero-order compression is retained as it is due to the sum of the local entropies of the chunks, which are preserved (small resulting segments after the splits are merged as usual).

8.3. Sequences and text indices

We now aim at maintaining a sequence $S[0, n-1]$ of symbols over an alphabet $[1, \sigma]$, so that we can insert and delete symbols, and also compute symbol $\mathit{rank}_c(S, i)$ and $\mathit{select}_c(S, i)$, for $1 \leq c \leq \sigma$. This has in particular applications to labeled trees: We can store the sequence S of the labels of a tree in preorder, so that $S[\mathit{pre\_rank}(i)]$ is the label of node i. Insertions and deletions of nodes must be accompanied with insertions and deletions of their labels at the corresponding preorder positions, and this can be extended to attaching and detaching subtrees. Then we not only have easy access to the label of each node, but can also use *rank* and *select* on S to find the r-th descendant node labeled c, or compute the number of descendants labeled c. If the balanced parentheses represent the tree in DFUDS format [Benoit et al. 2005], we can instead find the first child of a node labeled c using *select*.

We divide the sequence into chunks of maximum size $b = \frac{1}{2}\log_\sigma n$ symbols and store them using an extension of the (c_i, o_i) encoding for sequences [Ferragina et al. 2007]. Here $c_i = (c_i^1, \ldots, c_i^\sigma)$, where c_i^a is the number of occurrences of character a in the chunk. For this code to be of length $\mathcal{O}(\log n)$ we need $\sigma = \mathcal{O}(\log n / \log\log n)$; more stringent conditions will arise later. To this code we add the b_i component as in Section 8.2. This takes $nH_0(S) + \mathcal{O}(\frac{n\sigma \log\log n}{\log n})$ bits of space. In the range min-max tree nodes, which we again assume to hold $\Theta(\log^\epsilon n)$ children for some constant $0 < \epsilon < 1$, instead of a single f_c function as in Section 8.2, we must store one f_a function for each $a \in [1,\sigma]$, requiring extra space $\mathcal{O}(\frac{n\sigma \log\log n}{\log n})$. Symbol *rank* and *select* are easily carried out by considering the proper f_a function. Insertion and deletion of symbol a is carried out in the compressed sequence as before, and only f_b and f_a sums must be incremented/decremented along the path to the root.

In case a leaf node splits or merges, we must rebuild the partial sums for all the σ functions f_a (and the single function f_b) of a node, which requires $\mathcal{O}(\sigma \log^\epsilon n)$ time. In Section 7.5 we have shown how to limit the number of splits/merges to one per operation, thus we can handle all the operations within $\mathcal{O}(\log n / \log\log n)$ time as long as $\sigma = \mathcal{O}(\log^{1-\epsilon} n / \log\log n)$. This, again, greatly simplifies the solution by González and Navarro [2008], who used a collection of partial sums with indels.

Up to here, the result is useful for small alphabets only. González and Navarro [2008] handle larger alphabets by using a multiary wavelet tree (Section 2.1). Recall this is a complete r-ary tree of height $h = \lceil \log_r \sigma \rceil$ that stores a string over alphabet $[1,r]$ at each node. It solves all the operations (including insertions and deletions) by h applications of the analogous operation on the sequences over alphabet $[1,r]$.

Now we set $r = \log^{1-\epsilon} n / \log\log n$, and use the small-alphabet solution to handle the sequences stored at the wavelet tree nodes. The height of the wavelet tree is $h = \mathcal{O}\left(1 + \frac{\log \sigma}{(1-\epsilon)\log\log n}\right)$. The zero-order entropies of the small-alphabet sequences add up to that of the original sequence and the redundancies add up to $\mathcal{O}\left(\frac{n \log \sigma}{(1-\epsilon)\log^\epsilon n \log\log n}\right)$. The operations require $\mathcal{O}\left(\frac{\log n}{\log\log n}\left(1 + \frac{\log \sigma}{(1-\epsilon)\log\log n}\right)\right)$ time. By slightly altering ϵ we obtain Theorem 1.5, where term $\mathcal{O}(\sigma \log^\epsilon n)$ owes to representing the wavelet tree itself, which has $\mathcal{O}(\sigma/r)$ nodes.

The arity of the nodes fixed to $\Theta(\log^\epsilon n)$ allows us attach and detach substrings in time $\mathcal{O}(r \log^{1+\epsilon} n)$ on a sequence with alphabet size r. This has to be carried out on each of the $\mathcal{O}(\sigma/r)$ wavelet tree nodes, reaching overall complexity $\mathcal{O}(\sigma \log^{1+\epsilon} n)$.

The theorem has immediate application to the handling of compressed dynamic text collections, construction of compressed static text collections within compressed space, and construction of the Burrows-Wheeler transform (BWT) within compressed space. We state them here for completeness; for their derivation refer to the original articles [Mäkinen and Navarro 2008; González and Navarro 2008].

The first result refers to maintaining a collection of texts within high-entropy space, so that one can perform searches and also insert and delete texts. Here H_h refers to the h-th order empirical entropy of a sequence, see e.g. Manzini [2001]. We use a sampling step of $\log_\sigma n \log\log n$ to achieve it.

COROLLARY 8.2. *There exists a data structure for handling a collection $\mathcal{C}$ of texts over an alphabet $[1,\sigma]$ within size $nH_h(\mathcal{C}) + o(n\log\sigma) + \mathcal{O}(\sigma^{h+1}\log n + m\log n + w)$ bits, simultaneously for all h. Here n is the length of the concatenation of m texts, $\mathcal{C} = 0\ T_1 0\ T_2 \cdots 0\ T_m$, and we assume that $\sigma = o(n)$ is the alphabet size and $w = \Omega(\log n)$ is the machine word size under the RAM model. The structure supports counting of the occurrences of a pattern P in $\mathcal{O}(|P|\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$ time, and inserting and deleting a text T in $\mathcal{O}(\log n + |T|\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$ time. After counting, any occurrence can be located in time $\mathcal{O}(\frac{\log^2 n}{\log\log n}(1+\frac{\log\log n}{\log\sigma}))$. Any substring of length ℓ from any T in the collection can be displayed in time $\mathcal{O}(\frac{\log^2 n}{\log\log n}(1+\frac{\log\log n}{\log\sigma}) + \ell\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$. For $h \le (\alpha\log_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space complexity simplifies to $nH_h(\mathcal{C}) + o(n\log\sigma) + \mathcal{O}(m\log n + w)$ bits.*

The second result refers to the construction of the most succinct self-index for text within the same asymptotic space required by the final structure. This is tightly related to the construction of the BWT, which has many applications.

COROLLARY 8.3. *The Alphabet-Friendly FM-index [Ferragina et al. 2007], as well as the BWT [Burrows and Wheeler 1994], of a text $T[0, n-1]$ over an alphabet of size σ, can be built using $nH_h(T) + o(n\log\sigma)$ bits, simultaneously for all $h \le (\alpha\log_\sigma n) - 1$ and any constant $0 < \alpha < 1$, in time $\mathcal{O}(n\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$.*

On polylog-sized alphabets, we build the BWT in $o(n\log n)$ time. Even on a large alphabet $\sigma = \Theta(n)$, we build the BWT in $o(n\log^2 n)$ time. This slashes by a $\log\log n$ factor the corresponding previous result [González and Navarro 2008]. Other previous results that focus in using little space are as follows. Okanohara and Sadakane [2009] achieved optimal $\mathcal{O}(n)$ construction time with $\mathcal{O}(n\log\sigma\log\log_\sigma n)$ bits of extra space (apart from the $n\log\sigma$ bits of the sequence). Hon et al. [2009] achieve $\mathcal{O}(n\log\log\sigma)$ time and $\mathcal{O}(n\log\sigma)$ bits of extra space. Our construction, instead, works within compressed space.

Very recently, Navarro and Nekrich [2012] improved in part upon Theorem 1.5, by reducing the time of all the operations to $\mathcal{O}(\log n/\log\log n)$, independently of σ. The time for updates, however, is amortized. This allows them to partly outperform our results in Corollary 8.2 (yet with amortized update times), and completely outperform our Corollary 8.3, where they can build the FM-index and the BWT in time $\mathcal{O}(n\log n/\log\log n) = o(n\log n)$.

While it is not obvious at a first glance, our construction permits more general operations on the sequence, such as range searches [Mäkinen and Navarro 2007; Navarro 2012]. These require tracking sequence positions and intervals towards the sequence inside any node of the wavelet tree, which is done via *rank* and *select* operations on the node sequences. The solution of Navarro and Nekrich [2012] precisely avoids using those operations in the intermediate wavelet tree nodes. This is sufficient for accessing symbols and supporting *rank*$_c$ and *select*$_c$ operations, but not for general tracking of intervals inside the wavelet tree. The lower bound $\Omega((\log n/\log\log n)^2)$ for dynamic range counting [Patrascu 2007] suggests that our solution is indeed optimal if such range search capabilities are to be supported.

9. CONCLUDING REMARKS

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

$2n + \mathcal{O}(n/\text{polylog}(n))$ bits of space, for a tree of n nodes and a polylog of any degree. This significantly improves upon the redundancy of previous representations. The core of the idea is the range min-max tree. 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 show how constant time can be achieved by using the range min-max tree as a building block for handling larger trees.

The simple variant using one range min-max tree has actually been implemented and compared with the state of the art over several real-life trees [Arroyuelo et al. 2010]. It has been shown that it is by far the smallest and fastest representation in most cases, as well as the one with widest coverage of operations. It requires around 2.37 bits per node and carries out most operations within a microsecond on a standard PC.

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, which is optimal in the dynamic case even for a very reduced set of operations. The solution is based on dynamic range min-max trees, and especially the first variant is extremely simple and implementable (indeed, a very recent publication [Joannou and Raman 2012] studies its implementation issues, obtaining good performance results). The flexibility of the structure is illustrated by the fact that we can support much more complex operations, such as attaching and detaching whole subtrees.

Our work contains several ideas of independent interest. An immediate application to storing a dynamic sequence of numbers supporting operations *sum* and *search* achieves optimal time $\mathcal{O}(\log n/\log\log n)$. Another application is the storage of dynamic compressed sequences achieving zero-order entropy space and improving the redundancy of previous work. It also improves the times for the operations, achieving the optimal $\mathcal{O}(\log n/\log\log n)$ for polylog-sized alphabets. This in turn has several applications to compressed text indexing. Our *lrm* trees have recently been applied to the compression of permutations [Barbay et al. 2011a]. Our range min-max trees have also been used to represent longest common prefix information on practical compressed suffix trees [Cánovas and Navarro 2010].

Pătraşcu and Viola [2010] have recently shown that $n + n/w^{\Theta(t)}$ bits are necessary to compute *rank* or *select* on bitmaps in worst-case time $\mathcal{O}(t)$. This lower bound holds also in the subclass of balanced bitmaps[14] (i.e., those corresponding to balanced parenthesis sequences), which makes our redundancy on static trees optimal as well, at least for some of the operations: Since *rank* or *select* can be obtained from any of the operations *depth*, *pre_rank*, *post_rank*, *pre_select*, *post_select*, any balanced parentheses representation supporting any of these operations in time $\mathcal{O}(t)$ requires $2n+n/w^{\Theta(t)}$ bits of space. Still, it would be good to show a lower bound for the more fundamental set of operations *findopen*, *findclose*, and *enclose*.

On the other hand, the complexity $\mathcal{O}(\log n/\log\log n)$ is known to be optimal for several basic dynamic tree operations (*findopen*, *findclose*, and *enclose*), but not for all. It is also not clear if the redundancy $\mathcal{O}(n/r)$ achieved for the dynamic trees, $r = \log n$ for the simpler structure and $r = \log\log n/\log n$ for the more complex one, is optimal to achieve the corresponding $\mathcal{O}(r)$ operation times. Finally, it would be good to achieve $\mathcal{O}(\log n/\log\log n)$ time for all the operations or prove it impossible.

Acknowledgments

We thank Mihai Pătraşcu for confirming us the construction cost of his aB-tree and rank/select data structure [Pătraşcu 2008]. And we thank him again for confirming us

[14]M. Pătraşcu, personal communication.

that the lower bound of Pătraşcu and Viola [2010] holds for balanced sequences. We also thank Meng He for useful technical remarks on Section 7 and Jérémy Barbay for pointing out the relation between Cartesian and lrm trees.

REFERENCES

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

Arroyuelo, D., Cánovas, R., Navarro, G., and Sadakane, K. 2010. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM Press, 84–97.

Barbay, J., Fischer, J., and Navarro, G. 2011a. LRM-Trees: Compressed indices, adaptive sorting, and compressed permutations. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 6661. 285–298.

Barbay, J., He, M., Munro, J. I., and Rao, S. S. 2011b. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms 7,* 4, article 52.

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

Bender, M. and Farach-Colton, M. 2004. The level ancestor problem simplified. *Theoretical Computer Science 321,* 1, 5–12.

Benoit, D., Demaine, E. D., Munro, J. I., Raman, R., Raman, V., and Rao, S. S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4, 275–292.

Burrows, M. and Wheeler, D. 1994. A block sorting data compression algorithm. Tech. rep., Digital Systems Research Center.

Cánovas, R. and Navarro, G. 2010. Practical compressed suffix trees. In *Proc. 9th International Symposium on Experimental Algorithms (SEA)*. LNCS 6049. 94–105.

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

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

Davoodi, P., Raman, R., and Rao, S. S. 2012. Succinct representations of binary trees for range minimum queries. In *Proc. 18th Annual International Conference on Computing and Combinatorics (COCOON)*. LNCS 7434. 396–407.

Davoodi, P. and Rao, S. S. 2011. Succinct dynamic cardinal trees with constant time operations for small alphabet. In *Proc. 8th Annual Conference on Theory and Applications of Models of Computation (TAMC)*. 195–205.

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

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

Farzan, A. and Munro, J. I. 2011. Succinct representation of dynamic trees. *Theoretical Computer Science 412,* 24, 2668–2678.

Farzan, A., Raman, R., and Rao, S. S. 2009. Universal succinct representations of trees? In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*. 451–462.

Ferragina, P., Luccio, F., Manzini, G., and Muthukrishnan, S. 2009. Compressing and indexing labeled trees, with applications. *Journal of the ACM 57,* 1, article 4.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2007. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms 3,* 2, No. 20.

Fischer, J. 2010. Optimal succinctness for range minimum queries. In *Proc. 9th Symposium on Latin American Theoretical Informatics (LATIN)*. LNCS 6034. 158–169.

Fischer, J. and Heun, V. 2007. 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. 459–470.

Fleischer, R. 1996. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science 7,* 2, 137–149.

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

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

Geary, R., Rahman, N., Raman, R., and Raman, V. 2006a. A simple optimal representation for balanced parentheses. *Theoretical Computer Science 368,* 3, 231–246.

Geary, R., Raman, R., and Raman, V. 2006b. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms 2,* 4, 510–534.

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

González, R. and Navarro, G. 2008. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science 410*, 4414–4422.

Grossi, R., Gupta, A., and Vitter, J. S. 2003. High-Order Entropy-Compressed Text Indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

He, M. and Munro, I. 2010. Succinct representations of dynamic strings. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 6393. 334–346.

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

Hon, W. K., Sadakane, K., and Sung, W. K. 2009. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM Journal on Computing 38,* 6, 2162–2178.

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

Jansson, J., Sadakane, K., and Sung, W.-K. 2012. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and Systems Sciences 78,* 2, 619–631.

Joannou, S. and Raman, R. 2012. Dynamizing succinct tree representations. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*. 224–235.

Lu, H.-I. and Yeh, C.-C. 2008. Balanced parentheses strike back. *ACM Transactions on Algorithms 4,* 3, article 28.

Mäkinen, V. and Navarro, G. 2007. Rank and select revisited and extended. *Theoretical Computer Science 387,* 3, 332–347.

Mäkinen, V. and Navarro, G. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms 4,* 3, article 32.

Manzini, G. 2001. An Analysis of the Burrows-Wheeler Transform. *Journal of the ACM 48,* 3, 407–430.

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

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

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

Munro, J. I., Raman, V., and Rao, S. S. 2001a. Space efficient suffix trees. *Journal of Algorithms 39,* 2, 205–222.

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

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

Navarro, G. 2012. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 7354. 2–26.

Navarro, G. and Nekrich, Y. 2012. Optimal dynamic sequence representations. *CoRR abs/1206.6982v1*.

Okanohara, D. and Sadakane, K. 2009. A linear-time burrows-wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 90–101.

Pagh, R. 2001. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing 31,* 2, 353–363.

Patrascu, M. 2007. Lower bounds for 2-dimensional range counting. In *Proc. 39th Annual ACM Symposium on Theory of Computing (STOC)*. 40–46.

Pǎtraşcu, M. and Demaine, E. D. 2006. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing 35,* 4, 932–963.

Pǎtraşcu, M. and Viola, E. 2010. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 117–122.

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

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

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

Raman, R., Raman, V., and Rao, S. S. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms 3,* 4, article 8.

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

Sadakane, K. 2002. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 225–232.

Sadakane, K. 2007a. Compressed suffix trees with full functionality. *Theory of Computing Systems 41,* 4, 589–607.

Sadakane, K. 2007b. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms 5*, 12–22.

Sadakane, K. and Navarro, G. 2010. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 134–149.

Vuillemin, J. 1980. A unifying look at data structures. *Communications of the ACM 23,* 4, 229–239.