Noname manuscript No.
(will be inserted by the editor)

Succinct Dynamic Cardinal Trees

Diego Arroyuelo · Pooya Davoodi · Srinivasa Rao Satti

the date of receipt and acceptance should be inserted later

Abstract *Cardinal trees* (or *tries* of degree k) are a fundamental data structure, in particular for many text-processing algorithms. Each node in a cardinal tree has at most k children, each labeled with a symbol from the alphabet $\{1, \ldots, k\}$. In this paper we introduce succinct representations for dynamic cardinal trees on n nodes, requiring $2n + n \lg k + o(n \lg k)$ bits of space. These are the first dynamic cardinal tree representations that support a (fairly) complete set of operations while using almost optimal space. For $k = \mathcal{O}(\text{polylog}(n))$, we show how the navigational and query operations on the tree can be supported in $\mathcal{O}(1)$ time, while supporting insertions and deletions of leaves in $\mathcal{O}(1)$ amortized time. For $k = \omega(\text{polylog}(n))$ (and $\mathcal{O}(n)$), we show that the same set of operations can be supported in $\mathcal{O}(\lg k / \lg \lg k)$ time (amortized in the case of insertions/deletions). We also show how to associate b-bit satellite data to the tree nodes using $bn + o(n)$ extra bits. Finally, we show how the machinery introduced for dynamic cardinal trees can be adapted to represent dynamic binary trees using $2n + o(n)$ bits of space, so that the tree

Preliminary versions of this paper appeared in *Proceedings of 19th Annual Symposium on Combinatorial Pattern Matching* (*CPM* 2008), LNCS 5029, pages 277–289, and *Proceedings of 8th Annual Conference on Theory and Applications of Models of Computation* (*TAMC* 2011), LNCS 6648, pages 195–205.

Research partly supported by FONDECYT grant 11121556 (first author).

Research supported by NSF Grant CCF-1018370 and BSF Grant 2010437 (second author).

Research partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology, Grant number 2012-0008241 (third author).

Diego Arroyuelo
Dept. of Informatics, Univ. Técnica Federico Santa María & Yahoo! Labs Santiago
E-mail: darroyue@inf.utfsm.cl

Pooya Davoodi
Dept. of Computer Science and Engineering, Polytechnic Institute of NYU
E-mail: pooyadavoodi@gmail.com

Srinivasa Rao Satti
School of Computer Science and Engineering, Seoul National University
E-mail: ssrao@cse.snu.ac.kr

operations are supported in $\mathcal{O}(1)$ time (amortized in the case of insert/delete operations). We support adding satellite data to the tree nodes using $bn + o(n)$ extra bits (versus $bn + o(bn)$ extra bits of the fastest previous dynamic representation from the literature), while providing several trade-offs for accessing/modifying the data.

1 Introduction and Previous Work

An *ordinal tree* (or *ordered tree*) is a rooted tree where each node has an arbitrary (usually unbounded) number of children. These children are ordered, such that there is a first child, a second child, and so on. We say that the i-th child of a node x has rank i among its siblings. A k-ary *cardinal tree* is an ordinal tree of degree k (i.e., nodes have *at most* k children), where each edge is labeled with a unique symbol drawn from the *alphabet* $\{1, \ldots, k\}$. If we let $\mathsf{label}(x, i)$ to denote the label of the edge that connects x and its i-th child, then it holds that:

$$\mathsf{label}(x, i) < \mathsf{label}(x, j) \Leftrightarrow i < j.$$

In other words, the edge labels of the children are in the increasing order of their ranks. This kind of tree are also known as *tries* of degree k. In a *dynamic cardinal tree*, leaf nodes can be inserted or deleted from the tree. In this paper we study the succinct representation of *dynamic cardinal trees*, a problem that remained open for several years [28].

A *succinct data structure* is one that requires space close to the *information-theoretic lower bound* (besides lower-order additive terms). Since the number of different k-ary cardinal trees with n nodes is $\frac{1}{kn+1}\binom{kn+1}{n}$ [19], the information-theoretic lower bound for the number of bits needed to represent a cardinal tree is:

$$\mathcal{C}(n, k) = \left\lceil \lg \frac{1}{kn+1} \binom{kn+1}{n} \right\rceil,$$

where $\lg x$ will mean $\log_2 x$ throughout this paper. Assuming that k is a growing function of n, we have that $\mathcal{C}(n, k) \approx n(\lg k + \lg e) - o(n + \lg k)$ bits.

Besides requiring little space, succinct data structures in general support operations as efficiently as their non-space-efficient counterparts. In the case of trees, we are interested in succinct representations that can be efficiently navigated. In Table 1 we show the tree operations that are supported by the tree representations in this paper.

1.1 Model of Computation

1.1.1 Word RAM Model

We consider the *standard word* RAM model of computation, where the memory is regarded as an array of *words* numbered 0 to $2^w - 1$, for $w = \Theta(\lg n)$. Every memory word of w bits can be accessed in constant time. Basic arithmetic and logical operations can be computed in constant time.

Table 1 Tree operations supported by our dynamic representation of cardinal trees.

parent(x):	Returns the parent of node x.
child(x, i):	Returns the i-th child of node x.
label-child(x, α):	Returns the child of node x labeled with symbol $\alpha \in \{1, \ldots, k\}$.
child-rank(x):	Returns the rank of node x among its siblings.
label(x, i):	Returns the label of the i-th child of node x.
degree(x):	Returns the number of children of node x.
subtree-size(x):	Returns the size of the subtree rooted at node x.
preorder(x):	Returns the preorder rank of node x.
ancestor(x, y):	Returns true if node x is ancestor of node y; false otherwise.
access-data(x):	Returns the data associated to node x.
change-data(x, d):	Associates data d to node x.
insert-leaf(x, α):	Inserts a new leaf child of node x with label α.
delete-leaf(x, α):	Deletes the leaf child of node x with label α.

1.1.2 Dynamic-Memory Model

For the allocation of dynamic memory, we assume that there are no system calls for allocation and deallocation of memory, but the program must handle memory by itself. In this model, the space usage of an algorithm at a given time is the highest numbered memory word that is currently in use by the algorithm (this is the same as the memory model $\mathcal{M}_B$ described in [33]. This is one of the standard dynamic memory models for RAM, and assumes the least from the system.

1.1.3 Model of Tree Operations

We will assume that all navigational operations in the tree start from the tree root and that insertions and deletions occur at leaves. This is the so-called *traversal model* of tree operations. All operations have to be carried out on the current node, x, of the traversal. This is the same model as in previous work [28, 33]. This model is also usual in many applications, such as constructing the Lempel-Ziv 1978 parsing of a text [4, 22] within compressed space, and constructing suffix trees, for example.

1.2 Previous Related Work

The classical pointer-based representation of a tree requires $\Theta(n \lg n)$ bits. This is space consuming [3], hence data-intensive applications can only keep relatively small trees in main memory. Typical examples are that of DOM trees for XML documents [18], and suffix trees [1] for full-text search applications, just to name two.

The seminal work of Jacobson [21] started the quest for the succinct representation of trees (and succinct data structures in general). Over the years, trees have become one of the most successful examples of the application of succinct data structures. Nowadays, there are many succinct representations of trees that use sligthly more than 2 bits per node and support a comprehensive set of operations, both in theory [27, 7, 32, 18, 23, 14, 15, 13, 30], and in practice [3, 24]. We now summarize the most important results related to our work.

Succinct Static Cardinal Trees. For static k-ary cardinal trees on n nodes, Benoit et al. [7] gave a representation that requires $2n + n\lceil \lg k \rceil + o(n) + \mathcal{O}(\lg \lg k)$ bits, and supports the navigational operations and queries in $\mathcal{O}(1)$ time. Hence, the space bound of their structure is $\mathcal{C}(n,k) + \Theta(n)$ bits, as k grows. Raman et al. [32] improved the space to $\mathcal{C}(n,k) + o(n) + \mathcal{O}(\lg \lg k)$ bits, while supporting all the operations, except subtree-size, in $\mathcal{O}(1)$ time. Finally, a representation with the same space that also supports subtree-size in $\mathcal{O}(1)$ time was given by Farzan et. al. [14]. Table 2 summarizes the existing succinct representations of cardinal trees.

Succinct Dynamic Binary Trees. Succinct representation of dynamic trees was first studied for binary trees (i.e., for $k = 2$). In this case we are interested in operations left-child(x) (which yields the left child of a node x) and right-child(x) (which yields the right child of a node x). Since there are $\frac{1}{2n+1}\binom{2n+1}{n}$ different binary trees with n nodes, the information-theoretic lower bound is $\left\lceil \lg \frac{1}{2n+1}\binom{2n+1}{n} \right\rceil = 2n - \mathcal{O}(\lg n)$ bits.

Munro et al. [28] gave the first dynamic binary tree representation that uses $2n + o(n)$ bits. This representation, in the course of traversing the tree, supports navigational operations and queries in $\mathcal{O}(1)$ time and updates in $\mathcal{O}(\lg^2 n)$ amortized time. Their structure can also support accessing a b-bit piece of satellite data associated with each node in $\mathcal{O}(1)$ time, where $b = \Theta(\lg n)$. If $b = \mathcal{O}(1)$, they achieve $\mathcal{O}(\lg n)$ amortized update time, and if no satellite data is associated with the nodes, they obtain $\mathcal{O}(\lg \lg n)$ amortized update time. See Table 3 for a summary of existing succinct representations of dynamic binary trees.

For $b = \mathcal{O}(\lg n)$, Raman and Rao [33] improved the update time to $\mathcal{O}((\lg \lg n)^{1+\epsilon})$ amortized, while supporting the navigation and queries in $\mathcal{O}(1)$ worst-case time, in the course of traversing the tree. They also showed how to store the satellite data in $bn + o(n)$ bits. Their total space is $2n + bn + o(n)$ bits.

More recently, Farzan and Munro [13] proposed the *finger-update* model, which is stronger than the traversal model assumed in this paper. In the finger-update model, only the update operations are restricted to be performed on the current node of the traversal, whereas all the other operations are allowed to be performed on any node at any time. For satellite data of $b = \mathcal{O}(\lg n)$ bits, their data structure [13] supports the navigation operations in constant time and updates in constant amortized time. Their structure uses $2n + bn + o(bn)$ bits, which is worse than that of [33] because of the $o(bn)$ term needed to support the satellite data.

Succinct Dynamic Cardinal Trees. Darragh et al. [11] presented a compact form of representing cardinal trees named Bonsai trees, that uses $6n + n\lceil \lg k \rceil$ bits of space and supports navigation and inserting new leaves in $\mathcal{O}(1)$ expected time. Their method is essentially a way of storing trees in a compact form of hash table. The 6 bits that are stored at each node of the tree (which make the $6n$ bits of the total space) can be increased to 9 bits or even more in order to obtain a lower hash collision probability. However, it has been shown that 6 bits provide a small enough upper bound for the probability [11]. The efficient representation of succinct dynamic cardinal trees was posed as an open problem by Munro et al. [28]. The problem remained open until Arroyuelo's work [2], where a representation of dynamic cardinal trees is introduced, which uses space close to optimal: $2n + n\lceil \lg k \rceil + o(n \lg k)$ bits. However, just a simple set of operations is provided (child,

Table 2 Summary of the existing succinct representation of cardinal trees. Times marked with '†' are amortized. DFUDS corresponds to [7], RRR07 corresponds to [32], and FRR09 corresponds to [14]. Notice that these three methods are static.

Operation	DFUDS	RRR07	FRR09	Ours $k = \mathcal{O}(\text{polylog}(n))$	Ours $k = \omega(\text{polylog}(n))$
parent	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
child	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
label-child	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
child-rank	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
label	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
degree	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
subtree-size	$\mathcal{O}(1)$	$\times$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
preorder	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
ancestor	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg k/\lg\lg k)$
access-data	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)^{\dagger}$	$\mathcal{O}(\lg k/\lg\lg k)^{\dagger}$
change-data	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)^{\dagger}$	$\mathcal{O}(\lg k/\lg\lg k)^{\dagger}$
insert-leaf	$\times$	$\times$	$\times$	$\mathcal{O}(1)^{\dagger}$	$\mathcal{O}(\lg k/\lg\lg k)^{\dagger}$
delete-leaf	$\times$	$\times$	$\times$	$\mathcal{O}(1)^{\dagger}$	$\mathcal{O}(\lg k/\lg\lg k)^{\dagger}$
Space bound on top of $\mathcal{C}(n,k)$	$\Theta(n)$	$o(n)$	$o(n)$	$\Theta(n) + o(n\lg k)$	$\Theta(n) + o(n\lg k)$
Space for b-bit satellite data	bn	bn	bn	$bn + o(n)$	$bn + o(n)$

Table 3 Summary of the existing succinct representation of dynamic binary trees. Times marked with '†' are amortized. MRS01 corresponds to [28], RR03 corresponds to [33], and FM10 corresponds to [13].

Operation	MRS01	RR03	FM10	Ours
parent	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
left-child/right-child	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
subtree-size	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
preorder	$\times$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
ancestor	$\times$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
access-data	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)^{\dagger}$
change-data	$\mathcal{O}(1)$	$\mathcal{O}((\lg\lg n)^{1+\epsilon})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)^{\dagger}$
insert-leaf	$\mathcal{O}(\lg^2 n)^{\dagger}$	$\mathcal{O}((\lg\lg n)^{1+\epsilon})^{\dagger}$	$\mathcal{O}(1)^{\dagger}$	$\mathcal{O}(1)^{\dagger}$
delete-leaf	$\mathcal{O}(\lg^2 n)^{\dagger}$	$\mathcal{O}((\lg\lg n)^{1+\epsilon})^{\dagger}$	$\mathcal{O}(1)^{\dagger}$	$\mathcal{O}(1)^{\dagger}$
space for b-bit satellite data	$bn + o(n)$	$bn + o(n)$	$bn + o(bn)$	$bn + o(n)$

label-child, parent, insert-leaf, and delete-leaf), and no support for satellite data is given. Afterwards, Davoodi and Rao [12] improved the previous result for the case of small alphabets, namely $k = \mathcal{O}(\text{polylog}(n))$. In that case, they show how the tree operations can be supported in $\mathcal{O}(1)$ time (amortized in the case of updates).

1.3 Contributions of this Work

In this paper we give insight into a problem that has remained almost unexplored in the area of succinct data structures. We introduce succinct representations of dynamic cardinal trees, supporting the operations of Table 1 (that is, a more complete set of operations than those in [2] and [12]) and requiring space close to the information-theoretic lower bound: $2n + n \lg k + o(n \lg k)$ bits.

In the traversal model and for $k = \mathcal{O}(\text{polylog}(n))$, we show that the navigation operations can be supported in $\mathcal{O}(1)$ worst-case time, whereas updates are supported in $\mathcal{O}(1)$ amortized time. If we associate b-bit satellite data to the tree nodes (for $b = \mathcal{O}(\lg n)$), our data structure offers the following space/time trade-offs: (1) $bn+o(n)$ extra bits of space, $\mathcal{O}(1)$ time for operation access-data, and $\mathcal{O}(\lg n/\lg\lg n)$ amortized time for operation change-data; (2) $bn + \mathcal{O}(n \lg\lg n)$ bits, $\mathcal{O}(1)$ time for access-data, and $\mathcal{O}(1)$ amortized time for change-data; and (3) $bn + o(n)$ extra bits, and $\mathcal{O}(\lg\lg n/\lg\lg\lg n)$ amortized time for access-data and change-data. We summarize our results in Table 2. The table also shows results for existing succinct representations of static cardinal trees.

For $k = \omega(\text{polylog}(n))$ (and $k = \mathcal{O}(n)$), on the other hand, we show that the navigation operations can be supported in $\mathcal{O}(\lg k/\lg\lg k)$ worst-case time, whereas update operations are supported in $\mathcal{O}(\lg k/\lg\lg k)$ amortized time. Satellite data can be stored using $bn + o(n)$ extra bits of space, allowing the access/modification of the data in $\mathcal{O}(\lg k/\lg\lg k)$ amortized time. These results are also summarized in Table 2.

Finally, we show that our approaches are suitable to represent dynamic binary trees. We achieve $\mathcal{O}(1)$ time for the navigation operations, and $\mathcal{O}(1)$ amortized time for the update operations. These are the same as Farzan and Munro [13]. For b-bit satellite data, our data structure offers the same three trade-offs as for $k = \mathcal{O}(\text{polylog}(n))$ mentioned earlier. See Table 3 for a summary of our results and a comparison with previous work.

2 Preliminary Concepts

We summarize next some of the important concepts needed to understand the results of this paper.

2.1 Data Structures for rank and select Operations

Given a sequence $S[1..n]$ over an alphabet $\{1, \ldots, k\}$ and given any $c \in \Sigma$, we define operations:

- $\mathsf{rank}_c(S, i)$, which yields the number of occurrences of c in $S[1..i]$.
- $\mathsf{select}_c(S, j)$, which yields the position of the j-th c in S.
- $\mathsf{access}(S, i)$, which yields $S[i]$.

In the dynamic case we also want to insert/delete symbols into/from the sequence. The data structure of Navarro and Nekrich [29] supports all the operations (including insertion and deletion of symbols at arbitrary positions of the sequence) in $\mathcal{O}\left(\frac{\lg n}{\lg\lg n}\right)$ time (amortized in the case of insertions and deletions),

using $nH_0(S) + o(n \lg k)$ bits of space, where $H_0(S) \leqslant \lg k$ denotes the zero-order empirical entropy of S [26].

2.2 Data Structures for Searchable Partial Sums

Given an array $A[1..m]$ of m integers from the range $[0..r-1]$, a *searchable partial sum* data structure [17] allows one to retrieve any $A[i]$, as well as to support the following operations:

- $\mathsf{Sum}(A, i)$, which computes $\sum_{j=1}^{i} A[j]$;
- $\mathsf{Search}(A, i)$, which finds the smallest j such that $\mathsf{Sum}(A, j) \geqslant i$;
- $\mathsf{Update}(A, i, \delta)$, which sets $A[i] \leftarrow A[i] + \delta$, assuming that $A[i] + \delta < r$, and δ is less than a certain fixed number;
- $\mathsf{Insert}(A, i, e)$, which inserts a new element e into the array A between elements $A[i-1]$ and $A[i]$; and
- $\mathsf{Delete}(A, j)$, which deletes element $A[j]$.

This problem has been considered for different ranges of m, r and δ [31,20]. In particular, in this paper we will need the following result:

Lemma 1 ([30]) *There exists a representation for an array $A[1..m]$ of m integers (each in the range $[0..r-1]$, for $r = \mathcal{O}(\lg m)$) using $m \lg r(1 + o(1))$ bits of space and supporting the access to any $A[i]$ ($1 \leqslant i \leqslant m$), as well as operations* Sum, Search, Update, Insert *and* Delete *in $\mathcal{O}(\lg m / \lg \lg m)$ time.*

2.3 Data Structures for Dynamic Arrays

A *dynamic array* [33] is a data structure that supports accessing, inserting, and deleting arbitrary elements in arrays. The aim is to support dynamic-array operations fast and with a small memory overhead. For dynamic arrays with few elements, Raman and Rao [33] obtained the following result:

Lemma 2 ([33,34]) *There exists a data structure to represent an array of $\ell = w^{\mathcal{O}(1)}$ elements, each of size $r = \mathcal{O}(w)$ bits, using $\ell r + \mathcal{O}(c \lg \ell)$ bits, for any parameter $c \leqslant \ell$. This data structure supports accessing the element of the array at a given index in $\mathcal{O}(1)$ time, and inserting/deleting an element at a given index in $\mathcal{O}(1 + \ell r/cw)$ amortized time. The data structure requires a precomputed table of size $\mathcal{O}(2^{\epsilon w})$ bits for any fixed $\epsilon > 0$.*

For the general case, Raman et al. [31] obtained the following result:

Lemma 3 ([31]) *A dynamic array of n elements, each of size $b = \mathcal{O}(\lg n)$ bits, can be represented using $bn + o(n)$ bits of space. This structure supports accessing the element of the array at a given index, as well as inserting/deleting an element at a given index, all in $\mathcal{O}(\lg n / \lg \lg n)$ amortized time.*

2.4 Data Structures for Balanced Parentheses and Ordinal Trees

The problem of representing a sequence of balanced parentheses is related to the succinct representation of trees [27]. Given a sequence P of $2n$ balanced parentheses, we want to support the following operations:

- $\mathsf{findclose}(P, i)$: given $P[i] =$ '(', returns the position of the matching closing parenthesis;
- $\mathsf{findopen}(P, i)$: given $P[i] =$ ')', returns the position of the matching opening parenthesis;
- $\mathsf{excess}(P, i)$: returns the difference between the number of opening and closing parentheses up to position i in P;
- $\mathsf{enclose}(P, i)$: given a parenthesis pair whose opening parenthesis is at position i, returns the position of the opening parenthesis corresponding to the closest matching pair enclosing i.

Since there are $\frac{1}{2n+1}\binom{2n+1}{n}$ different sequences of n pairs of matching balanced parentheses, the information-theoretic lower bound to represent such a sequence is $\lceil \lg \frac{1}{2n+1}\binom{2n+1}{n} \rceil = 2n - \mathcal{O}(\lg n)$ bits.

Munro and Raman [27] showed how to implement all these operations in constant time and using $2n + o(n)$ bits. They also showed the main application of balanced-parenthesis sequences: the succinct representation of ordinal trees. The main operations that must be supported by an ordinal tree are:

- $\mathsf{first\text{-}child}(x)$, which yields the first child of node x (if it exists);
- $\mathsf{next\text{-}sibling}(x)$, which yields the next sibling of node x (if it exists);
- $\mathsf{prev\text{-}sibling}(x)$, which yields the previous sibling of node x (if it exists); and
- $\mathsf{parent}(x)$, which yields the parent of node x (if it exists).

Given an ordinal tree, the following procedure transforms it into a balanced-parenthesis sequence (showing a one-to-one mapping between the two classes of objects): perform a depth-first traversal on the tree and write a '(' in the sequence P when we visit a node x for the first time, then visit the subtree of x in preorder, and finally write a ')'. In this way, each node x in the tree is represented by a pair of matching '(' and ')', which enclose the parentheses corresponding to the nodes in the subtree of x. In particular, node x is represented by the position of the '(' corresponding to it. The above-mentioned ordinal-tree operations can be supported via balanced-parenthesis operations [27].

After Munro and Raman's data structure, several others were introduced, which extend the supported functionality [25, 30].

In the dynamic case, the parenthesis sequence can change over time, by inserting/deleting a pair of *matching* parentheses into/from the sequence. The dynamic data structure of Navarro and Sadakane [30] supports the above operations (plus rank and select on the parenthesis sequence), including also insertions and deletions, in $\mathcal{O}(\lg n / \lg \lg n)$ time. The data structure uses $2n + o(n)$ bits of space, as summarized below.

Lemma 4 ([30]) *There exists a representation for a dynamic sequence of $2n$ balanced parentheses using $2n + o(n)$ bits of space and supporting operations* findclose*,* findopen*,* excess*,* enclose*,* rank*, and* select*, as well as insertions and deletions of pairs of matching parentheses, all of them in $\mathcal{O}(\lg n / \lg \lg n)$ worst-case time.*

2.5 Succinct Representation of Static Cardinal Trees

Among the existing succinct representations of static cardinal trees [7, 32, 23, 14], we are particularly interested in Depth-First Unary Degree Sequence [7] (DFUDS, for short), which seems (at first) easier to dynamize. To get this representation, we carry out a depth-first traversal on the tree. For every node reached in this traversal, we write its degree in unary, using parentheses. For instance, a node of degree 3 is written as '`(((`', while a leaf is just '`)`'. What we get is almost a balanced-parenthesis sequence: we only need to add a '`(`' at the beginning of the sequence to make it balanced. A node of degree d is identified by the position of the first of the $d+1$ parentheses representing the node. In the case of a leaf, the position of '`)`' is used to represent the node. Let us call $\mathtt{D}[1..2n]$ the sequence of balanced parentheses generated by DFUDS.

2.5.1 Supporting Basic Tree Operations

According to [7], the tree operations can be supported in $\mathcal{O}(1)$ time by means of rank, select, and balanced-parenthesis operations on the sequence D as follows:

$$\begin{aligned}
\mathsf{parent}(x) &\equiv \mathsf{select}_{)}(\mathtt{D}, \mathsf{rank}_{)}(\mathtt{D}, \mathsf{findopen}(\mathtt{D}, x-1))) + 1,\\
\mathsf{child}(x,i) &\equiv \mathsf{findclose}(\mathtt{D}, \mathsf{select}_{)}(\mathtt{D}, \mathsf{rank}_{)}(\mathtt{D}, x) + 1) - i) + 1,\\
\mathsf{child\text{-}rank}(x) &\equiv \mathsf{select}_{)}(\mathtt{D}, \mathsf{rank}_{)}(\mathtt{D}, \mathsf{findopen}(\mathtt{D}, x-1)) + 1) - \mathsf{findopen}(\mathtt{D}, x-1),\\
\mathsf{degree}(x) &\equiv \mathsf{select}_{)}(\mathtt{D}, \mathsf{rank}_{)}(\mathtt{D}, x) + 1) - x,\\
\mathsf{subtree\text{-}size}(x) &\equiv (\mathsf{findclose}(\mathtt{D}, \mathsf{enclose}(\mathtt{D}, x)) - x)/2 + 1,\\
\mathsf{preorder}(x) &\equiv \mathsf{rank}_{)}(\mathtt{D}, x-1),\\
\mathsf{ancestor}(x,y) &\equiv \mathtt{true} \text{ iff } x \leqslant y \leqslant \mathsf{findclose}(\mathtt{D}, \mathsf{enclose}(\mathtt{D}, x)); \mathtt{false} \text{ otherwise.}
\end{aligned}$$

2.5.2 Supporting label-child *and* label *Operations*

To support $\mathsf{label\text{-}child}(x, \alpha)$ on DFUDS, we define a set L that represents the edge labels as follows. For each node x in the tree with preorder $i = 0, \ldots, n-1$ and d children labeled $\alpha_1, \ldots, \alpha_d$, let us define the set $C_i = \{ik + \alpha_1, \ldots, ik + \alpha_d\}$. Then, we define $\mathtt{L} = \bigcup_{i=0}^{n-1} C_i$. Notice that this is similar to the set defined in the proofs of Theorems 6.1 and 6.2 of [32], except that we use depth-first order here. Actually, it is the same as the set defined in the proof of Theorem 6 of [14]. Since the tree has $n-1$ edges, it holds that $|\mathtt{L}| = n-1$. Also, the universe of set L is $\{1, \ldots, kn\}$.

Set L is represented using the indexable dictionary from [32], using $\lceil \lg \binom{kn}{n-1} \rceil + o(n)$ bits of space, which is $\mathcal{C}(n,k) + o(n) + \mathcal{O}(\lg \lg k)$ bits. This data structure allows us to compute operations $\mathsf{rank}(\mathtt{L}, j)$ (which returns -1 iff $j \notin \mathtt{L}$, otherwise it yields $|\{y < j \mid y \in \mathtt{L}\}|$) and $\mathsf{select}(\mathtt{L}, i)$, which yields the i-th smallest element in L. Both operations are supported in $\mathcal{O}(1)$ time [32].

The operation $\mathsf{label\text{-}child}(x, \alpha)$ is undefined if $\mathsf{rank}(\mathtt{L}, k \cdot \mathsf{preorder}(x) + \alpha) = -1$. Otherwise, we compute it using the equation

$$\mathsf{label\text{-}child}(x, \alpha) = \mathsf{child}(x, \mathsf{rank}(\mathtt{L}, k \cdot \mathsf{preorder}(x) + \alpha) - (\mathsf{rank}_{(}(\mathtt{D}, x-1) - 1) + 1),$$

which can be computed in $\mathcal{O}(1)$ time. The operation $\mathsf{label}(x, i)$ can also be supported in $O(1)$ time using the equation

$$\mathsf{label}(x, i) \equiv \mathsf{select}(\mathtt{L}, \mathsf{rank}_{(}(\mathtt{D}, x-1) - 1 + i) - k \cdot \mathsf{preorder}(x).$$

Note that this structure uses $\mathcal{C}(n,k) + 2n + o(n) + \mathcal{O}(\lg\lg k)$ bits of space. To avoid storing the DFUDS sequence (thus saving $2n$ bits and achieving optimal space usage, up to lower-order terms), Farzan et al. [14] show how to retrieve from the encoding of $\mathtt{L}$, any $\lg n$-bit subsequence of the DFUDS sequence in constant time.

2.5.3 Satellite Data

Finally, to associate satellite data to the tree nodes, we use an array $\mathtt{Data}[0..n-1]$ such that $\mathtt{Data}[\mathsf{preorder}(x)]$ stores the data associated with node x.

3 A Dynamic DFUDS Representation for Cardinal Trees

Though recent approaches support insertions and deletions in ordinal trees of n nodes in $\mathcal{O}(\lg n/\lg\lg n)$ time [30], we are interested in dynamic k-ary *cardinal* trees in this paper. We show in this section how the DFUDS representation of a tree can be dynamized. This shall be the building block for the results obtained in later sections.

3.1 Tree Topology

We represent the DFUDS of the cardinal tree with the dynamic data structure of Lemma 4. We denote this representation by $\mathtt{D}$, which requires $2n + o(n)$ bits. The edge labels are stored in a dynamic sequence $\mathtt{L}$. The labels of the children of node x are stored contiguously and in order in $\mathtt{L}$, following a depth-first traversal. In order to compute operation label-child efficiently, we must keep the symbols of a node always sorted. This means that operation $\mathsf{insert\text{-}leaf}(x,\alpha)$ must compute the insertion rank of the new child of x among its siblings, and then insert α in $\mathtt{L}$.

To support this functionality efficiently under insertions, we use *gap encoding* for the labels: the label of the first child of a node is stored in absolute form (i.e., using the actual encoding of the symbol), whereas the remaining labels are represented in differential form. For instance, if a node x has children labeled a, b and f, we store a, $b-a$ and $f-b$ in consecutive positions. The opening parentheses in $\mathtt{D}$ are used to index $\mathtt{L}$: if node x has degree d, the consecutive positions

$$\mathtt{L}[\mathsf{rank}_{(}(\mathtt{D},x) - 1..\mathsf{rank}_{(}(\mathtt{D},x) + d - 2]$$

store the symbols labeling the children of x. We represent $\mathtt{L}$ using the dynamic data structure for searchable partial sums with insertions and deletions, of Navarro and Sadakane [30]. This requires $n\lg k + o(n\lg k)$ bits of space, and supports searchable-partial-sum operations in $\mathcal{O}(\lg n/\lg\lg n)$ worst-case time. The overall space requirement for $\mathtt{D}$ and $\mathtt{L}$ is thus $2n + n\lg k + o(n\lg k)$ bits.

3.2 Supporting Basic Tree Operations

We use the translations of the operations on DFUDS given in Section 2.5.1 on the balanced-parenthesis operations provided in Lemma 4. This supports operations parent, child, child-rank, degree, subtree-size, preorder, and ancestor in

$\mathcal{O}(\lg n/\lg\lg n)$ worst-case time. We add also operation $\mathsf{selectnode}(i)$, which yields the DFUDS position of the node with preorder i, and can be computed using select on the balanced parentheses.

3.3 Supporting label-child and label Operations

To support operation $\mathsf{label\text{-}child}(x,\alpha)$, let $j = \mathsf{rank}_{(}(\mathtt{D},x)-1$ be the position in $\mathtt{L}$ that stores the label of the first child of node x. Let $s = \mathsf{Sum}(\mathtt{L},j-1)$ be the sum of the values stored in $\mathtt{L}$ up to position $j-1$, and let $j' = \mathsf{Search}(\mathtt{L},s+\alpha)$. If $\mathsf{Sum}(\mathtt{L},j') - \mathsf{Sum}(\mathtt{L},j-1)$ equals α, then the position j' corresponds to the position of symbol α within the labels of the children of x (otherwise, there is no child of x labeled α). Thus the child of x labeled α can be found with $\mathsf{child}(x,j'-j)$, in $\mathcal{O}(\lg n/\lg\lg n)$ extra time [30].

Operation $\mathsf{label}(x,i)$ is computed as $\mathsf{Sum}(\mathtt{L},j+i-1) - \mathsf{Sum}(\mathtt{L},j-1)$, in $\mathcal{O}(\lg n/\lg\lg n)$ time, where $j = \mathsf{rank}_{(}(\mathtt{D},x)-1$.

3.4 Supporting Updates on DFUDS

To perform $\mathsf{insert\text{-}leaf}(x,\alpha)$, we first compute the rank i of the new child of node x. Since the children of x are sorted according to the edge labels, we use α to determine i. We then compute $j = \mathsf{rank}_{(}(\mathtt{D},x)-1$ and $j' = \mathsf{Search}(\mathtt{L},s+\alpha)$. Notice that the rank of α is $i = j'-j+1$ (recall that $\mathtt{L}[j]$ stores the symbol of the first child of x). If $\mathtt{L}[j+i-1]$ represents symbol α, the new node cannot be inserted because there already exists a child of x labeled α. Otherwise, the label α must be inserted at position j' of $\mathtt{L}$. (Recall that $\mathtt{L}$ is gap encoded, hence we must be careful when inserting α to maintain this encoding.)

Next, we insert a new i-th child of x. This process is carried out by inserting a new pair of opening/closing parentheses in $\mathtt{D}$, at positions i' and i'', respectively. Positions i' and i'' are computed as follows. The insertion of the new leaf increases the degree of its parent node x. Since in DFUDS a node is represented by its degree in unary, we must increase the degree of x by adding an '`(`' at position $i' = x+\mathsf{degree}(y)-i$ within the representation of x in $\mathtt{D}$. Next we represent the new leaf node adding a '`)`' at position $i'' = \mathsf{findclose}(\mathtt{D},i')$ within $\mathtt{D}$. This shifts to the right the last '`)`' in the subtree of the $(i-1)$-th child of y, which now represents the new leaf. As the new pair of parentheses is inserted at positions i' and $i'' = \mathsf{findclose}(\mathtt{D},i')$, they are a matching pair. Hence, the insertion can be handled with the data structure of Lemma 4 in $\mathcal{O}(\lg n/\lg\lg n)$ time. Deletions are carried out in a similar fashion, decreasing the degree of node x and deleting the leaf.

3.5 Satellite Data

In order to associate b-bit satellite data to the tree nodes, for $b = \mathcal{O}(\lg n)$, we can use the *dynamic array* data structure of Lemma 3. We use the preorder numbering of the nodes to index into this array.

3.6 Main Result for Dynamic DFUDS

Combining all these structures, we obtain the following result.

Theorem 1 *There exists a dynamic* DFUDS *representation for a cardinal tree T of n nodes and alphabet size $k \leqslant n$ requiring $2n + n \lg k + o(n \lg k)$ bits of space. This representation allows us to compute operations* parent, child, label-child, child-rank, label, degree, subtree-size, preorder, ancestor, selectnode, insert-leaf, *and* delete-leaf, *all in $\mathcal{O}(\lg n / \lg \lg n)$ worst-case time. If b-bit satellite data is associated with the nodes, for $b = \mathcal{O}(\lg n)$, this representation uses $bn + o(n)$ extra bits of space. Operations* access-data *and* change-data *are supported in $\mathcal{O}(\lg n / \lg \lg n)$ amortized time.*

4 Succinct Dynamic Cardinal Trees for Moderate-sized Alphabets

We introduce in this section a succinct representation of dynamic cardinal trees of n nodes. This representation works for any $k = \mathcal{O}(n)$, yet it is able to take advantage over the data structure of Theorem 1 when k is asymptotically smaller than n (more specifically, when $\lg k = o(\lg n / \lg \lg n)$). Also, when $k = \mathcal{O}(\text{polylog}(n))$, we describe another structure in Section 5 that supports all operations in constant time. Thus the representation of introduced in this section is efficient for moderate-sized alphabet. Since we handle the small alphabets separately, in this section we assume that the alphabet size k is $\omega(\lg n)$ (to simplify the complexity terms).

4.1 Decomposition into Micro Trees

To support efficient navigation and updates of the tree, we incrementally decompose it into disjoint *micro trees* of bounded size, as nodes are inserted or deleted from the tree. This is similar to previous approaches [28,33,13]. However, in our case k can be big, and hence we face additional challenges. In particular, to support operation label-child, every node x must store (somehow) the labels of its children in a data structure that allows us to compute the rank of a given symbol among the symbols labelling the children of x [7,32,14]. Thus, we cannot use the tree-decomposition schemes of [18,13] as such. We shall first redefine the bounds for the micro-tree sizes in order to achieve the desired space and time costs, as well as define a way to determine the split point of a micro tree when it becomes too large, among other things.

Every micro tree τ of $|\tau|$ nodes represents a connected component of the original tree, such that $n_m/2 \leqslant |\tau| \leqslant n_M$, for given minimum and maximum micro-tree sizes $n_m/2$ and n_M, respectively. We arrange these micro trees in a tree by adding pointers between micro trees, and thus the entire tree is represented by a *tree of micro trees*.

Definition 1 Given a micro tree τ, we denote by $\tau_1, \ldots, \tau_{n_f(\tau)}$ its $n_f(\tau)$ child micro trees. We say that τ is *adjacent* to each τ_i, for $i = 1, \ldots, n_f(\tau)$, and viceversa.

For each internal node x of the original tree we have that x is internal to a micro tree τ, or x is a leaf of τ.

Definition 2 A node x in micro tree τ is called a *boundary node* iff x is a leaf in τ but not a leaf of the whole tree. We denote by $\mathtt{B}_\tau$ the set of boundary nodes of τ.

The j-th boundary node x (in preorder) stores a pointer to a child micro tree τ_j. This micro tree τ_j is the starting point of the subtree of node x. We duplicate the boundary node x by storing it as a fictitious root of τ_j, hence it has two representations:

1. As a leaf in micro tree τ; and
2. As the root node of the child micro tree τ_j.

These two points introduce the following property.

Property 1 Given a non-leaf node x in a micro tree τ such that $x \notin \mathtt{B}_\tau$, all the child nodes of x also belong to τ.

Note also that this property implies that sibling nodes belong all to the same micro tree. These properties, along with the fact that every micro tree is a tree by itself will be useful later to simplify the navigation on the tree. Also, it shall ensure that a micro tree can be always partitioned in the right way when it achieves the maximum allowed size. From now on, we use the subscript τ to indicate the tree operations that are local to micro tree τ. Hence, we have operations $\mathsf{insert\text{-}leaf}_\tau$, child_τ, degree_τ, and so on.

4.2 Defining Micro-Tree Sizes

Minimum Micro-Tree Size. Pointers between adjacent micro-trees must use $o(n)$ bits overall. Hence, a pointer of $w = \Theta(\lg n)$ bits must point to a micro tree of $\omega(\lg n)$ nodes. In particular, we define $n_m = \lg^3 n$. Recall that the micro trees in our data structure have at least $n_m/2$ nodes. The factor $1/2$ will be made clear in the amortized analysis of deletions, in Section 4.8.8.

Maximum Micro-Tree Size. A micro tree τ should have room to store *at least* the potential k children of the micro-tree root (recall that sibling nodes must belong to the same micro tree). Also, we must define n_M in such a way that when inserting a node in a micro tree τ of maximal size n_M (i.e., the micro tree overflows) we can split τ to create a new child micro tree τ'. It is important to note that after creating τ', both τ and τ' must have size at least $n_m/2$. Actually, we will define n_M such that micro trees can be created with at least n_m nodes. This will be needed to amortize the cost of deletions (Section 4.8.8).

By defining $n_M = 2kn_m$, the case that generates the smallest possible new micro tree τ' is as follows: the root of micro tree τ has its k possible children, the subtree of each such child having n_m nodes. Thus, upon a micro-tree overflow, any such subtrees can be detached to create the new child micro tree. The factor 2 in the definition of n_M is needed for the proof of the amortized cost of insertions (see Section 4.8.7).

Definition 3 A node z of micro tree τ is called a *splitting node* if $n_m \leqslant \mathsf{subtree\text{-}size}_\tau(z) < kn_m$. We denote $\mathtt{Split}_\tau$ the set of splitting nodes of τ.

The idea is that upon the overflow of micro tree τ, any node $z \in \mathtt{Split}_\tau$ can be used to split τ. Standard dynamic approaches based on tree decomposition [28,33, 13] use micro-tree representations which are updated using precomputed lookup tables. However, in our case micro trees are larger, and hence the precomputed tables would require a prohibitively large amount of space. Thus, we must use appropriate dynamic data structures to represent the micro trees.

4.3 The Micro Tree Layout

Every micro tree τ of $|\tau|$ nodes, having $n_f(\tau)$ child micro trees and root node r_τ is represented by an 8-tuple $\langle \mathtt{D}_\tau, \mathtt{L}_\tau, \mathtt{PTR}_\tau, \mathtt{B}_\tau, \mathtt{Split}_\tau, \mathtt{Size}_\tau, \mathtt{Data}_\tau, p_\tau \rangle$ as follows:

- $\mathtt{D}_\tau$: the tree topology of the micro tree;
- $\mathtt{L}_\tau$: the edge labels of τ;
- $\mathtt{PTR}_\tau$: the $n_f(\tau)$ pointers to child micro trees;
- $\mathtt{B}_\tau$: the $n_f(\tau)$ boundary nodes of τ;
- $\mathtt{Split}_\tau$: the splitting nodes of τ;
- $\mathtt{Size}_\tau$: the subtree size of the $n_f(\tau)$ boundary-node subtrees;
- $\mathtt{Data}_\tau[0..|\tau|-1]$: the b-bit satellite data associated to the nodes of τ; and
- p_τ: space to store the pointer to the representation of r_τ in the parent micro tree.

We show next how each of these components is represented.

4.4 Tree Topology and Edge Labels of Micro Trees

We represent the tree topology $\mathtt{D}_\tau$ and the edge labels $\mathtt{L}_\tau$ of each micro tree τ with the dynamic DFUDS data structure of Theorem 1. Hence, the tree operations within a micro tree are supported in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|)$ time. Since in our case $|\tau| \leqslant k \lg^3 n$ and $k = \omega(\lg n)$, the operations on $\mathtt{D}_\tau$ are supported in $\mathcal{O}(\lg k/\lg\lg k)$ worst-case time. The overall space requirement of the tree structure for all micro trees sums up to $2n + n\lg k + o(n\lg k)$ bits.

4.5 The Boundary Nodes of Micro Trees

The boundary $\mathtt{B}_\tau$ of τ must be represented such that we can support membership queries on it, as well as insertions and deletions (i.e., we need a dictionary data structure). In particular, we represent the boundary of τ using a *conceptual* sorted array $\mathtt{Pre}_\tau[0..n_f(\tau)]$ that stores the local preorders (i.e., the preorder within micro tree τ) of the $n_f(\tau)$ nodes in the boundary, except for $\mathtt{Pre}_\tau[0] = 0$.

Since the preorder of a node can change upon updates in $\mathtt{D}_\tau$, we need to keep $\mathtt{Pre}_\tau$ up to date. To avoid reconstructing $\mathtt{Pre}_\tau$ in linear time upon updates of τ, we represent the boundary with array $\mathtt{B}_\tau[1..n_f(\tau)]$. We gap-encode $\mathtt{B}_\tau$ such that $\mathtt{B}_\tau[i] = \mathtt{Pre}_\tau[i] - \mathtt{Pre}_\tau[i-1]$, for $i = 1, \ldots, n_f(\tau)$. Array $\mathtt{B}_\tau$ is maintained using the searchable-partial-sum data structure of Lemma 1. Thus, the j-th boundary node (in preorder) is obtained as $\mathtt{Pre}_\tau[j] \equiv \mathsf{Sum}(\mathtt{B}_\tau, j)$. Finally, for testing membership in $\mathtt{B}_\tau$, we make use of the fact that $x \in \mathtt{B}_\tau$ iff $\mathtt{Pre}_\tau[\mathsf{Search}(\mathtt{B}_\tau, \mathsf{preorder}_\tau(x))] = \mathsf{preorder}_\tau(x)$.

Analysis of the Data Structure. The total number of entries in arrays $\mathtt{B}_\tau$ equals the number of micro trees (because each boundary node stores a pointer between adjacent micro trees). Hence, the overall space requirement can be easily proved to be $o(n)$ bits.

The j-th node in the boundary (i.e., the conceptual value $\mathtt{Pre}_\tau[j]$), can be obtained in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|)$ time. Membership in $\mathtt{B}_\tau$ can also be tested in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|)$ time. Since in the worst case $|\tau| = n_M$, the time for these operations is $\mathcal{O}(\frac{\lg k+\lg\lg n}{\lg(\lg k+\lg\lg n)})$, which is $\mathcal{O}(\lg k/\lg\lg k)$ as $k = \omega(\text{polylog}(n))$. The node represented by $\mathtt{B}_\tau[j]$ can be obtained with $\mathsf{selectnode}_\tau(\mathsf{Sum}(\mathtt{B}_\tau, j))$. The total time is again $\mathcal{O}(\lg k/\lg\lg k)$.

4.6 Pointers to Child Micro Trees

In micro tree τ we store pointers to child micro trees in the *conceptual* array $\mathtt{PTR}_\tau[1..n_f(\tau)]$, sorted according to the preorders of the nodes in the boundary of τ. That is, $\mathtt{PTR}_\tau[i]$ stores a pointer to micro tree τ_i. The array functionality for $\mathtt{PTR}_\tau$ can be easily achieved by simplifying, for instance, the dynamic partial sum data structure of Lemma 1, such that only accesses to any $\mathtt{PTR}_\tau[i]$, insertions and deletions are supported.

Space Usage of Pointers. The overall number of pointers equals the number of micro trees in the structure, which is $\mathcal{O}(n/\lg^3 n)$. Thus, the overall space for pointers is $o(n)$ bits.

4.7 Satellite Data

To associate b-bit sattelite data to the tree nodes, for $b = \mathcal{O}(\lg n)$, we store a dynamic array $\mathtt{Data}_\tau[0..|\tau|-1]$ in each micro tree τ. As usual, array $\mathtt{Data}_\tau$ is indexed using operation $\mathsf{preorder}_\tau$: the data for node x in τ is stored at position $\mathtt{Data}_\tau[\mathsf{preorder}_\tau(x)]$. We use the representation of Lemma 3, using $b|\tau| + o(|\tau|)$ bits of space.

Analysis. The overall space usage of arrays $\mathtt{Data}_\tau$ over the entire tree is $bn + o(n)$ bits. According to Lemma 3, accessing the data associated to a node takes $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|) = \mathcal{O}(\lg k/\lg\lg k)$ amortized time. Inserting and deleting the data associated to a given node also takes $\mathcal{O}(\lg k/\lg\lg k)$ amortized time.

4.8 Supporting Tree Operations

We support now the basic navigation operations for our dynamic data structure.

4.8.1 Operations child*,* label-child*,* child-rank*, and* label

To compute $\mathsf{child}(x, i)$, if node x is not a leaf in $\mathtt{D}_\tau$ we simply use operation $\mathsf{child}_\tau(x, i)$ in micro tree τ (because node x and its children belong all to τ). If, on the other hand, node x is a leaf in $\mathtt{D}_\tau$, we check whether $x \in \mathtt{B}_\tau$ or not. If not, x is a leaf of the tree and hence operation child gets undefined. Otherwise, x is the root of the child micro tree τ_j, for $j = \mathsf{Search}(\mathtt{B}_\tau, \mathsf{preorder}_\tau(x))$. In this case, we follow the pointer $\mathtt{PTR}_\tau[j]$ to micro tree τ_j. Finally, $\mathsf{child}_{\tau_j}(x, i)$ on the root x of τ_j yields the i-th child of x we are looking for. Operations $\mathsf{label\text{-}child}(x, \alpha)$, $\mathsf{child\text{-}rank}(x)$, and $\mathsf{label}(x, i)$ are computed similarly, using $\mathsf{label\text{-}child}_{\tau_j}(x, \alpha)$, $\mathsf{child\text{-}rank}_{\tau_j}(x)$, and $\mathsf{label}_{\tau_j}(x, i)$, respectively.

Analysis. Because of the searchable-partial-sum operations on $\mathtt{B}_\tau$, either operation child, label-child, child-rank, and label are supported in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|) = \mathcal{O}(\lg k/\lg\lg k)$ time.

4.8.2 Operation $\mathsf{parent}(x)$

If x is not the root of micro tree τ, the operation is computed locally by using operation $\mathsf{parent}_\tau(x)$. Otherwise, we must first move to the parent micro tree τ'. In the traversal model, notice that we have arrived to τ navigating from τ'. Recall that each micro tree τ has space to store a pointer p_τ to the parent micro tree τ'. However, we only store the parent pointers for those nodes (micro tree roots) that are on the root-to-current-node path. Every time we get to micro tree τ, we store in p_τ the pointer to its parent micro tree τ', as well as the integer j such that τ is the j-th child of τ'. Every time we must move from τ to its parent τ', on the other hand, we retrieve the pointer to τ' and the j value. We then compute the position of the representation of x in the parent micro tree τ' as $\mathsf{selectnode}_{\tau'}(\mathsf{Sum}(\mathtt{B}_{\tau'}, j))$. Finally, we use $\mathsf{parent}_{\tau'}(x)$ in micro tree τ' to get the node we are looking for.

Analysis. Operation parent is supported in $\mathcal{O}(\lg k/\lg\lg k)$ time.

4.8.3 Operation $\mathsf{degree}(x)$

As in the case of parent and child operations, we first check whether node x is a boundary node of τ. If x is not a boundary node, we use return its local degree, $\mathsf{degree}_\tau(x)$. Otherwise, we first follow the pointer to the child micro tree τ_j and apply operation degree_{τ_j} on the root node of τ_j.

Analysis. The total time for operation $\mathsf{degree}(x)$ is therefore $\mathcal{O}(\lg k/\lg\lg k)$.

4.8.4 Operation $\mathsf{subtree\text{-}size}(x)$

We adapt the method of [33] to our data structure. If the subtree of x is completely contained in micro tree τ (i.e., there are no boundary nodes in the subtree rooted at x, which corresponds to the case $p_1 = p_2$, where p_1 and p_2 are defined below), we can simply return $\mathsf{subtree\text{-}size}_\tau(x)$. However, the subtree of x can span more than just one micro tree. We use here the array $\mathtt{Size}_\tau[1..n_f(\tau)]$. The idea is that $\mathtt{Size}_\tau[j]$

stores the number of nodes of the subtree pointed by $\texttt{PTR}_\tau[j]$. We represent $\texttt{Size}_\tau$ with the data structure for searchable partial sums of Lemma 1. If x does not lie in the boundary of micro tree τ, let $\texttt{Size}_\tau[p_1..p_2]$ the segment of $\texttt{Size}_\tau$ corresponding to node x, where $p_1 = \mathsf{Search}(\texttt{B}_\tau, \mathsf{preorder}_\tau(x))$ and $p_2 = \mathsf{Search}(\texttt{B}_\tau, \mathsf{preorder}_\tau(x) + \mathsf{subtree\text{-}size}_\tau(x) - 1)$. Then subtreesize can computed using:

$$\mathsf{subtree\text{-}size}(x) = \mathsf{subtree\text{-}size}_\tau(x) + \mathsf{Sum}(\texttt{Size}_\tau, p_2) - \mathsf{Sum}(\texttt{Size}_\tau, p_1 - 1) - (p_2 - p_1 + 1).$$

Note that the latter term is for subtracting the number of duplicated nodes in the portion of the boundary corresponding to the local subtree of x.

Analysis. The extra space requirement is $o(n)$ bits. After the insertion or deletion of a node, arrays $\texttt{Size}_\tau$ in the path from the tree root up to the node must be updated accordingly. As we assume the traversal model, this cost amortizes with the traversal. Operation $\mathsf{subtree\text{-}size}_\tau$ can be computed in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|)$ time according to Theorem 1. Operation Sum, on the other hand, is also computed in $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|)$ according to Lemma 1. The total time for operation $\mathsf{subtree\text{-}size}$ is thus $\mathcal{O}(\lg k/\lg\lg k)$.

4.8.5 Operation $\mathsf{preorder}(x)$

The preorder of a node x in the tree can be decomposed into three components: (1) the preorder number of the micro-tree root r_τ; (2) the preorder number of x within micro tree τ; and (3) the sum of the subtree sizes of the boundary nodes of micro tree τ whose preorder is smaller than that of x. Thus the preorder number can be computed using the following equation:

$$\mathsf{preorder}(x) = \mathsf{preorder}(r_\tau) + \mathsf{preorder}_\tau(x) + \mathsf{Sum}(\texttt{Size}_\tau, p) - p,$$

where $p = \mathsf{Search}(\texttt{B}_\tau, \mathsf{preorder}_\tau(x)) - 1$. Here, $\mathsf{preorder}(r_\tau)$ is computed incrementally, as follows. Every time a descent in the tree starts from the tree root, we set $s \leftarrow 0$. While descending, if we go to a child micro tree, say from the boundary node x' in the current micro tree τ', we do the following. First, let $p_1 \leftarrow \mathsf{Search}(\texttt{B}_{\tau'}, \mathsf{preorder}_{\tau'}(x'))$ the entry of $\texttt{Size}_{\tau'}$ corresponding to x'. Then, we update $s \leftarrow s + preorder_{\tau'}(x') + \mathsf{Sum}(\texttt{Size}_{\tau'}, p_1 - 1) - (p_1 - 1)$. Upon arriving at the root r_τ of micro tree τ, the current value of s is the preorder number of r_τ. Notice that this does not affect the time complexity of operations child and $\mathsf{label\text{-}child}$.

Analysis. The time for computing operation $\mathsf{preorder}$ is clearly $\mathcal{O}(\lg k/\lg\lg k)$.

4.8.6 Operation $\mathsf{ancestor}(x, y)$

$\mathsf{ancestor}(x, y)$ is true iff $\mathsf{preorder}(x) < \mathsf{preorder}(y) < \mathsf{preorder}(x) + \mathsf{subtree\text{-}size}(x)$. Thus, it can supported in $\mathcal{O}(\lg k/\lg\lg k)$ time using $\mathsf{preorder}$ and $\mathsf{subtree\text{-}size}$ operations.

4.8.7 Operation $\mathsf{insert\text{-}leaf}(x, \alpha)$

Assume that node x belongs to micro tree τ. Hence, we must update τ accordingly. As in Section 3.4, we first use $\texttt{L}$ to compute the rank of the new symbol α among its siblings, and then insert α into $\texttt{L}$. Then, we use the rank of α to insert the new node in $\texttt{D}_\tau$, using operation $\mathsf{insert\text{-}leaf}_\tau$.

Updating the Boundary of a Micro Tree. Let y denote the new inserted node. Note that y increments (by one) the preorders of some nodes in micro tree τ, therefore we must update the boundary of τ. Indeed, after inserting y, the preorder number of every node that whose preorder number is greater or equal to that of y must be incremented to reflect the change. The gap encoding used for $\mathtt{B}_\tau$ and the searchable-partial-sum data structure used to represent it allows us to do this efficiently.

The idea is to look for position $j = \mathsf{Search}(\mathtt{B}_\tau, \mathsf{preorder}_\tau(y))$ in $\mathtt{B}_\tau$, which corresponds to the left-most preorder to be incremented. Next, we increment $\mathtt{B}_\tau[j] \leftarrow \mathtt{B}_\tau[j] + 1$ by using operation Update. Since the preorder numbers are obtained by means of operation Sum on $\mathtt{B}_\tau$, this single increment of $\mathtt{B}_\tau[j]$ automatically increments all preorder numbers represented by positions $j' > j$ in $\mathtt{B}_\tau$. As the parent pointers actually point to the leaves of the searchable-partial-sum tree $\mathcal{T}_{\mathtt{B}_\tau}$, this also updates the parent pointers for the child micro trees of τ. The insertion cost according to this procedure is $\mathcal{O}(\lg|\tau|/\lg\lg|\tau|) = \mathcal{O}(\lg k/\lg\lg k)$ time, because of the time to update $\mathtt{B}_\tau$.

Micro-Tree Overflow. When inserting in a micro tree τ of maximal size n_M, we must first split τ into two smaller micro trees. To carry out the split, we first select a splitting node z in τ. Every node in the *local subtree* of z (i.e., the descendants of z that belong to τ) will be reinserted in a new (initially empty) child micro tree τ' (including z itself). Then, they will be deleted from τ, leaving node z still in τ. This attains the desired effect of storing node z along with its children in τ' and along with its siblings and parent in τ, keeping the properties of our data structure, as defined in Section 4.1.

When the local subtree of node z is reinserted in τ', we also copy to τ' the portions of arrays $\mathtt{B}_\tau$ and $\mathtt{PTR}_\tau$ corresponding to the subtree of node z, via insertions in $\mathtt{B}_{\tau'}$ and $\mathtt{PTR}_{\tau'}$ and the corresponding deletions in $\mathtt{B}_\tau$ and $\mathtt{PTR}_\tau$. Next, we insert a new pointer in $\mathtt{PTR}_\tau$, pointing to micro tree τ' and add node z to the boundary of τ. We also add a parent pointer in τ', pointing to the leaf corresponding to z in the tree $\mathcal{T}_{\mathtt{B}_\tau}$. In this way we keep up-to-date all of the parent pointers of the children of τ.

After the micro-tree split, the insertion of node y is carried out in the appropriate micro tree, either τ or τ'. Notice that both micro trees have room for a new node, hence no overflow is produced this time. This is important in order to upper-bound the cost of an insertion.

Maintaining the Splitting Nodes. To have a good amortized cost, the overall splitting process must be carried out in time proportional to $\mathsf{subtree\text{-}size}_\tau(z)$. This can be achieved on $\mathtt{D}_\tau$, $\mathtt{L}_\tau$, $\mathtt{B}_\tau$, and $\mathtt{PTR}_\tau$ by using the corresponding insert and delete operations. However, the question is how to efficiently choose a splitting node z. We cannot carry out a linear search in the micro tree, as for general k this cost can be bigger than $\mathsf{subtree\text{-}size}_\tau(z)$, and would not amortize the cost.

To quickly find the splitting node z, we maintain the set of splitting nodes $\mathtt{Split}_\tau$ for each micro tree τ. We represent $\mathtt{Split}_\tau$ using the same dynamic dictionary data structure as for $\mathtt{B}_\tau$, using gap encoding over a searchable-partial-sum data structure.

To maintain $\mathtt{Split}_\tau$ up-to-date, we dynamically sample nodes of $\mathtt{D}_\tau$ such that every time we need to split τ, it holds that $|\mathtt{Split}_\tau| \geqslant 1$. We must also ensure that the space for the $\mathtt{Split}_p$ data structures is $o(n)$ bits for the whole tree.

The idea is that every time we descend in the tree we maintain the last visited node z in micro tree τ such that $\mathsf{subtree\text{-}size}_\tau(z) \geqslant n_m$ holds. When we find the insertion point of the new node y, say at micro tree τ, before adding z to $\texttt{Split}_\tau$ we first carry out:

- $p_1 \leftarrow \mathsf{Search}(\texttt{Split}_\tau, \mathsf{preorder}_\tau(z))$,

and then

- $p_2 \leftarrow \mathsf{Search}(\texttt{Split}_\tau, \mathsf{preorder}_\tau(z) + \mathsf{subtree\text{-}size}_\tau(z) - 1)$.

Then, we add z to $\texttt{Split}_\tau$ whenever:

1. Node z is not the root of micro tree τ; and
2. There is no other splitting node in the subtree of z (which is true iff $p_1 = p_2$).

If across the descent we have already found a splitting node z' in $\texttt{Split}_\tau$ which is an ancestor of z, then after inserting z into $\texttt{Split}_\tau$ we delete z' from $\texttt{Split}_\tau$. In this way we keep the lowest splitting nodes z, meaning that no other node in the local subtree of z could be a splitting node by itself. This is a key to bound the maximum size that the local subtree of a splitting node can have (as the lemma below shows). Also, we avoid the case where the local subtree of a splitting node becomes too large over time, which would not guarantee a fair micro-tree split.

Lemma 5 *For every splitting node z in $\texttt{Split}_\tau$, it holds that $\mathsf{subtree\text{-}size}_\tau(z) < kn_m$.*

Proof Since we maintain the lowest splitting nodes for each node z in $\texttt{Split}_\tau$, for every descendant z' of z, $\mathsf{subtree\text{-}size}_\tau(z') < n_m$. In this way, the biggest splitting-node subtree is one where z has k children, each child having a local subtree of $n_m - 1$ nodes. Hence, this subtree has local size $1 + k(n_m - 1) < kn_m$. □

Remark 1 For every splitting node z in $\texttt{Split}_\tau$, $n_m \leqslant \mathsf{subtree\text{-}size}_\tau(z) < kn_m$.

Also, we can easily prove that after every split, the two resulting micro trees (τ and τ') have both size between n_m and n_M.

Notice that for a given splitting node z, there are no splitting nodes in its local subtree. Thus, we have a splitting node out of (at least) n_m nodes and, hence, the total space to manage them is $o(n)$ bits. We also ensure that every time a micro tree becomes full, we have at least one splitting node in $\texttt{Split}_\tau$, because every full micro tree has a local subtree of size at least n_m. Finally, choosing a splitting node from $\texttt{Split}_\tau$ takes $\mathcal{O}(\lg k / \lg \lg k)$ time.

Amortized Analysis. Summarizing, when a micro tree overflows, we choose a splitting node whose local subtree is reinserted into a new initially-empty micro tree. The reinsertion cost is proportional to the size of the reinserted subtree, which is $\mathcal{O}(\lg k / \lg \lg k)$ per reinserted node. The first time a node is reinserted into another micro tree, the reinsertion cost amortizes with the cost of the original insertion. Unfortunately, there are no bounds on the number of times a node can be reinserted. Hence, in further reinsertions of a node it is not clear how the reinsertion cost can be amortized. We use next an *accounting argument* [10] to prove that the cost of every reinsertion amortizes with the cost of inserting (for the first time) other nodes.

Let $\hat{c} = 2$ be the cost paid the first time a node is inserted into the tree. Let $c = 1$ be the actual cost of an insertion. Every insertion will spend one unit for the insertion itself and will reserve the remaining unit to pay for the future reinsertion of a node. For the analysis, let us think that every micro tree τ of the tree maintains a separate reserve R_τ. When a micro tree τ is created after a split, its reserve is set to $R_\tau \leftarrow 0$. Every time a micro tree overflows, its reserve is decreased by $I_{\tau'}$, the initial size of the new child micro tree τ'. We prove now that every time a micro tree overflows, it has enough reserves to pay for the costly operation of reinserting a set of nodes.

Let $n_m \leqslant I_\tau < kn_m$ be the initial number of nodes of a micro tree τ. The only exception is the root micro tree, where $I_\tau = 0$ holds.

Lemma 6 *Every time a micro tree τ overflows and a new child micro tree τ' of $I_{\tau'}$ nodes is created, it holds that $I_{\tau'} < R_\tau$.*

Proof When a micro tree τ overflows, its reserve is $R_\tau = n_M - I_\tau$. Because of Lemma 5, we have that $I_\tau < kn_m$ and $I_{\tau'} < kn_m$. Hence, $I_\tau + I_{\tau'} < 2kn_m = n_M$. Hence, $I_{\tau'} < n_M - I_\tau = R_\tau$. □

This proves that R_τ, the reserve of τ, is bigger than the size of the new micro tree τ'. In other words, every time we reinsert a node, its cost has been already paid by the original insertion of another node. Thus, the amortized cost of insertions is $\mathcal{O}(\lg k / \lg \lg k)$.

4.8.8 Operation delete-leaf(x, α)

To delete a leaf node $y = \mathsf{label\text{-}child}(x, \alpha)$ in micro tree τ, we update the data structure by using operation $\mathsf{delete\text{-}leaf}_\tau$. After deleting the node, we check whether there is a splitting node z in $\mathtt{Split}_\tau$ which is an ancestor of x and whose subtree becomes smaller than n_m after deleting y. As there is at most one ancestor of y in $\mathtt{Split}_\tau$, z is the node at position $\mathsf{Search}(\mathtt{Split}_\tau, \mathsf{preorder}_\tau(y)) - 1$ of $\mathtt{Split}_\tau$; the subtraction comes from the fact that with the search in $\mathtt{Split}_\tau$ we find a splitting node which is next (in preorder) to z in $\mathtt{Split}_\tau$. After deleting z from $\mathtt{Split}_\tau$, we try to insert in $\mathtt{Split}_\tau$ the last node z' found in the descent (carried out to find the deletion point) whose subtree size is at least n_m, following the same policies as for operation insert-leaf.

Micro Tree Underflow. If we delete y from a micro tree τ of size $n_m/2$, then a *micro tree underflow* occurs. In such a case, we find the representation of the micro-tree root r_τ in the parent micro tree τ', by using the parent pointer. From that node we insert all nodes of τ into τ'. Note that in the worst case there will be only one micro-tree overflow in τ' when reinserting, since τ has less than $n_m/2$ nodes, and after an overflow in τ' there will be room for *at least* n_m new nodes. If τ is not a leaf in the tree of micro trees, we insert the boundary of τ into the boundary of τ'.

Amortized Analysis. Let $\hat{d} = 2$ be the amortized cost of deletions, $d = 1$ being the actual deletion cost, i.e., every deletion uses one unit for the deletion itself and reserves the remaining unit in R_τ. Recall that micro trees are created with at least n_m nodes (Remark 1) and $R_\tau = 0$. Since underflows occur when a micro tree has

less than $n_m/2$ nodes, we ensure that at that point $R_\tau \geqslant n_m/2$ holds. Equality holds in case of micro trees that were created with n_m nodes, then received no insertions, and finally $n_m/2$ deletions are carried out on it. This means that we have enough reserves to pay for the reinsertion of $n_m/2$ nodes into the parent micro tree τ'. In τ', each reinsertion is treated as a normal insertion, increasing $R_{\tau'}$ as explained for operation insert-leaf. Overall, there are always enough reserves to pay for overflows and underflows, thus the amortized cost of deletions is $\mathcal{O}(\lg k/\lg\lg k)$.

4.9 Managing Dynamic Memory

The model of memory allocation is a fundamental issue of succinct dynamic data structures, since we must be able to manage the dynamic memory fast and without requiring much extra memory space due to memory fragmentation. Recall from Section 1.1 that we have assumed the $\mathcal{M}_B$ memory model from [33].

We manage the memory of every micro tree separately, each in a "contiguous" memory space. However, micro trees are dynamic and therefore this memory space must grow and shrink accordingly. If we use an *Extendible Array* (EA) [8] to manage the memory of each micro tree, we end up with a collection of at most $\mathcal{O}(n/\lg^3 n)$ EAs, which must be maintained under the operations:

- create, which creates a new empty EA and adds it to the collection;
- destroy, which destroys an EA from the collection;
- grow(A), which increases the size of array A by one;
- shrink(A), which shrinks the size of array A by one; and
- access(A, i), which returns the i-th item of array A.

Raman and Rao [33] show how access can be supported in $\mathcal{O}(1)$ worst-case time, create, grow and shrink in $\mathcal{O}(1)$ amortized time, and destroy in $\mathcal{O}(s'/w)$ time, where s' is the nominal size (in bits) of array A to be destroyed. (The nominal size of an array of size n where each element is stored in b bits is nb bits.) The space requirement for the whole collection is $s + \mathcal{O}(a^*w + \sqrt{sa^*w})$ bits, where a^* is the maximum number of EAs that ever existed simultaneously in the collection, and s is the nominal size of the collection.

Analysis of Time Complexity. We first analyze the time overhead introduced due to dynamic-memory management. To simplify the analysis we store every micro-tree component in different collection of EAs (i.e., we have a collection for $\mathtt{D}_\tau$, a collection for $\mathtt{L}_\tau$, and so on). The memory for $\mathtt{D}_\tau$, $\mathtt{L}_\tau$, and $\mathtt{Data}_\tau$ inside the corresponding EA is managed as in [30,31]. For the case of $\mathtt{B}_\tau$, $\mathtt{Split}_\tau$, $\mathtt{Data}_\tau$, and $\mathtt{PTR}_\tau$ we manage the corresponding EA by using standard techniques to allocate and free dynamic memory.

We use operation grow on the corresponding EA every time we insert a node in the tree, operation shrink when we delete a node, and operation create upon micro tree overflows, all of them in $\mathcal{O}(1)$ amortized time. Operation destroy, on the other hand, is used upon micro tree underflows. Consider the EA collection storing $\mathtt{L}_\tau$ for every micro tree τ of the tree. The micro tree τ' which underflows has size less than $(\lg^3 n)/2$ and thus the nominal size of the EA storing $\mathtt{L}_{\tau'}$ is less than $\Theta(\lg^3 n \cdot \lg k)$ bits. Therefore operation destroy takes $\Theta(\lg^2 n \cdot \lg k)$ time, which is negligible since we must reinsert all nodes of τ' in the parent micro tree, at a

higher total cost. Using a similar analysis, destroy on $\texttt{Data}_\tau$ takes $\mathcal{O}(\lg^3 n)$ time, which is also negligible. The EAs storing the remaining components of τ' can be destroyed even faster.

Analysis of Space Usage We analyze now the space overhead due to memory fragmentation. For the analysis, it is important to note that every time $\lg n$ changes, the tree must be rebuilt from scratch to adapt the changes. This also involves rebuilding the data structures needed to maintain the collections of EAs. The amortized cost of update operations over the tree still remains the same. Let n' be the maximum number of nodes that ever existed in the tree since the last reconstruction (i.e., the last change of $\lg n$). As reconstructions occur when n is a power of two, then both n and n' lie between (the same) two consecutive powers of two, and thus we can prove that $n \leqslant n' \leqslant 2n$ holds, which means $n' = \Theta(n)$. Thus, we can conclude that the maximum number of EAs that we can have between reconstructions is $a^* = \mathcal{O}(n/\lg^3 n)$.

The nominal size of the EA collection for $\texttt{D}_\tau$s is $2n + o(n)$ bits. Then, this collection requires $2n + o(n) + \mathcal{O}\Big(\frac{n}{\lg^2 n} + \frac{n}{\lg n}\Big) = 2n + o(n)$ bits of space [33]. The nominal size of the collection for $\texttt{L}_\tau$s is $n \lg k + o(n \lg k)$, and thus we have $n \lg k + o(n \lg k) + \mathcal{O}\Big(\frac{n}{\lg^2 n} + \frac{n}{\sqrt{\lg n}}\Big) = n \lg k + o(n \lg k)$ bits overall. For $\texttt{Data}_\tau$, the nominal size is $bn + o(n)$ bits, hence the total space usage is $bn + o(n) + \mathcal{O}\Big(\frac{n}{\lg^2 n} + \frac{n}{\sqrt{\lg n}}\Big) = bn + o(n)$ bits, for any $b = \mathcal{O}(\lg n)$.

The overall result is that $o(n \lg k)$ extra bits are used because of dynamic memory fragmentation.

4.10 The Overall Result for Moderate-sized Alphabets

Combining all these structures, we obtain the following result for $k = \omega(\lg n)$:

Theorem 2 *There exists a representation for dynamic cardinal k-ary trees on n nodes, for any $k = \omega(\lg n)$, using $2n + n \lg k + o(n \lg k)$ bits of space, supporting* parent, child, label-child, child-rank, label, degree, subtree-size, preorder, *and* ancestor, *all in $\mathcal{O}(\lg k / \lg \lg k)$ worst-case time, and* insert-leaf *and* delete-leaf *in $\mathcal{O}(\lg k / \lg \lg k)$ amortized time. If b-bit satellite data is associated with the tree nodes, for $b = \mathcal{O}(\lg n)$, this representation uses $bn + o(n)$ extra bits of space. The data of a node can be accessed/modified in $\mathcal{O}(\lg k / \lg \lg k)$ amortized time. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

5 Succinct Dynamic Cardinal Trees for Small Alphabets

We consider now the case where the size of the alphabet is small, in particular $k = (\lg n)^{\mathcal{O}(1)}$. In such a case, the data structure of Theorem 2 supports the operations in $\mathcal{O}(\lg \lg n)$ time (amortized in the case of updates). We introduce next a succinct representation for cardinal trees which also uses $2n + n \lg k + o(n \lg k)$ bits and is able to squeeze the operation times to $\mathcal{O}(1)$ when $k = (\lg n)^{\mathcal{O}(1)}$.

The basic structure is similar to that of Section 4. The input tree is decomposed into disjoint micro trees. Each operation is performed within the micro tree that

contains the current node of the traversal, and in the case of the navigational operations, we might traverse to an adjacent micro tree. The micro tree representation of Section 4 supports the operations in logarithmic time (for general alphabets). We improve here the time to $\mathcal{O}(1)$ for small alphabets.

5.1 Tree Decomposition

We use the greedy decomposition algorithm of [28] to decompose the input tree to micro trees of size in the range $[\lg^2 n..k^2 \lg^2 n]$. The micro tree containing the root might be smaller than $\lg^2 n$. This algorithm performs a postorder traversal of the tree. During the traversal, every at least $\lg^2 n$ visited nodes make a micro tree (see [28] for more details). We modify the algorithm of [28] slightly to satisfy the conditions and the properties of Section 4.1.

5.2 Micro Tree Representation

Every micro tree τ of $|\tau|$ nodes, having $n_f(\tau)$ child micro trees and root node r_τ is represented by a 7-tuple $\langle \mathtt{D}_\tau, \mathtt{L}_\tau, \mathtt{PTR}_\tau, \mathtt{B}_\tau, \mathtt{Size}_\tau, \mathtt{Data}_\tau, p_\tau \rangle$, defined as in Section 4.

Notice that, unlike the representation of Section 4, here we do not have an explicit set of splitting nodes in the micro trees. This is because this time the micro trees have $\Theta(\text{polylog}(n))$ nodes and, upon overflows, micro trees will be split into two micro trees whose size is also $\Theta(\text{polylog}(n))$. Hence, splitting nodes can be sought by traversing the micro tree, with no time penalties.

For each of the seven micro-tree components, we make data structures to perform the corresponding operations on them efficiently. The space usage of these data structures for a micro tree τ is $2|\tau| + |\tau| \lg k + o(|\tau| \lg k)$ bits, which makes the total space usage $2n + n \lg k + o(n \lg k)$ bits since the micro trees are roughly disjoint (they only intersect each other at the boundary nodes). In the following, we describe all the seven components of the micro-tree representations.

5.3 Tree Topology and Edge Labels of Micro Trees

We present $\mathtt{D}_\tau$ and $\mathtt{L}_\tau$, the first two parts of the 7-tuple that represents a micro tree τ. The data structure $\mathtt{D}_\tau$ represents the topology of τ using $2|\tau| + o(|\tau|)$ bits and supports all the tree operations except label-child, insert-leaf, and delete-leaf, within τ in $O(1)$ time. This data structure is able to insert and delete nodes in τ, however it cannot perform insert-leaf and delete-leaf since these two operations involve edge labels which are not stored by $\mathtt{D}_\tau$. Thus, we also design the dynamic data structure $\mathtt{L}_\tau$ which represents the edge labels of τ using $|\tau| \lg k + o(|\tau| \lg k)$ bits and supports label-child in $O(1)$ time. The two update operations insert-leaf and delete-leaf can be then performed using both data structures $\mathtt{D}_\tau$ and $\mathtt{L}_\tau$ in $O(1)$ amortized time. In the following, we first describe $\mathtt{D}_\tau$ and then $\mathtt{L}_\tau$:

5.3.1 $\mathtt{D}_\tau$*: Representation of the Topology of Micro Trees*

We represent the topology of a micro tree τ by its DFUDS sequence. More specifically, we maintain $\mathtt{D}_\tau$ as a dynamic DFUDS representation of τ. Recall from Section

2.5 that a DFUDS representation of a static ordinal tree with n nodes supports the navigational and query operations on the tree in constant time using $2n + o(n)$ bits. Such a data structure consists of the DFUDS sequence of the tree in addition to sub-structures that support rank, select, findclose, findopen, and enclose operations on the sequence. Here, we show how to dynamize such a DFUDS representation for a small enough ordinal tree, which is then used as $\mathtt{D}_\tau$. The operations insert-leaf and delete-leaf, in an ordinal tree, respectively insert and delete a leaf that is the i-th child of a node for a given i. The following lemma states this result:

Lemma 7 *An ordinal tree τ, where $|\tau| = \text{polylog}(n)$, can be maintained using a dynamic* DFUDS *representation of size $2|\tau| + o(|\tau|)$ bits that supports the operations* parent*,* child*,* degree*,* subtree-size*,* ancestor*,* preorder*,* insert-leaf*, and* delete-leaf *in $\mathcal{O}(1)$ time (updates are amortized). The structure assumes access to precomputed tables of size $o(n)$ bits, which are independent of τ.*

As described in Section 2.5, a DFUDS representation can be obtained by constructing a data structure supporting operations rank, select, findclose, findopen, and enclose on the DFUDS sequence. In our application, the length of the DFUDS sequence is polylog(n). We note that the update operations insert-leaf and delete-leaf on the ordinal tree can be also supported by inserting and deleting a pair of matching parentheses in the DFUDS sequence (see Section 3.4). In the following, we describe how we handle the problem of constructing a dynamic data structure for balanced parentheses.

Dynamic data structure for balanced parentheses. We briefly describe the data structure of [9, Section 5] that maintains a sequence of $m \leqslant n$ pairs of balanced parentheses and supports the operations findclose, findopen, enclose, and insertion and deletion of a pair of matching parentheses.

We divide the sequence into chunks of size ℓw bits, and construct a B-tree with branching factor b, for some parameters ℓ and b to be chosen later. The i-th leaf of the B-tree maintains the i-th chunk from left to right. So, the number of leaves is $2m/\ell w$, the number of internal nodes is $\mathcal{O}(2m/\ell w b)$, and the height of the tree is $\mathcal{O}(\lg_b m)$.

At each internal node u, seven arrays are maintained where each array contains b numbers of size $\mathcal{O}(\lg m)$ bits. The i-th item of these seven arrays stores the following information about the parenthesis chunks located in the i-th subtree of u (numbers refer to each of the seven arrays): 1) the number of parentheses, 2) the number of unmatched close parentheses, 3) the number of unmatched open parentheses, 4) the number of unmatched close parentheses whose matching parentheses are located in a subtree rooted at some other child of u, 5) the number of unmatched open parentheses whose matching parentheses are located in a subtree rooted at some other child of u, 6) the number of unmatched close parentheses whose matching parentheses are located outside the subtree, 7) the number of unmatched open parentheses whose matching parentheses are located outside the subtree. These seven array are used to perform the parenthesis operations. The size of each array is $\mathcal{O}(b \lg m)$ bits. Each array is represented by a searchable partial sums data structure.

It has been shown that all the parenthesis operations and updates on the input sequence can be performed by following appropriate root-to-leaf paths in the B-tree

and using or updating the chunks at the leaves and the seven arrays at the internal nodes [9]. The running times of the operations are affected by the following: 1) the height of the tree $\mathcal{O}(\lg_b m)$ since each operation takes a root to leaf path, 2) the size of each chunk which is ℓw bits and thus insertion and deletion take $\mathcal{O}(\ell)$ time, and 3) the running time of the operations of the searchable partial sums data structure denoted by t. Therefore, the total running time will be $\mathcal{O}(\lg_b m + \ell + t)$.

The total space of the data structure is $2m$ bits for storing all the chunks plus $\mathcal{O}(2m/\ell wb \cdot s)$ bits for storing searchable partial sums data structures of size $\mathcal{O}(s)$ bits for the seven arrays at each internal node. Therefore, the total space is $2m + \mathcal{O}(2m/\ell wb \cdot s)$ bits.

For the case where $m = n$, a result was achieved in [9] by doing the following: 1) choosing ℓ from the range $[\lg n/\lg\lg n \ldots 2\lg n/\lg\lg n]$, 2) choosing b from the range $[\sqrt{\lg n}/2 \ldots \sqrt{\lg n}]$, and 3) using a searchable partial sums structure from [31], which supports operations in $t = \mathcal{O}(1)$ time using $s = \mathcal{O}(b \lg m)$ bits for each internal node besides extra precomputed tables of size $o(n)$ bits. The following lemma states this result:

Lemma 8 *([9]) A sequence of n pairs of balanced parentheses can be maintained in a data structure of size $2n + o(n)$ bits that supports* findclose, findopen, enclose, *and insertion/deletion of a pair of matching parentheses in $\mathcal{O}(\lg n/\lg\lg n)$ time.*

We obtain a result for the case where $m = \text{polylog}(n)$. For this data structure, we also use the searchable partial sums structure of [31]; we chose ℓ from the range $[\sqrt{\lg n} \ldots 2\sqrt{\lg n}]$; and we chose b from the range $[\lg^{1/4} n \ldots 2\lg^{1/4} n]$. In order to improve the running time of the parentheses operations and updates to $\mathcal{O}(1)$, we store each of the chunks in a dynamic array of Lemma 2. This dynamic array allows us to access, insert, or delete a parenthesis at a given index of a chunk in $\mathcal{O}(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits. This is an improvement over the data structure of [9] which inserts and deletes at each chunk in $\mathcal{O}(\ell)$ time. Although this improvement comes at the cost of increasing the space of representing each chunk from ℓw bits to $\ell w + \mathcal{O}(\ell \lg \ell)$ bits, this only increases the total space by a negligible amount: $2m/\ell w(\ell w + \mathcal{O}(\ell \lg \ell)) + o(m) = 2m + o(m)$ bits. The following lemma states our result:

Lemma 9 *A sequence of m pairs of balanced parentheses, where $m = \text{polylog}(n)$, can be maintained in a data structure of size $2m + o(m)$ bits which supports the operations* rank, select, findclose, findopen, *and* enclose *in $\mathcal{O}(1)$ time, and can insert and delete a pair of matching parentheses at a given location in $\mathcal{O}(1)$ amortized time. The structure assumes access to precomputed tables of size $o(n)$ bits.*

Proof The data structure is the one described above except for the operations rank and select. For these two operations, we add another array to the seven arrays stored at each internal node u of the B-tree. This array also maintains b items, where the i-th item is the number of open parentheses in the chunks that are descendants of the i-th child of u. This array is also represented by the searchable partial sums data structure of [31] which allows to support rank and select in constant time. □

5.3.2 Edge Labels of Micro Trees

Let $\mathtt{L}_\tau$ be the sequence containing all the edge labels of τ, in the same order as in Section 2.5. To perform label-child(x, α) on τ, we find the rank i of α among all the edge labels between the current node and its children, and then we use child(x, i) to find the required child. To find i, we find the number of occurrences of α before the current node, and then find the position of the next occurrence of α using a rank/select data structure on both $\mathtt{D}_\tau$ and $\mathtt{L}_\tau$. To perform insert-leaf(x, α), we again need to find i to simply insert the label. But finding i if there is no α among all the edge labels needs more information. To support this, we construct a dynamic predecessor structure for all the edge labels below each internal node.

Note that $\mathtt{L}_\tau$ consists of contiguous sub-sequences s_i, for $i = 1, \ldots, I_\tau$, such that s_i represents all the labels below the i-th internal node of τ in preorder, where I_τ is the number of internal nodes in τ. Note also that $|s_i| \leqslant k$. We construct the following: (1) a data structure that supports the operations rank, select, insertions and deletions on $\mathtt{L}_\tau$, (2) a data structure for each s_i, if $|s_i| > \lg n / \lg\lg n$, which supports the operations predecessor, insertions and deletions on s_i. In the following, we explain these two structures, and then we combine them.

Dynamic rank/select *Data Structure* In the following lemma, we present a data structure which is used to perform label-child(x, α) in a micro tree.

Lemma 10 *There exists a dynamic representation of size $m \lg k + o(m \lg k)$ bits for a sequence $\mathtt{L}$ of m symbols from an alphabet of size k using precomputed tables of size $o(n)$ bits, where m and k are $(\lg n)^{\mathcal{O}(1)}$. This data structure supports the operations* rank *and* select *in $\mathcal{O}(1)$ time, and supports inserting and deleting symbols at arbitrary positions in the sequence in $\mathcal{O}(1)$ amortized time. The structure assumes access to precomputed tables of size $o(n)$ bits.*

Proof There exists a static data structure that supports the operations rank and select in $\mathcal{O}(1)$ time for an alphabet of size k, using a multi-ary wavelet tree with $\mathcal{O}(1)$ height (Theorem 3.2 of [16]). We dynamize their structure in the following way. We set the branching factor of their wavelet tree to be $k' = \sqrt{\lg n}$. At each internal node we use a dynamic rank/select data structure for an alphabet of size k'. In the following, we explain this data structure. Note that the update operations do not change the structure of the wavelet tree, and thus only the internal node structures should be dynamized.

We pack every ℓ symbols of the sequence into a chunk of size $\ell \lg k'$ bits, where $\ell = (w / \lg k') \lg^{1/4} n$. Each chunk is represented by a dynamic array of size $\ell \lg k' + \mathcal{O}(\lg^{1/4} n \lg \ell)$ bits, which allows us to access, insert, or delete a symbol at a given index in $\mathcal{O}(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits (see Lemma 2). Therefore, the total space used for the chunks is $m \lg k' + o(m \lg k')$ bits.

Now, we make a B-tree with branching factor $\lg^{\frac{1}{4}} n$. Each leaf of the B-tree stores a pointer to a sub-chunk of size w bits in one of the chunks such that scanning the sub-chunks of the leaves from the left to right in the B-tree gives the original sequence. Therefore, each chunk corresponds to $\lg^{1/4} n$ leaves. The number of leaves is $m/(\ell \lg^{1/4} n)$ and the depth of the B-tree is $\mathcal{O}(1)$. At each internal node u, we maintain $k' + 1$ arrays, each of length $\lg^{1/4} n$. One of the arrays

is denoted by S. The i-th element of the array S maintains the number of symbols in the sub-chunks that are descendants of the i-th child of u. Each of the other k' arrays is for a symbol in the alphabet, and its i-th element maintains the number of occurrences of the corresponding symbol in the leaves that are descendants of the i-th child of u. We represent each of these arrays by a searchable-partial-sum data structure with $\mathcal{O}(1)$ time for the partial sums operations, using a precomputed table of size $o(n)$ bits, since the arrays are small (i.e., $\mathcal{O}(\lg^{\frac{1}{4}} n \cdot \lg\lg n)$ bits).

To perform the operation $\mathsf{rank}_\alpha(\mathtt{L}, i)$, we traverse the B-tree top-down starting from the root. Let h be the sub-chunk containing the i-th symbol of the original sequence. At each internal node u, we count the number of occurrences of α in the sub-chunks that are to the left of h, and are descendants of u. This counting can be performed in $\mathcal{O}(1)$ time, using the partial sums structures that are constructed for the array S and the array corresponding to α. At the leaf level, where we should perform rank in a sub-chunk of size w bits, we read the sub-chunk in $\mathcal{O}(1)$ time and perform the rank using precomputed tables. The operation $\mathsf{select}_\alpha(\mathtt{L}, j)$ can be performed similarly in $\mathcal{O}(1)$ time (array S is not required for select).

For insertions and deletions in $\mathtt{L}$ we perform them on the appropriate chunks in $\mathcal{O}(1)$ amortized time (with the support of the dynamic arrays), and then we update the nodes of the B-tree along the appropriate path in a straightforward manner. Therefore, the total update time is $\mathcal{O}(1)$ amortized. □

Dynamic Predecessor. In the following lemma, we present a structure to find the rank of α among its siblings, used in $\mathsf{insert\text{-}leaf}(x, \alpha)$.

Lemma 11 *There exists a dynamic predecessor data structure for a sorted array of m elements, where $m = (\lg n)^{\mathcal{O}(1)}$ and each element is from the range $[0 \dots k-1]$, that uses $o(m)$ additional bits. Assuming access to a precomputed table of size $o(n)$ bits, this data structure supports the operation* predecessor *in $\mathcal{O}(1)$ time, and supports insertions and deletions in $\mathcal{O}(1)$ amortized time.*

Proof For this structure, we use the same packing strategy and dynamic arrays as we used in the proof of Lemma 10. We make a B-tree with branching factor b, where $b = \sqrt{\lg n}$. Each leaf maintains b elements from the array, such that concatenating the leaves from left to right, gives the original array. The height of the tree is $\mathcal{O}(1)$. At each internal node, we maintain b guiding indexes. Every node (including leaves) has $b \lg k = o(\lg n)$ bits which can be handled using a precomputed table of size $o(n)$ bits. To perform the operations, we traverse the tree top-down in $\mathcal{O}(1)$ time. For the update operations, we also update the internal nodes in a bottom-up traversal. The rebalancing is applied as needed. □

The following lemma combines Lemma 10 and Lemma 11, and shows how to perform the operation $\mathsf{child}(x, \alpha)$ on τ using the data structures for D_τ and L_τ.

Lemma 12 *For a k-ary cardinal tree τ of at most $k^2 \lg^2 n$ nodes, for $k = (\lg n)^{\mathcal{O}(1)}$, there exists a dynamic representation of size $2|\tau| + |\tau| \lg k + o(|\tau| \lg k)$ bits that supports the operation* $\mathsf{label\text{-}child}_\tau$ *in $\mathcal{O}(1)$ time, and supports the update operations* $\mathsf{insert\text{-}leaf}_\tau$ *and* $\mathsf{delete\text{-}leaf}_\tau$ *in $\mathcal{O}(1)$ amortized time. The structure uses precomputed tables of size $o(n)$ bits.*

Proof As in Section 4, we represent the tree τ with $\mathtt{D}_\tau$ and $\mathtt{L}_\tau$. We construct a data structure for each of $\mathtt{D}_\tau$ and $\mathtt{L}_\tau$ using Lemmas 7, 10, and 11 using a total of $2|\tau| + |\tau| \lg k + o(|\tau| \lg k)$ bits. □

5.4 Boundaries of Micro Trees

While supporting the operations on a micro tree τ, we need to check whether the current node is a boundary node of τ or not. In case it is a boundary node, we may need to move the current node to the corresponding adjacent micro tree by following a pointer to the child micro tree . To efficiently check whether a node is a boundary node, we store the boundary nodes of τ in an array $\mathtt{B}_\tau$ of $n_f(\tau)$ elements. The representation of pointers is explained in Section 5.5. The i-th element of the array $\mathtt{B}_\tau$ contains the difference between two preorder numbers which belong to the i-th and $(i+1)$-st boundary nodes of τ in the preorder traversal of τ.

We represent $\mathtt{B}_\tau$ using a searchable partial sums structure. Raman et al. [31] gave a data structure that solves the problem (to represent m r-bit integers) for $m = w^\epsilon$ and $r \leqslant w$, for any fixed $0 \leqslant \epsilon < 1$. Their data structure achieves $\mathcal{O}(1)$ time for operations Sum, Update, and Search, and uses $\mathcal{O}(mw)$ bits of space. We show that when both m and r are $\mathcal{O}(w^c)$, for any constant $c > 0$, we can obtain a data structure with $\mathcal{O}(1)$ time for all the operations, using $m \lg r + o(m \lg r)$ bits of space.

Lemma 13 *For any integer $n < 2^w$, there exists a searchable-partial-sum data structure to represent an array $A[1..m]$ whose elements are in the range $[0..r-1]$, using $m \lg r + o(m \lg r)$ bits, where m and r are $(\lg n)^{\mathcal{O}(1)}$. This data structure supports the operations* Sum, Update, *and* Search *in $\mathcal{O}(1)$ time, assuming access to precomputed tables of size $o(n)$ bits.*

Proof We pack every $w/\lg r$ elements of the array into a word. Within each word, every b numbers denote a chunk, where $b = \lg^{1/4} n$. Within each chunk, the operations can be supported in $\mathcal{O}(1)$ time using a precomputed table of size $o(n)$ bits. The space usage to store all the chunks is $m \lg r + o(m \lg r)$ bits.

Now, we make a B-tree with branching factor at most b. Each leaf of the B-tree stores a pointer to one of the chunks such that scanning the chunks of the leaves from left to right in the B-tree gives the original array. The number of leaves is m/b and the depth of the B-tree is $\mathcal{O}(1)$. At each internal node u, we maintain two arrays of length b. The i-th element of the first array maintains the sum of all the elements in the chunks that are descendants of the i-th child of u. The i-th element of the second array maintains the number of all the elements in the chunks that are descendants of the i-th child of u. The operations on these two arrays can be supported in $\mathcal{O}(1)$ time, using a precomputed table of size $o(n)$ bits. Since the number of internal nodes is $\mathcal{O}(m/b^2)$, the space usage for the B-tree is $\mathcal{O}((m/b^2) \cdot (b(\lg r + \lg m))) = o(m \lg r)$ bits.

The operations on the input array can be carried out by traversing the tree top-down and computing the operations at the internal nodes in $\mathcal{O}(1)$ time. □

Since each element of $\mathtt{B}_\tau$ takes $\mathcal{O}(\lg |\tau|)$ bits, the searchable partial sums data structure takes $n_f(\tau) \lg |\tau| + o(n_f(\tau) \lg |\tau|)$ bits. Thus the overall space for all the micro trees is $o(n)$ bits. To check whether the current node is a boundary node or not, we use Search on $\mathtt{B}_\tau$ for the preorder number of the current node.

5.5 Pointers Between Adjacent Micro Trees

There are two cases where we need to traverse from the micro tree τ containing the current node x, to an adjacent micro tree: (1) if x is a boundary node of τ, and we need to follow a pointer to the micro tree rooted at x, and (2) if x is the root of τ, and we need to follow a pointer to move to the parent micro tree of τ.

In the first case, for each boundary node x of τ, we store a pointer to the micro tree rooted at x. These pointers are represented in the following way. Let τ_i be the micro tree rooted at $\mathtt{B}_\tau[i]$, the i-th boundary node of τ. We make an array $\mathtt{PTR}_\tau$ of $n_f(\tau)$ elements such that $\mathtt{PTR}_\tau[i]$ maintains a pointer to τ_i. Therefore, whenever the current node is $\mathtt{B}_\tau[i]$ (we can check this using the representation of Section 5.4), we can traverse to the root of the micro tree τ_i. The space usage to store $\mathtt{PTR}_\tau$ for all the micro trees is $o(n)$ bits. For parent pointers, we use the same approach as in Section 4.8.2.

5.6 Satellite Data

Similar to Section 4.7, we store all the b-bit data associated to the nodes of each micro tree in a dynamic array, where the location of the data associated with each node is determined by the the preorder number of the node in the corresponding micro tree. The number of dynamic arrays is equal to the number of micro trees; the length of each dynamic array is equal to the size of each micro tree which is $(\lg n)^{\mathcal{O}(1)}$; and the size of the data associated to each node is $b = \mathcal{O}(\lg n)$ bits.

We utilize the dynamic array of Lemma 2 which maintains an array of length $w^{\mathcal{O}(1)}$ containing $\mathcal{O}(w)$-bit elements. For a micro tree τ, this lemma implies a data structure of size $b|\tau| + \mathcal{O}(k_\tau \lg\lg n)$ bits with $\mathcal{O}(1)$ query time and $\mathcal{O}(|\tau|/k_\tau)$ amortized update time, where $k_\tau \leqslant |\tau|$ is a parameter. By setting $k_\tau = c|\tau| \lg\lg n/\lg n$ for small enough constant c, we obtain a data structure of size $bn + o(n)$ bits in total, with $\mathcal{O}(1)$ query time and $\mathcal{O}(\lg n/\lg\lg n)$ amortized update time (similar to Section 4.7). Moreover, we can also obtain another trade-off by setting $k_{|\tau|} = c|\tau|$, for small enough constant c, to achieve $\mathcal{O}(1)$ query time and $\mathcal{O}(1)$ amortized update time using $bn + \mathcal{O}(n \lg\lg n)$ bits.

5.7 Supporting Tree Operations

5.7.1 Subtree Sizes

We construct a data structure that allows us to compute the subtree size of the current node in $\mathcal{O}(1)$ time. Let τ be the micro tree containing the current node. Lemma 7 shows that we can perform subtree-size on the current node within τ in $\mathcal{O}(1)$ time. But, to this number, we should add the subtree sizes of the roots of all the child micro trees of τ that are descendants of the current node (similar to the algorithm described in Section 4.8.4). For this, we make an array $\mathtt{Size}_\tau$ of length $n_f(\tau)$ such that $\mathtt{Size}_\tau[i]$ maintains the subtree size of the root of τ_i, where τ_i is the child micro tree rooted at the i-th boundary node of τ in the preorder traversal of τ. We represent $\mathtt{Size}_\tau$ by a searchable-partial-sum data structure using $n_f(\tau) \lg n + o(n_f(\tau) \lg n)$ bits (see Lemma 13). The overall space for all the micro trees is $o(n)$

bits. To compute the subtree size, we need to find $\sum_{i=j_\ell}^{j_r} \texttt{Size}_\tau[i]$, where τ, τ_{j_ℓ} and τ_{j_r} are the left-most and right-most child micro trees of τ, respectively, that are descendants of the current node. To find τ_{j_ℓ}, we do a predecessor search in the array $\texttt{B}_\tau$ for the preorder number of the current node. Let e be the left-most leaf of τ that is also a descendant of the current node. To find τ_{j_r}, we first find the preorder number of e within τ by adding the preorder number of the current node and its subtree size within τ. Then we do a predecessor search in the array $\texttt{B}_\tau$ for the preorder number of e.

5.7.2 Operation insert-leaf(x, α)

To perform insert-leaf(x, α) in a micro tree τ, we update the representation of τ in the following way. We update $\texttt{D}_\tau$ by inserting a pair of matching parentheses as in Section 3.4, using the rank i of the new symbol α. In order to compute i, we use the predecessor search structure that we make for each contiguous sub-sequence of $\texttt{L}_\tau$ (see Section 5.3.2). First we find the sub-sequence in $\texttt{L}_\tau$ that contains the edge-labels of the node x using the preorder number of x within τ. We then perform a predecessor search for α within the found sub-sequence to determine i. We update $\texttt{L}_\tau$ by inserting α into $\texttt{L}_\tau$ at position i. The new leaf is not a boundary node, but if it is inserted between two boundary nodes, then it changes the difference between their preorder numbers. Therefore, we increment the appropriate element of $\texttt{B}_\tau$. All the above operations can be performed in $\mathcal{O}(1)$ time.

If $|\tau|$ exceeds the value of $k^2 \lg^2 n$ (i.e., the micro tree τ overflows), we split τ into several micro trees whose sizes are in the range $[2 \lg^2 n \ldots 2k \lg^2 n]$ using the decomposition algorithm that we used in Section 5.1. Then we reconstruct the representation of each new micro tree. This can be performed by inserting leaves one by one into the new micro trees. The split and the construction of micro tree representations can both be performed in $\mathcal{O}(|\tau|) = \mathcal{O}(k^2 \lg^2 n)$ time. Since this procedure makes micro trees of small enough size (at most $2k \lg^2 n$), therefore, $O(k^2 \lg^2 n)$ number of insert-leaf operations are required to make any of them full, and hence the insertion time is $\mathcal{O}(1)$ amortized.

5.7.3 Operation delete-leaf

To perform delete-leaf(x, α) in a micro tree τ, we update the representation of τ similar to the algorithm for insert-leaf(x, α). If $|\tau|$ becomes smaller than $\lg^2 n$, then we combine τ with its parent micro tree τ'. This can be performed by inserting the nodes of τ in preorder into τ' using a total of $\lg^2 n$ insert-leaf operations, which takes $\mathcal{O}(|\tau|) = \mathcal{O}(\lg^2 n)$ time. In the amortized analysis, we charge the cost of each of these insert-leaf to a delete-leaf that will occur in τ' later on. This increases the amortized cost of delete-leaf by only a constant factor.

5.7.4 Memory Management

We store each micro tree in a separate location of the memory using an extendible array [8]. Since the number of micro trees is at most $n/\lg^2 n$, and the nominal size of all the micro trees is $s = 2n + n \lg k + o(n \lg k)$ bits, then the space requirement for the whole collection of micro trees is $s + \mathcal{O}(nw/\lg^2 n + \sqrt{snw/\lg^2 n}) = 2n + n \lg k + o(n \lg k)$ bits [34].

5.8 The Overall Result for $k = \mathcal{O}(\text{polylog}(n))$

The following theorem states our result for small-alphabet dynamic cardinal trees.

Theorem 3 *There exists a representation for dynamic cardinal k-ary trees of n nodes using $2n + n\lg k + o(n \lg k)$ bits of space, where $k = (\lg n)^{\mathcal{O}(1)}$, supporting operations* parent, child, label-child, child-rank, label, degree, subtree-size, preorder, *and* ancestor, *all in $\mathcal{O}(1)$ worst-case time. Operations* insert-leaf *and* delete-leaf *are supported in $\mathcal{O}(1)$ amortized time. If b-bit satellite data is associated with the tree nodes, for $b = \mathcal{O}(\lg n)$, we provide the following trade-offs:*

1. *$bn + o(n)$ extra bits of space, operation* access-data *in $\mathcal{O}(1)$ time, and operation* change-data *in $\mathcal{O}(\lg n / \lg\lg n)$ amortized time;*
2. *$bn + \mathcal{O}(n \lg\lg n)$ extra bits of space, operation* access-data *in $\mathcal{O}(1)$ time, and operation* change-data *in $\mathcal{O}(1)$ amortized time; and*
3. *$bn + o(n)$ extra bits of space, operations* access-data *and operation* change-data *in $\mathcal{O}(\lg\lg n / \lg\lg\lg n)$ amortized time.*

The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.

6 Succinct Dynamic Binary Trees

We now use the machinery we have developed in the previous sections to obtain improved succinct representations for dynamic binary trees [28,33,13].

We represent a binary tree T_b of n nodes using the following bijection between T_b and a corresponding ordinal tree T_o [27]. We define T_o with $n+1$ nodes, such that each node x of T_b corresponds to a node $t(x)$ in T_o, except for a dummy root that we must add to T_o. The root of T_b corresponds to the first (and only) child of the dummy root of T_o. The left child of node x in T_b corresponds to the first child of $t(x)$ in T_o, and the right child of node x in T_b corresponds to the next sibling of $t(x)$ in T_o. Since ordinal trees can be represented using balanced parentheses, this representation of binary trees uses $2n + o(n)$ bits, as desired. Navigational operations on T_b are supported by balanced-parenthesis operations on T_o in $\mathcal{O}(1)$ time.

We dynamically divide the binary tree into micro trees as in Section 5. Each micro tree τ having $n_f(\tau)$ child micro trees is represented by a 6-tuple $\langle \mathtt{D}_\tau, \mathtt{PTR}_\tau, \mathtt{B}_\tau, \mathtt{Size}_\tau, \mathtt{Data}_\tau, p_\tau \rangle$. These have the same meaning as in Sections 4 and 5, except $\mathtt{D}_\tau$, which now is the ordinal-tree representation of the binary micro tree. We use the representation of Lemma 9 for $\mathtt{D}_\tau$, to support the ordinal-tree operations in $\mathcal{O}(1)$ time. Then, each binary-tree operation can be simulated as follows:

- Operation left-child(x) in T_b is computed as first-child$(t(x))$ in T_o.
- Operation right-child(x) in T_b is computed as next-sibling$(t(x))$ in T_o.
- Operation parent(x) in T_b is computed according to the following two cases: if $t(x)$ is the first child of its parent in T_o, then we use parent$(t(x))$ in T_o; otherwise, we use prev-sibling$(t(x))$ in T_o.
- Operation subtree-size(x) in T_b is computed adding subtree-size$(t(x))$ (which corresponds to the subtree of the left child of x in T_b, plus x itself) plus the size of the subtrees of all siblings to the right of $t(x)$ in T_o (which correspond to the

subtree of the right child of x in T_b). Let ℓ be the rightmost leaf in the subtree of node $\mathsf{parent}(t(x))$ in T_o. Hence, $\mathsf{preorder}(t(x)) - \mathsf{preorder}(\ell)$ in T_0 equals $\mathsf{subtree\text{-}size}(x)$ in T_b.
- Operations preorder and ancestor are carried out as in Sections 4 and 5.
- Operations insert-leaf and delete-leaf are carried out inserting/deleting a '`()`' at the corresponding position in T_o.
- Finally, satellite data can be supported as in Section 5.

Since the balanced-parentheses operations within the mini trees are supported in $\mathcal{O}(1)$ time (see Lemma 9), and each binary-tree operation can be simulated with $\mathcal{O}(1)$ balanced-parentheses operations, and the navigation between adjacent micro-trees can be also supported in $\mathcal{O}(1)$ time as in Section 5, we have proved:

Theorem 4 *There exists a representation for dynamic binary trees of n nodes using $2n + o(n)$ bits of space and supporting operations* parent, left-child, right-child, subtree-size, preorder, *and* ancestor, *all in $\mathcal{O}(1)$ worst-case time. Operations* insert-leaf *and* delete-leaf *are supported in $\mathcal{O}(1)$ amortized time. If b-bit satellite data is associated with the tree nodes, for $b = \mathcal{O}(\lg n)$, we provide the following trade-offs:*

1. *$bn + o(n)$ extra bits of space, operation* access-data *in $\mathcal{O}(1)$ time, and operation* change-data *in $\mathcal{O}(\lg n / \lg \lg n)$ amortized time;*
2. *$bn + \mathcal{O}(n \lg \lg n)$ extra bits of space, operation* access-data *in $\mathcal{O}(1)$ time, and operation* change-data *in $\mathcal{O}(1)$ amortized time; and*
3. *$bn + o(n)$ extra bits of space, operations* access-data *and operation* change-data *in $\mathcal{O}(\lg \lg n / \lg \lg \lg n)$ amortized time.*

The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.

7 Conclusions and Future Work

Succinct data structures have become crucial to store and manage big amounts of data in many applications. Specially, succinct dynamic data structures have particular importance, as they also allow efficient updates. In this paper, we have introduced succinct representation for dynamic k-ary cardinal trees (or tries) on n nodes. These are fundamental data structures for text-processing algorithms. Our cardinal tree representations require $2n + n \lg k + o(n \lg k)$ bits of space, which is close to $\mathcal{C}(n, k) \approx n(\lg k + \lg e) + o(n + \lg k)$, the lower bound for representing a k-ary cardinal tree on n nodes.

Ours are the first dynamic cardinal tree representations that support a complete set of operations, while using almost optimal space. For $k = \mathcal{O}(\text{polylog}(n))$, we support navigation and query operations on the tree in $\mathcal{O}(1)$ worst-case time, whereas insertions and deletions of tree leaves are supported in $\mathcal{O}(1)$ amortized time. For $k = \omega(\text{polylog}(n))$ (and $\mathcal{O}(n)$), we show that the same set of operations can be supported in $\mathcal{O}(\lg k / \lg \lg k)$ time (amortized in the case of insertions/deletions). Our data structures are also able to associate b-bit satellite data to the tree nodes, providing several space/time trade-offs for space usage and accessing/modifying the data.

We also showed that dynamic binary trees on n nodes can be represented using $2n + o(n)$ bits of space, so that the tree operations are supported in $\mathcal{O}(1)$ time (amortized in the case of insert/delete operations). We support adding satellite

data to the tree nodes using $bn + o(n)$ extra bits (versus $bn + o(bn)$ extra bits of the fastest previous dynamic representation from the literature [13]), while providing several trade-offs for accessing/modifying the data.

After our work, we identify the following lines for future research:

- Currently, our data structures for dynamic k-ary cardinal trees use $\Theta(n) + o(n \lg k)$ bits on top of the lower bound $\mathcal{C}(n, k)$. An interesting open question is whether we can reduce the extra space to just $o(n)$ bits or not, as in the static case [32, 14].
- For dynamic binary trees, would it be possible to support all tree operations in $\mathcal{O}(1)$ time (amortized for insertions and deletions) just as in [13], yet using only $bn + o(n)$ extra bits of space for b-bit satellite data, as achieved by our data structure? Currently, we support all binary tree operations in constant time, except for access-data and change-data.
- Would it be possible for our data structures to support insertions and deletions in worst-case time, rather than amortized?
- Would it be possible for our data structures to support a richer set of operations, as for instance level-ancestor [5] and lowest-common-ancestor [6] queries?

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *Proc. of 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5029 of *Lecture Notes in Computer Science*, pages 277–289. Springer, 2008.
3. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. of 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM, 2010.
4. D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Information and Computation*, 209(7):1070–1102, 2011.
5. M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
6. M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
7. D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
8. A. Brodnik, S. Carlsson, E. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proc. of 6th International Workshop on Algorithms and Data Structures (WADS)*, volume 1663 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 1999.
9. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd. ed.)*. MIT Press, 2009.
11. J. Darragh, J. Cleary, and I. Witten. Bonsai: a compact representation of trees. *Software - Practice and Experience*, 23(3):277–291, 1993.
12. P. Davoodi and S. S. Rao. Succinct dynamic cardinal trees with constant time operations for small alphabet. In *Proc. of 8th Annual Conference on Theory and Applications of Models of Computation (TAMC)*, volume 6648 of *Lecture Notes in Computer Science*, pages 195–205. Springer, 2011.
13. A. Farzan and J. I. Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, 412:2668–2678, 2011.
14. A. Farzan, R. Raman, and S. S. Rao. Universal succinct representations of trees? In *Proc. of 36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2009.

15. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009.
16. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
17. M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, 1982.
18. R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.
19. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics - a foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.
20. W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoretical Computer Science*, 412(39):5176–5186, 2011.
21. G. Jacobson. Space-efficient static trees and graphs. In *Proc. of 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989.
22. J. Jansson, K. Sadakane, and W.-K. Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 2013. To appear.
23. J. Jansson, K. Sadakane, and W.-K.Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012.
24. S. Joannou and R. Raman. Dynamizing succinct tree representations. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 224–235. Springer, 2012.
25. H.-I Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4(3):28:1–28:13, 2008.
26. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
27. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
28. J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. of 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536. ACM/SIAM, 2001.
29. G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.
30. G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.
31. R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proc. of 7th International Workshop on Algorithms and Data Structures (WADS)*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer, 2001.
32. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
33. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. of 30th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *Leture Notes in Computer Science*, pages 357–368. Springer, 2003.
34. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. Manuscript, 2008.