

Succinct Dynamic Cardinal Trees with Constant Time Operations for Small Alphabet

Pooya Davoodi¹ and S. Srinivasa Rao²

¹ MADALGO*, Department of Computer Science, Aarhus University, IT Parken, Åbogade 34, DK-8200 Aarhus N, Denmark. E-mail: pdavoodi@cs.au.dk

² School of Computer Science and Engineering, Seoul National University, 599 Gwanakro, Gwanak-Gu, Seoul 151-744, Republic of Korea. E-mail: ssrao@cse.snu.ac.kr

Abstract. A k -ary cardinal tree is a rooted tree in which each node has at most k children, and each edge is labeled with a symbol from the alphabet $\{1, \dots, k\}$. We present a succinct representation for k -ary cardinal trees of n nodes where $k = O(\text{polylog}(n))$. Our data structure requires $2n + n \log k + o(n \log k)$ bits and performs the following operations in $O(1)$ time: `parent`, `child(i)`, `label-child(α)`, `degree`, `subtree-size`, `preorder`, `is-ancestor(x)`, `insert-leaf(α)`, `delete-leaf(α)`. The update times are amortized. The space is close to the information theoretic lower bound. The operations are performed in the course of traversing the tree. This improves the succinct dynamic k -ary cardinal trees representation of Arroyuelo [1] for small alphabet, by speeding up both the query time of $O(\log \log n)$, and the update time of $O((\log \log n)^2 / \log \log \log n)$ to $O(1)$, solving an open problem in [1].

1 Introduction

In this paper, we present a succinct representation for dynamic k -ary cardinal trees, i.e., rooted trees in which each node has at most k children and each edge is labeled by a symbol from the alphabet $\{1, \dots, k\}$, for a fixed k . They are also known as tries with degree k . We consider the case where for a k -ary cardinal tree of n nodes, the size of the alphabet is small, in particular $k = (\log n)^{O(1)}$.

A succinct data structure is a representation of an input which uses an amount of space close to the information theoretic lower bound, and supports the required operations efficiently. The information theoretic lower bound for representing a k -ary cardinal tree of n nodes is computed by taking the logarithm of the number of distinct such trees, i.e., $\log \mathcal{C}(n, k) = \log \left(\binom{kn+1}{n} / (kn+1) \right) \approx 2n + n \log k - o(n + \log k)$ bits [1, 9]. The required operations are the following:

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

parent: the parent of the current node.
child(i): the i -th child of the current node.
label-child(α): the child of the current node such that the in-between edge is labeled by α .
degree: the number of children of the current node.
subtree-size: the total number of nodes in the subtree rooted at the current node.
is-ancestor(x): true if the current node is an ancestor of a node x ; otherwise false.
preorder: the rank of the current node in the preorder traversal of the tree.
insert-leaf(α): insert a new leaf as a child of the current node with an edge labeled by α .
delete-leaf(α): delete the child of the current node with the in-between edge labeled by α (assuming that the child is a leaf).

Our data structure supports the operations in the course of traversing the tree, i.e., a traversal of the tree starts from the root, moves through the tree by performing the navigational operations on the current node, and ends at the root. All the operations have to be performed on the current node of the traversal. This is the same model that was used in [12, 15, 1]. There are applications where this assumption holds, e.g., constructing Lempel-Ziv indexes which is used for dynamic compressed full-text indexes; and constructing suffix trees, if we supplement the data structure with satellite data. If we do not restrict the operations to be only performed on the current node (if we allow them to be performed on any arbitrary node), Farzan and Munro [6] showed an amortized lower bound of $\Omega(\log n / \log \log n)$ for **child**, **subtree-size**, **insert-leaf**, and **delete-leaf**.

We use the unit-cost RAM model with word size $w = \Theta(\log n)$ bits.

Previous work. For static k -ary cardinal trees of n nodes, Benoit et al. [2] gave a representation that requires $2n + n \log k + o(n) + O(\log \log k)$ bits, and supports the navigational operations and queries in $O(1)$ time. The space bound of their structure is $\log \mathcal{C}(n, k) + \Omega(n)$ bits, as k grows. Raman et al. [14] improved the space to $\log \mathcal{C}(n, k) + o(n) + O(\log \log k)$ bits, while supporting all the operations except **subtree-size** in $O(1)$ time. Recently, a representation with the same space that also supports **subtree-size** in $O(1)$ time was given in [7].

For the dynamic k -ary cardinal trees, when $k = 2$, i.e., for dynamic binary trees, Munro et al. [12] gave the first representation that uses $2n + o(n)$ bits. This representation, in the course of traversing the tree, supports navigational operations and queries in $O(1)$ time and updates in $O(\log^2 n)$ amortized time. Their structure can also support accessing a b -bit satellite data associated with a node in $O(1)$ time, where $b = \Theta(\log n)$. If $b = O(1)$, they achieve $O(\log n)$ amortized update time, and if no satellite data is associated with the nodes, they obtain $O(\log \log n)$ amortized update time. For $b = O(\log n)$, Raman and Rao [15] improved the update time to amortized $O((\log \log n)^{1+\epsilon})$ while supporting the navigations and queries in $O(1)$ time, in the course of traversing the tree. They also showed how to store the satellite data in $bn + o(n)$ bits. Indeed, the total space of their structure is $2n + bn + o(n)$ bits. More recently, Farzan and Munro [6] proposed the *finger-update* model which is stronger than the

traversal pattern that is used in [12, 15, 1] and this paper as well. In the finger-update model, only the update operations are restricted to be performed on the current node of the traversal (indeed, finger-update is the current node), and all the other operations are allowed to be performed on any node at any time. For $b = O(\log n)$ and for ordinal trees which are the generalized binary trees where there is an order between the children of the nodes, Farzan and Munro [6] presented a data structure that supports all the queries in constant time and updates in constant amortized time. But their structure uses $2n + bn + o(bn)$ bits which is worse than that of [15].

The succinct representation of dynamic k -ary cardinal trees was posed as an open problem in [12]. In 1993, Darragh et al. [5] presented a compact representation of cardinal trees that uses $6n + n\lceil \log k \rceil$ bits of space and achieves $O(1)$ expected time for the operations. Recently, Arroyuelo [1] presented a data structure for this problem that uses $2n + n \log k + o(n \log k)$ bits of space, and supports navigational operations and queries in $O(\log k + \log \log n)$ time and updates in $O((\log k + \log \log n)(1 + (\log k)/(\log(\log k + \log \log n))))$ amortized time. When $k = (\log n)^{O(1)}$, his data structure achieves $O(\log \log n)$ query time and $O((\log \log n)^2 / \log \log \log n)$ amortized update time. Improving this was posed as an open problem by Arroyuelo [1]. We address this problem by presenting a data structure that uses $2n + n \log k + o(n \log k)$ bits and performs the navigation and query operations in $O(1)$ time, and the update operations in $O(1)$ amortized time. Associating satellite data with the nodes is supported by neither our data structure nor the one in [1]. The following theorem states our result.

Theorem 1. *For a k -ary cardinal tree with n nodes, there exists a dynamic data structure of size $2n + n \log k + o(n \log k)$ bits, where $k = (\log n)^{O(1)}$. This data structure supports the operations `parent`, `child`, `label-child`, `degree`, `subtree-size`, `preorder`, and `is-ancestor` in $O(1)$ time, and supports `insert-leaf` and `delete-leaf` in $O(1)$ amortized time.*

2 Preliminaries

Dynamic arrays. A dynamic array [15] is a structure that supports accessing, inserting, and deleting elements in arrays efficiently with a small memory overhead.

Lemma 1. (Dynamic arrays [15, 16]) *There exists a data structure to represent an array of $\ell = w^{O(1)}$ elements, each of size $r = O(w)$ bits, using $\ell r + O(k \log \ell)$ bits, for any parameter $k \leq \ell$. This data structure supports accessing the element of the array in a given index in $O(1)$ time, and inserting/deleting an element in/from a given index in $O(1 + \ell r/kw)$ amortized time. The data structure requires a precomputed table of size $O(2^{\epsilon w})$ bits for any fixed $\epsilon > 0$.*

Searchable partial sums. In the searchable partial sums problem for an array A of m numbers from the range $[0, \dots, k - 1]$, we have to maintain A under the following operations:

$\text{sum}(i)$: return the value $\sum_{j=1}^i A[j]$,
 $\text{update}(i, \delta)$: set $A[i] = A[i] + \delta$, assuming that $A[i] + \delta < k$, and δ is less than a certain fixed number,
 $\text{search}(i)$: return the smallest j such that $\text{sum}(j) \geq i$.

This problem has been considered for different ranges of m and k [13, 10]. But we are only interested in solving this problem for small m and k . Raman and Rao [13] gave a data structure that solves the problem for $m = w^\epsilon$ and $k \leq w$, for any fixed $0 \leq \epsilon < 1$. Their data structure achieves $O(1)$ time for all the operations and uses $O(mw)$ bits of space. In the following, we show that when both m and k are $O(w^c)$ for a constant $c > 0$, we can obtain a data structure with $O(1)$ time for all the operations that uses $m \log k + o(m \log k)$ bits of space.

Lemma 2. *For any integer $n < 2^w$, there exists a searchable partial sums structure to represent an array of m elements from the range $[0, \dots, k - 1]$, using $m \log k + o(m \log k)$ bits and a precomputed table of size $o(n)$ bits, where m and k are $(\log n)^{O(1)}$. This data structure supports the operations sum , update , and search in $O(1)$ time.*

Proof. We pack every $w/\log k$ elements of the array into a word. Within each word, every b numbers denote a chunk, where $b = \log^{1/4} n$. Within each chunk, the operations can be supported in $O(1)$ time using a precomputed table of size $o(n)$ bits. The space usage to store all the chunks is $m \log k + o(m \log k)$ 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 the left of the B-tree to the right gives the original array. The number of leaves is m/b and the depth of the B-tree is $O(1)$. 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 $O(1)$ time, using a precomputed table of size $o(n)$ bits. Since the number of internal nodes is $O(m/b^2)$, the space usage for the B-tree is $O((m/b^2) \cdot (b(\log k + \log m))) = o(m \log k)$ bits.

The operations on the input array, can be performed by traversing the tree top-down and computing the operations at the internal nodes in $O(1)$ time. \square

3 Data structure and static operations

We present a succinct representation for k -ary cardinal trees which uses $2n + n \log k + o(n \log k)$ bits, supports the navigational operations and queries in $O(1)$ time, and the updates in $O(1)$ amortized time. Our structure is similar to the structure of [1]. 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. Each micro tree representation of [1] supports the operations in logarithmic time. We improve the time to $O(1)$ for small alphabet.

Decomposition. We use the greedy decomposition algorithm of [12] to decompose the input tree to micro trees of size in the range $[\log^2 n \dots k^2 \log^2 n]$. The micro tree containing the root might be smaller than $\log^2 n$. This algorithm performs a postorder traversal of the tree. During the traversal, every at least $\log^2 n$ visited nodes make a micro tree (see [12] for more details). We change the algorithm of [12] a little bit to maintain the following. Let τ be a micro tree. The number of nodes (size) of τ is denoted by $|\tau|$. A frontier node of τ is a node, except the root of τ , that is adjacent to nodes in other micro trees. If the root of τ is adjacent to a frontier node of another micro tree τ' , then τ' is the parent micro tree of τ , and τ is a child micro tree of τ' . We duplicate the frontier nodes of τ such that every frontier is also the root of a child micro tree of τ . Therefore, all the children of a frontier node are in the same micro tree, each frontier node is a leaf and is adjacent to only one child micro tree, i.e., the number of frontier nodes of τ denoted by $n_f(\tau)$ equals the number of child micro trees of τ .

Micro tree representation. Each micro tree τ is represented with the tuple $(D_\tau, L_\tau, F_\tau, P_\tau, r_\tau, S_\tau)$ defined as follows.

- D_τ : the tree topology of τ , using the DFUDS representation of τ
- L_τ : the edge labels of τ in the DFUDS order
- F_τ : frontiers of τ
- P_τ : pointers to the child micro trees of τ
- r_τ : a pointer to the parent micro tree of τ
- S_τ : the subtree size of all the child micro trees of τ .

Let τ be the micro tree that contains the current node of the traversal. We perform the navigations within τ using D_τ , and for label-child using L_τ . In the case of traversing to a child micro tree of τ , we find the pointer to the child micro tree using F_τ and P_τ . For traversing to the parent micro tree, we use r_τ . To compute subtree-size within τ , we use D_τ . To compute subtree-size of the current node in the whole tree, we use S_τ to compute subtree-size of the root of each child micro tree of τ that is a descendant of the current node. Then we add the subtree size of the current node within τ with all the computed subtrees sizes. The operation is-ancestor can be easily performed using subtree-size.

For each of the six parts except r_τ , we make data structures to perform the corresponding operations on them efficiently. The space usage for D_τ is $2|\tau| + o(|\tau|)$ bits, for L_τ is $|\tau| \log k + o(|\tau| \log k)$ bits, for r_τ is $\log n_f(\tau)$ bits, and for F_τ , P_τ , and S_τ is $o(n_f(\tau))$ bits. Since the micro trees are roughly disjoint, the total space usage is $2n + n \log k + o(n \log k)$ bits. In the following, we describe all the six parts of the micro tree representations.

3.1 Tree topology of micro trees

We make a data structure that maintains a micro tree τ of size at most $k^2 \log^2 n = O(\text{polylog}(n))$ nodes using $2|\tau| + o(|\tau|)$ bits, which supports all the required operations within τ (including updates) except label-child in $O(1)$ time.

We represent the structure of τ by its DFUDS sequence which is a string of $2 \cdot |\tau|$ parentheses [2]. Benoit et al. [2] showed that the navigation and query operations on a static ordinal tree of size n can be supported in $O(1)$ time using $2n + o(n)$ bits of space by performing rank/select and the balanced parenthesis operations: `findclose`, `findopen`, and `enclose` on the DFUDS sequence of the tree. Let D_τ be the DFUDS sequence of τ . Our data structure supports rank/select and the balanced parenthesis operations as well as update operations all in $O(1)$ time on D_τ . Essentially, our data structure is a dynamic DFUDS sequence of length $O(\log^2 n)$. Note that inserting and deleting of leaves in τ correspond to inserting and deleting of the pair of parentheses `"()`" in D_τ .

Lemma 3. *There exists a dynamic data structure of size $2m + o(m)$ bits to maintain a sequence of m pairs of balanced parenthesis using precomputed tables of size $o(n)$, where $m = (\log n)^{O(1)}$. This data structure supports the operations: `findclose`, `findopen`, and `enclose` in $O(1)$ time, and supports inserting and deleting of the pair of parentheses `"()`" in $O(1)$ amortized time.*

Proof. This representation is similar to [4]. We divide the sequence into chunks of size $w\ell$ bits, where $\ell = O(\sqrt{\log n})$. Each chunk is represented by a dynamic array of size $w\ell + O(\sqrt{\log n} \log \ell)$ bits (see Lemma 1), which allows us to access, insert, or delete a parenthesis at a given index in $O(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits. Therefore, the total space used for the chunks is $2m + o(m)$ bits.

Now, we make a B-tree with branching factor b , where $b = O(\log^{1/4} n)$. Each leaf of the tree stores a pointer to a sub-chunk of size ℓ such that scanning the sub-chunks of the leaves from the left of the tree to the right gives the original sequence. The number of leaves is $2m/\ell$, and the depth of the tree is $O(1)$. At each internal node u , we maintain an array of length b such that its i -th element stores the number of open parenthesis in the chunks that are descendants of the i -th child of u . Since the array is small (i.e., $O(\log^{1/4} n \cdot \log \log n)$ bits), we can represent it by a searchable partial sums structure using a precomputed table of size $o(n)$ bits. This array is used to perform the operations `rank` and `select` in $O(1)$ time by traversing the tree from its root to the appropriate leaf. In addition to this array, similar to [4], we store seven arrays containing different information about the parentheses stored in the subtrees of u . These arrays are used to perform the parenthesis operations. Update operations are also straightforward. See [4] for more details. Since the number of internal nodes is $O(2m/(b\ell))$, the space usage for the B-tree is $O(2m/(b\ell) \cdot b \log m) = o(m)$ bits. \square

The following lemma presents our dynamic DFUDS structure based on the dynamic parenthesis maintenance structure of Lemma 3.

Lemma 4. *There exists a dynamic DFUDS representation of size $2|\tau| + o(|\tau|)$ bits for an ordinal tree τ of $(\log n)^{O(1)}$ nodes using precomputed tables of size $o(n)$ bits. This data structure supports the operations `parent`, `child`, `degree`, `subtree-size`, `is-ancestor`, and `preorder` all in $O(1)$ time, and supports the update operations `insert-leaf` and `delete-leaf` in $O(1)$ amortized time.*

Proof. Recall that the DFUDS sequence D_τ contains $2 \cdot |\tau|$ balanced parenthesis. It has been shown that all the operations `parent`, `child`, `degree`, `subtree-size`, and `is-ancestor` on τ can be supported using the balanced parenthesis operations and `rank/select` on D_τ [2]. Also the operation `preorder` can be supported using the balanced parenthesis operations and `rank/select` on D_τ [11]. Inserting and deleting leaves correspond to inserting and deleting the pair of parentheses “()”. \square

3.2 Edge labels of micro trees

Let L_τ be the sequence containing all the edge labels of τ in the DFUDS ordered. To perform `label-child`(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`(i). To find i , we find the number of α before the current node, and then find the position of the next α using `rank/select` structure on both D_τ and L_τ . To perform `insert-leaf`(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. For that, we construct a dynamic predecessor structure for all the edge labels below each internal node.

Note that L_τ consists of contiguous sub-sequences s_i , for $i = 1 \cdots I_\tau$, 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 that $|s_i| \leq k$. We construct the following: (1) a data structure that supports the operations `rank`, `select`, `insert`, and `delete` on L_τ , (2) a data structure for each s_i , if $|s_i| > \log n / \log \log n$, which supports the operations `predecessor`, `insert`, and `delete` on s_i . In the following, we explain these two structures, and then we combine them.

Dynamic rank/select structure. In the following lemma, we present a data structure which is used to perform `label-child` in a micro tree.

Lemma 5. *There exists a dynamic representation of size $m \log k + o(m \log k)$ bits for a sequence of m symbols from an alphabet of size k using precomputed tables of size $o(n)$ bits, where m and k are $(\log n)^{O(1)}$. This data structure supports the operations `rank` and `select` in $O(1)$ time, and supports the update operations `insert` and `delete` in $O(1)$ amortized time.*

Proof. There exists a static data structure that supports the operations `rank` and `select` in $O(1)$ time for an alphabet of size k , using a multi-ary wavelet tree with $O(1)$ height (Theorem 3.2 of [8]). We dynamize their structure in the following way. We set the branching factor of their wavelet tree to be k' , where $k' = O(\sqrt{\log n})$. At each internal node we use a dynamic `rank/select` 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 $\ell \log k'$ bits, for $\ell = (w / \log k') \log^{1/4} n$. Each chunk is represented by a dynamic array of size $\ell \log k' + O(\log^{1/4} n \log \ell)$ bits, which allows us to access, insert, or delete a

symbol at a given index in $O(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits (see Lemma 1). Therefore, the total space used for the chunks is $m \log k' + o(m \log k')$ bits.

Now, we make a B-tree with branching factor at most $\log^{\frac{1}{4}} n$. 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 of the B-tree to the right gives the original sequence. Therefore, each chunk corresponds to $\log^{1/4} n$ leaves. The number of leaves is $m/(\ell \log^{1/4} n)$ and the depth of the B-tree is $O(1)$. At each internal node u , we maintain $k + 1$ arrays, each of length $\log^{1/4} n$. One of the arrays is denoted by `Size`. The i -th element of the array `Size` 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 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 sums structure with $O(1)$ time for the partial sums operations, using a precomputed table of size $o(n)$ bits, since the arrays are small (i.e., $O(\log^{\frac{1}{4}} n \cdot \log \log n)$ bits).

To perform the operation $\text{rank}_\alpha(i)$, 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 α in the sub-chunks that are to the left of h , and are descendants of u . This counting can be performed in $O(1)$ time, using the partial sums structures that are constructed for the array `Size` 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 $O(1)$ time and perform the rank using word-level computation. The operation $\text{select}_\alpha(i)$ can be performed similarly in $O(1)$ time (the array `Size` is not required for `select`).

For the operations `insert` and `delete`, we perform them on the appropriate chunks in $O(1)$ 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 $O(1)$ amortized. \square

Dynamic predecessor. In the following lemma, we present a structure used for $\text{insert}(\alpha)$ to find the rank of α among its siblings.

Lemma 6. *There exists a dynamic predecessor data structure of size $o(m)$ bits for a sorted array of m elements, where $m = (\log n)^{O(1)}$ and each element is from the range $[0 \dots k - 1]$, using a precomputed table of size $o(n)$ bits. This data structure supports the operation `predecessor` in $O(1)$ time, and supports the update operations `insert` and `delete` in $O(1)$ amortized time.*

Proof. For this structure, we use the same packing strategy and dynamic arrays as we used in the proof of Lemma 5. We make a B-tree with branching factor b , where $b = \sqrt{\log n}$. 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 $O(1)$. At each internal nodes, we maintain b guiding indexes. Every node (including leaves) has $b \log k = o(w)$ 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 $O(1)$ time. For the update operations, we also update the internal nodes in a bottom-up traversal. The rebalancing is applied as needed. \square

The following lemma combines Lemma 5 and 6, and shows how to perform the operation `label-child` on τ using the data structures of D_τ and L_τ .

Lemma 7. *For a k -ary cardinal tree τ of at most $k^2 \log^2 n$ nodes where $k = (\log n)^{O(1)}$, there exists a dynamic representation of size $2|\tau| + |\tau| \log k + o(|\tau| \log k)$ bits that supports the operation `label-child` in $O(1)$ time, and supports the update operations `insert-leaf` and `delete-leaf` in $O(1)$ amortized time. The structure uses precomputed tables of size $o(n)$ bits.*

Proof. Similar to [1], we represent the tree τ with D_τ and L_τ . We make a data structure for each of D_τ and L_τ using Lemma 4, 5, and 6 in totally $2|\tau| + |\tau| \log k + o(|\tau| \log k)$ bits. \square

3.3 Frontiers of micro trees

During performing the operations on a micro tree τ , we need to check whether the current node is a frontier of τ or not, and if it is a frontier, then we may need to traverse to the micro tree rooted at that frontier using a pointer. For the checking, we represent the frontiers of τ with an array F_τ of $n_f(\tau)$ elements. The representation of pointers is explained in Section 3.4. The i -th element of the array F_τ contains the difference between two preorder numbers which belong to the i -th and $i + 1$ -st frontiers of τ in the preorder traversal of τ . We make a searchable partial sums structure for F_τ . Since F_τ has $(\log n)^{O(1)}$ elements, each of size $O(\log F_\tau)$ bits, we use the searchable partial sums structure of Lemma 2 that supports the operations `sum`, `update`, and `search` in $O(1)$ time, using $n_f(\tau) \log |\tau| + o(n_f(\tau) \log |\tau|)$ bits. Thus the overall space for all the micro trees is $o(n)$ bits. To check whether the current node is a frontier or not, we use `search` on F_τ for the preorder number of the current node.

3.4 Pointers to other micro trees

There are two cases where we need to traverse from τ , containing the current node x , to another micro tree: 1) if x is a frontier of τ , then we need to follow a pointer to the child micro tree rooted at x , 2) if x is the root of τ , then we need to follow a pointer to move to the parent micro tree of τ .

For the first case, for each frontier of τ , we store a pointer to another micro tree that is rooted at that frontier. These pointers are represented in the following way. Let τ_i be the micro tree rooted at $F_\tau[i]$, the i -th frontier of τ . We make an array P_τ of $n_f(\tau)$ elements such that $P_\tau[i]$ maintains a pointer to τ_i . Therefore, whenever the current node is $F_\tau[i]$ (that we can check using the representation of Section 3.3), we can traverse to τ_i . The space usage to store P_τ for all the micro trees is $o(n)$ bits. For the second case, since x is a frontier of the parent micro tree τ' , we store r_τ such that $F_{\tau'}[r_\tau]$ maintains the preorder number of x .

3.5 Subtree sizes

We make a data structure that allows us to compute the subtree size of the current node in $O(1)$ time. Let τ be the micro tree containing the current node. Lemma 4 shows that we can perform `subtree-size` on the current node within τ in $O(1)$ time. But, to this number, we should add the subtree size of the root of each child micro tree of τ that is a descendant of the current node. For this, we make an array S_τ of length $n_f(\tau)$ such that $S_\tau[i]$ maintains the subtree size of the root of τ_i , where τ_i is the child micro tree rooted at the i -th frontier of τ in the preorder traversal of τ . We represent S_τ by a searchable partial sums structure using $n_f(\tau) \log |\tau| + o(n_f(\tau) \log |\tau|)$ bits (see Lemma 2). The overall space for all the micro trees is $o(n)$ bits. To compute the subtree size, we need to find $\sum_{i=j_\ell}^{j_r} S_\tau[i]$, 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 F_τ 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 F_τ for the preorder number of e .

4 Update operations

Operation insert-leaf. To perform `insert-leaf`(α) in a micro tree τ , we update the representation of τ in the following way. We update D_τ by inserting “()” as a leaf into a position i that we find by a predecessor search in s_j of L_τ corresponding to the current node. We update L_τ by inserting α as a new label into position i . The new leaf is not a frontier, but if it is inserted between two frontiers, then it changes the difference between the preorder numbers of them. Therefore, we increment the appropriate element of F_τ . All the above operations are performed in $O(1)$ time.

If $|\tau|$ exceeds the value of $k^2 \log^2 n$, we split τ into micro trees of size in $[2 \log^2 n \cdots 2k \log^2 n]$ using the decomposition algorithm that we used in Section 3. 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 are both can be performed in $O(|\tau|) = O(k^2 \log^2 n)$ time. Since, this procedure makes micro trees of small enough size (at most $k \log^2 n$), therefore, $k^2 \log^2 n$ number of `insert-leaf` is required to make any of them full and the insertion time is $O(1)$ amortized.

Operation delete-leaf. To perform `delete-leaf`(α) in a micro tree τ , we update the representation of τ similarly as `insert-leaf`(α). If $|\tau|$ becomes smaller than $\log^2 n$, then we combine τ with its parent micro tree. This can be performed by inserting the nodes of τ into the parent micro tree, in the preorder traversal of τ using `insert-leaf`. This procedure takes $O(|\tau|) = O(\log^2 n)$ time. Since the new micro trees that we construct in the split procedure of Section 4 are large enough (at least $2 \log^2 n$ size), the deletion time is $O(1)$ amortized.

Memory management. We store each micro tree in a separate location of the memory using an Extendible Array [3]. Since the number of micro trees is at most $n/\log^2 n$, and the nominal size of all the micro trees is $s = 2n + n \log k + o(n \log k)$ bits, then the space requirement for the whole collection of micro trees is $s + O(nw/\log^2 n + \sqrt{snw/\log^2 n}) = 2n + n \log k + o(n \log k)$ bits [16].

References

1. D. Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching*, pages 277–289. Springer-Verlag, 2008.
2. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
3. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proc. 6th Workshop on Algorithms and Data Structures*, pages 37–48. Springer-Verlag, 1999.
4. H.-L. Chan, W.-K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.
5. J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: a compact representation of trees. *Software - Practice and Experience*, 23(3):277–291, 1993.
6. A. Farzan and J. I. Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, In Press, Corrected Proof, 2010.
7. A. Farzan, R. Raman, and S. S. Rao. Universal succinct representations of trees? In *Proc. 36th International Colloquium on Automata, Languages and Programming*, pages 451–462. Springer, 2009.
8. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.
9. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Math*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1988.
10. W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. 14th International Symposium on Algorithms and Computation*, pages 505–516. Springer, 2003.
11. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 575–584. SIAM, 2007.
12. J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 529–536. SIAM, 2001.
13. R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proc. 7th Workshop on Algorithms And Data Structures*, pages 426–437. Springer-Verlag, 2001.
14. R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
15. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, pages 357–368. Springer-Verlag, 2003.
16. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. Manuscript, 2008.