Succinct representation of labeled trees

Dekel Tsur*

Abstract

We give a representation for labeled ordered trees that supports labeled queries such as finding the i-th ancestor of a node with a given label. Our representation is succinct, namely the redundancy is small-o of the optimal space for storing the tree. This improves the representation of He et al. [ISAAC 2012] which is succinct only when the entropy of the labels is $\omega(1)$.
Keywords: succinct data-structures; labeled trees.

1 Introduction

A problem which was extensively studied in recent years is representing an ordered rooted tree using space close to the information-theoretic lower bound while supporting numerous queries on the tree, e.g. [3–5,7,9–12]. Geary et al. [7] studied an extension of this problem, in which the nodes of the tree are labeled with characters from alphabet $\{1,\ldots,\sigma\}$. The queries on labeled trees receive a character α as an additional argument, and the goal of a query is to locate a certain node whose label is α or to count the nodes satisfying some property and whose labels are α. The set of queries considered by Geary et al. is given in Table 1. Geary et al. gave a representation that uses $n\log\sigma+2n+O(n\sigma\log\log\log n/\log\log n)$ bits, where n is the number of nodes and σ is the size of the alphabet, and answers queries in constant time. For $\sigma=o(\log\log n/\log\log\log n)$, the space is $n\log\sigma+2n+o(n)$, namely, the space is $o(n)$ more than the information-theoretic lower bound.

Barbay et al. [1] and Ferragina et al. [6] gave labeled tree representations that use space close to the lower bound for large alphabets, but the set of supported queries is more restricted. He et al. [8] improved the result of Geary et al. by showing a labeled tree representation based upon a rank-select structure on the string P_T that contains the labels of the nodes in preorder. Using the rank-select structure of Belazzougui and Navarro [2] the following results were obtained: (1) For $\sigma=w^{O(1)}$, there is a representation that uses $nH_0(P_T)+O(n)$ bits and answers queries in $O(1)$ time, where w is the word size and $H_0(P_T)$ is the zero-order entropy of P_T. (2) For $\sigma\leq n$, there is a representation that uses $nH_0(P_T)+o(nH_0(P_T))+O(n)$ bits. Label queries are answered in $O(1)$ time, and preorder_rank queries are answered in $O(f(n))$ time for any $f(n)=\omega(1)$, or vice versa. Other queries are answered in $O(\log\frac{\log\sigma}{\log w})$ time. The representation of He et al. supports all the queries of Table 1 and additional queries. Note that the representation is succinct only if $H_0(P_T)=\omega(1)$.

In this paper we give a fully succinct representation of labeled trees. Our result is given in the following theorem.

Theorem 1. *Let T be a labeled tree with n nodes and labels from $\{1,\ldots,\sigma\}$.*

*Department of Computer Science, Ben-Gurion University of the Negev. Email: dekelts@cs.bgu.ac.il

Table 1: Supported queries on a labeled tree. A node with label α is called an α-node. A α-child of a node is a child which is an α-node. Other α- terms are defined similarly.

Query	Description
label(x)	The label of x.
$\text{depth}_\alpha(x)$	The number of α-nodes on the path from the root to x.
$\text{level\_ancestor}_\alpha(x, i)$	The α-ancestor y of x for which $\text{depth}_\alpha(y) = \text{depth}_\alpha(x) - i$.
$\text{parent}_\alpha(x)$	$\text{level\_ancestor}_\alpha(x, 1)$.
$\deg_\alpha(x)$	The number of α-children of x.
$\text{child\_rank}_\alpha(x)$	The rank of x among its α-siblings.
$\text{child\_select}_\alpha(x, i)$	The i-th α-child of x.
$\text{num\_descendants}_\alpha(x)$	The number of α-descendants of x.
$\text{preorder\_rank}_\alpha(x)$	The preorder rank of x among the α-nodes.
$\text{preorder\_select}_\alpha(i)$	The i-th α-node in the preorder.
$\text{postorder\_rank}_\alpha(x)$	The postorder rank of x among the α-nodes.
$\text{postorder\_select}_\alpha(i)$	The i-th α-node in the postorder.

1. *For $\sigma = w^{O(1)}$, there is a representation of T that uses $nH_0(P_T) + 2n + o(n)$ bits and answers the queries of Table 1 in $O(f(n))$ time for any $f(n) = \omega(1)$.*

2. *For $\sigma \leq n$, there is a representation of T that uses $nH_0(P_T) + 2n + o(nH_0(P_T)) + o(n)$ bits. Label queries are answered in $O(1)$ time, and* preorder_rank *queries are answered in $O(f(n))$ time for any $f(n) = \omega(1)$, or vice versa. The rest of the queries of Table 1 are answered in $O(\log \frac{\log \sigma}{\log w})$ time.*

Note that our representation supports only the queries considered by Geary et al. and it does not support the additional queries considered by He et al. The additional queries are the queries $\text{height}_\alpha(x)$ (the maximum number of α-nodes on a path from x to a descendant leaf), $\text{leaf\_rank}_\alpha(x)$ (number of α-leaves with preorder number less than or equal to x), $\text{leaf\_select}_\alpha(i)$ (i-th α-leaf in preorder), and other queries.

2 Preliminaries

This section contains known and new results that we use in our labeled tree representation. Since the proofs of the new results are quite long, we chose to give these proofs after Section 3 that contains the main result of this paper.

2.1 Rank-select structures

A *rank-select structure* stores a string S over alphabet $\{1, \ldots, \sigma\}$ and supports the following queries: (1) $\text{rank}_\alpha(S, i)$ returns the number of occurrences of α in the first i characters of S (2) $\text{select}_\alpha(S, i)$ returns the i-th occurrence of α in S (3) $\text{access}(S, i)$ returns the i-th character of S. The problem of designing a succinct rank-select structure with efficient query times was studied extensively. For our purpose, we use the following results.

Theorem 2 (Belazzougui and Navarro [2])**.** *A rank-select structure can be built on a string S of length n over alphabet $\{1, \ldots, \sigma\}$ such that (1) If $\sigma = w^{O(1)}$, the space is $nH_0(S) + o(n)$ bits, and the structure answers rank queries in $O(1)$ time (2) If $\sigma \leq n$, the space is $nH_0(S) + o(nH_0(S)) + o(n)$ bits, and the structure answers rank queries in*

Table 2: Supported queries on a weighted tree. BP denotes the balanced parentheses representation of the tree. A node is represented by the index of its opening parenthesis in BP.

Query	Description
$w_a(x)$	The w_a-weight of x.
$\text{depth}_a(x)$	The w_a-weight of the nodes of the path from the root to x.
$\text{level\_ancestor}_a(x, i)$	The lowest ancestor y of x for which the w_a-weight of the nodes of the path from y to x, excluding x, is at least i.
$\text{parent}_a(x)$	$\text{level\_ancestor}_a(x, 1)$.
$\deg_a(x)$	The w_a-weight of the children of x.
$\text{child\_rank}_a(x)$	The w_a-weight of x and its left siblings.
$\text{child\_select}_a(x, i)$	The leftmost child y of x for which $\text{child\_rank}_a(y) \geq i$.
$\text{num\_descendants}_a(x)$	The w_a-weight of the proper descendants of x.
$\text{bp\_close}(x)$	The index of the '0' character in BP that corresponds to x.
$\text{bp\_rank}_{a,b}(i)$	The w_a-weight of the nodes that correspond to '1' characters of BP with index at most i, plus the w_b-weight of the nodes that correspond to '0' characters of BP with index at most i.
$\text{bp\_select}_{a,b}(i)$	The minimum index j for which $\text{bp\_rank}_{a,b}(j) \geq i$.

$O(\log \frac{\log \sigma}{\log w})$ *time. The structure answers access queries in* $O(1)$ *time and select queries in* $O(f(n))$ *time for any* $f(n) = \omega(1)$*, or vice versa.*

Theorem 3 (Pătraşcu. [13])**.** *For any constant integer* t*, a rank-select structure can be built on a binary string* S *of length* n *that contains* k *ones such that the space is* $k \log(n/k) + O(k + n/\log^t n)$ *bits, and the structure answers queries in* $O(1)$ *time.*

2.2 Representation of unlabeled trees

We use the following result on representation of unlabeled ordered trees. We are interested in a representation that supports the unlabeled versions of the queries listed in Table 1 and additional queries such as lca queries (finding the lowest common ancestor of two nodes).

Theorem 4 (Navarro and Sadakane [12])**.** *An ordered tree can be stored using* $2n + o(n)$ *bits such that the queries on the tree are answered in* $O(1)$ *time.*

2.3 Representation of weighted trees

In this section we consider the problem of representing ordered trees with weights on the nodes. We will use weighted trees in our representation of labeled trees.

Let T be a tree with weights $w_1(v), \ldots, w_s(v)$ for each node v, where each weight is from $\{0, \ldots, X-1\}$. For a set of nodes U, the w_a*-weight* of U is $\sum_{v \in U} w_a(v)$. Throughout this section we assume that the balanced parentheses string of a tree is a binary string, where opening and closing parenthesis are represented by 1 and 0, respectively. A node is represented by the index of its opening parenthesis in the balanced parentheses representation of the string. The weighted tree queries we need are described in Table 2.

Lemma 5. *A weighted tree with* n *nodes and* $s = O(1)$ *weight functions with weights from* $\{0, \ldots, X-1\}$*, where* $X = O(\log n)$*, can be stored using at most* $2n \log(2X^s) + o(n)$ *bits such that the queries in Table 2 are answered in* $O(1)$ *time.*

The proof of Lemma 5 is given in Section 4.

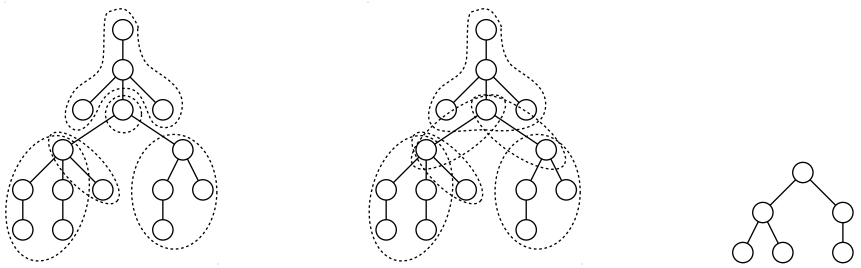

Figure 1: Example of tree decomposition using $L = 4$. Figure (a) shows the decomposition of Farzan and Munro, and Figure (b) shows our modified decomposition. Figure (c) shows the tree T_L.

2.4 Tree decomposition

In the next lemma we present a tree decomposition that we will use in our labeled tree representation. The decomposition is a slightly modified version of the decomposition of Farzan and Munro [5]. An example for the decomposition is given in Figure 1b.

Lemma 6. *For a tree T with n nodes and an integer L, there is a collection $\mathcal{D}_{T,L}$ of subtrees of T with the following properties.*

1. *Every edge of T appears in exactly one tree of $\mathcal{D}_{T,L}$.*
2. *The size of each tree in $\mathcal{D}_{T,L}$ is at most $2L + 1$.*
3. *The number of trees in $\mathcal{D}_{T,L}$ is $O(n/L)$.*
4. *For every tree $T' \in \mathcal{D}_{T,L}$ there are two intervals of integers I_1 and I_2 such that a node $x \in T$ is a non-root node of T' if and only if* preorder_rank$(x) \in I_1 \cup I_2$, *where* preorder_rank(x) *is the preorder rank of x in T. The node with preorder rank* $\min(\max(I_1), \max(I_2))$ *is called the* special node *of T'.*
5. *For every $T' \in \mathcal{D}_{T,L}$, only the root and the special node of T' can appear in other trees of $\mathcal{D}_{T,L}$.*

The proof of Lemma 6 is given in Section 5. For a tree T and an integer L we define a tree T_L as follows (see Figure 1c). Construct a tree decomposition $\mathcal{D}_{T,L}$ according to Lemma 6. If the root r of T appears in several trees of $\mathcal{D}_{T,L}$, add to $\mathcal{D}_{T,L}$ a tree that consists of r. The set of nodes of T_L is the set $\mathcal{D}_{T,L}$. For two trees $x', y' \in \mathcal{D}_{T,L}$, x' is the parent of y' in T_L if and only if the root of y' is equal to the special node of x' (or x' is the tree that consists of r).

3 Representation of labeled trees

As in He et al. [8], we build trees T^α for every $\alpha \in \{1, \ldots, \sigma\}$. To build T^α, we temporarily add to T a new root with label α. Then, let X_α be the set of all α-nodes in T and their parents. The nodes of T^α are the nodes of X_α, and x is a parent of y in T^α if and only if x is the lowest (proper) ancestor of y that appears in X_α. Unlike He et al., we do not store the tree T^α. Instead, we store a weighted tree that contains only part of the information of T^α. The weighted tree is the tree T^α_L obtained from the tree decomposition of Lemma 6.

For the case $\sigma = w^{O(1)}$ the value of L is $L = f(n)$, where f is a function that satisfies $f(n) = \omega(1)$ and $f(n) = O(\log n)$, and for larger σ we set $L = \sqrt{\log \frac{\log \sigma}{\log w}}$. Denote by n_α the number of α-nodes in T. We say that a character α is *frequent* if $n_\alpha \geq L$. We only construct the trees T_L^α for frequent characters.

For a node x' in T_L^α, let $V(x')$ be the set of all nodes in the tree x', excluding the root of x', and let $V_\alpha(x')$ be the α-nodes in $V(x')$. Let $C(x')$ be the leftmost α-child of the root of x' among the nodes of $V_\alpha(x')$. We define the following weight functions for T_L^α.

1. $w_1(x')$ is the number of nodes in $V_\alpha(x')$.
2. $w_2(x')$ is the number of nodes in $V_\alpha(x')$ whose preorder rank in T^α is less than or equal to the preorder rank of the special node of x'.
3. $w_3(x')$ is equal to $w_1(x') - w_2(x')$.
4. $w_4(x')$ is the number of nodes in $V_\alpha(x')$ which are ancestors of the special node of x'.
5. $w_5(x')$ is the number of α-children of the root of x'.
6. $w_6(x')$ is the rank of $C(x')$ among the nodes of $V_\alpha(x')$, when the nodes are sorted according to preorder (if $C(x')$ does not exist, $w_6(x') = 0$).
7. $w_7(x')$ is equal to 1 if the special node of x' is an α-node, and 0 otherwise.

Our representation of T consists of the following data-structures. We store a rank-select structure on P_T, using one of the structures of Theorem 2 according to the size of the alphabet. We also store an unlabeled tree $\hat{T}$ obtained from T by removing the labels. $\hat{T}$ is stored using Theorem 4. We also store the trees T_L^α. In order to reduce the space, we do not store these trees individually. Instead, we merge them into a single tree T'. The tree T' contains a new root node, on which the trees T_L^α are hanged, ordered by increasing values of α. The tree T' is stored using the representation of Lemma 5. In order to map a node of T_L^α to the corresponding node of T', and vice versa, we store the rank-select structure of Theorem 3 on the string $N = 10^{n_{\alpha_1}} 10^{n_{\alpha_2}} \cdots$, where $\alpha_1 < \alpha_2 < \cdots$ are the frequent characters. We also store the rank-select structure of Theorem 3 on a binary string F of length σ in which $F[\alpha] = 1$ if α is frequent. Since a mapping between nodes of T_L^α and nodes of T' can be computed in constant time, in the following we shall assume that the trees T_L^α are available.

We next analyze the space complexity of the representation. The space for P_T is $nH_0(P_T) + o(n)$ bits for small alphabet, and $nH_0(P_T) + o(nH_0(P_T)) + o(n)$ bits for large alphabet. The space for $\hat{T}$ is $2n + o(n)$ bits. The tree T' has $O(n/L)$ nodes, and the weight functions have ranges $\{0, \ldots, L\}$. Thus, the space for T' is $O((n/L) \log L) + o(n) = o(n)$ bits. The strings N and F have at most n zeros and at most n/L ones. Thus, the space for the rank-select structures on these strings is $O((n/L) \log L) + o(n) = o(n)$ bits. Therefore, the total space of the representation is $nH_0(P_T) + 2n + o(n)$ for small alphabet, and $nH_0(P_T) + 2n + o(nH_0(P_T)) + o(n)$ for large alphabet.

In the following, when we use an unlabeled tree operation (e.g., parent(x)) we assume that the operation is performed on $\hat{T}$, and when we use a weighted tree operation we assume that the operation is performed on T_L^α.

3.1 Mapping from T_L^α to T

Let x' be a node of T_L^α. From property 4 of Lemma 6 there are two intervals $[l_1(x'), r_1(x')]$ and $[l_2(x'), r_2(x')]$ such that an α-node x of T is a non-root node of x' if and only if the rank of x among all the α-nodes of T, sorted in preorder, is in one of the intervals. These intervals can be computed as follows:

$$\begin{aligned} r_1(x') &= \text{bp\_rank}_{2,3}(x') \\ l_1(x') &= r_1(x') - w_2(x') + 1 \\ r_2(x') &= \text{bp\_rank}_{2,3}(\text{bp\_close}(x')) \\ l_2(x') &= r_2(x') - w_3(x') + 1 \end{aligned}$$

The set $V_\alpha(x')$ can be computed with

$$V_\alpha(x') = \{\text{preorder\_select}(\text{select}_\alpha(P_T, s)) : s \in [l_1(x'), r_1(x')] \cup [l_2(x'), r_2(x')]\},$$

and $C(x')$ can be computed with $\text{preorder\_select}(\text{select}_\alpha(P_T, k))$, where $k = l_1(x') + w_6(x') - 1$ if $w_6(x') \leq w_2(x')$, and $k = l_2(x') + w_6(x') - w_2(x') - 1$ otherwise.

3.2 Mapping from T to T_L^α

Let x be a node of T and α be a frequent character. If $x \in X_\alpha$, we want to compute the node $x' \in T_L^\alpha$ such that $x \in V(x')$. We denote this node by $\text{map}(x)$. Additionally, we compute whether x is the special node of $\text{map}(x)$.

We consider two cases. The first case is when x is an α-node. Then,

$$\text{map}(x) = \text{bp\_select}_{2,3}(\text{preorder\_rank}_\alpha(x)).$$

Moreover, x is the special node of $\text{map}(x)$ if and only if $w_7(x') = 1$ and $\text{preorder\_rank}_\alpha(x) = r_1(x')$.

Now suppose that x is not an α-node. The algorithm for computing $\text{map}(x)$ is as follows.

1. Let u and v be the α-descendants of x with minimum and maximum preorder ranks, respectively (u, v are computed using the structures on P_T and $\hat{T}$). If u, v do not exist return NULL.

2. Compute $u' = \text{map}(u)$ and $v' = \text{map}(v)$.

3. If u' is an ancestor of v' (including $u' = v'$), compute $V_\alpha(u')$. Scan the nodes of $V_\alpha(u')$ and check for each node whether it is a child of x. If no child of x was found, return NULL. Compute $C(u')$. If $C(u')$ exists and $\text{parent}(C(u')) = x$ then $\text{map}(x) = \text{parent}(u')$ and x is the special node of $\text{map}(x)$. Otherwise, $\text{map}(x) = u'$ and x is not the special node.

4. If v' is a proper ancestor of u', handle this case analogously to the handling of the previous case.

5. If neither u' or v' is an ancestor of the other, compute $y' = \text{child\_select}_5(\text{lca}(u', v'), 1)$. If y' does not exists, return NULL. Compute $C(y')$. If $C(y')$ exists and $\text{parent}(C(y)) = x$, then $\text{map}(x) = \text{lca}(u', v')$ and x is the special node of $\text{map}(x)$. Otherwise, return NULL.

We now show the correctness of the algorithm above. It is easy to verify that the algorithm is correct when x is an α-node. Moreover, the algorithm returns a non NULL value only after finding an α-child of x. Thus, the algorithm is correct when $x \notin X_\alpha$. Now, suppose that $x \in X_\alpha$ and x is not an α-node. By definition, x has at least one α-child. Denote by u_2 and v_2 the ancestors of u and v that are children of x in T^α, respectively (note that u_2 can be equal to v_2), and let $u'_2 = \text{map}(u_2)$ and $v'_2 = \text{map}(v_2)$. By the definition of u and v, every α-child of x is between u_2 and v_2 in T^α.

We first claim that if $u'_2 = v'_2$ (namely, if u_2 and v_2 are in the same tree in the decomposition of T^α) then one of the nodes u' and v' is equal to u'_2 and the other node is a descendant of u'_2 (or equal to u'_2). This claim is true since u_2 and v_2 are siblings in T^α, and therefore at most one of them is an ancestor of the special node of u'_2. Thus, at most one of u_2 and v_2 has descendants that are not in u'_2, and the claim follows.

Suppose that u' is an ancestor of v'. We show that $u'_2 = v'_2$. Suppose conversely that $u'_2 \neq v'_2$. Since u_2 and v_2 are siblings in T^α, u'_2 and v'_2 are siblings in T^α_L. Now, u' is a descendant of u'_2 and v' is a descendant of v'_2. Thus, u' is not an ancestor of v', a contradiction.

Since $u'_2 = v'_2$, from the claim above it follows that $u' = u'_2$. We conclude that all the α-children of x are in $V_\alpha(u')$. Therefore, the scan of $V_\alpha(u')$ finds an α-child of x. By the definition of $\text{map}(\cdot)$, u is not the root of u', and therefore the node x is also a node of u'. Thus, $\text{map}(x) = \text{parent}(u')$ if x is the root of u', and $\text{map}(x) = u'$ otherwise. Since u the former case occurs if and only if $C(u')$ exists and $\text{parent}(C(u')) = x$.

Now consider the case when neither u' or v' is an ancestor of the other. From the claim above, $u'_2 \neq v'_2$. Moreover, u'_2 and v'_2 are children of $\text{map}(x)$. Since u' is a descendant of u'_2 and v' is a descendant of v'_2, it follows that $\text{map}(x) = \text{lca}(u', v')$ and x is the special node of $\text{map}(x)$. Moreover, we have in this case that y' and $C(y')$ exist, and $\text{parent}(C(y)) = x$.

3.3 Answering queries

In this section we describe how to implement the queries of Table 1. The queries label, preorder_rank, preorder_select, num_descendants and postorder_rank are answered as in [8]. Answering the remaining queries is also similar to [8], but here additional steps are required as a weighted tree T^α_L holds less information than T^α. The general idea is to use the tree T^α_L to get an approximate answer to the query. Then, by enumerating the nodes of constant number of $V_\alpha(\cdot)$ set, the exact answer is found. The time complexity of answering a query is thus $O(L \cdot t_{\text{select}})$, where t_{select} is the time for a select query on P_T. Since $t_{\text{select}} = O(1)$ for small alphabet and $t_{\text{select}} = o(\sqrt{\log \frac{\log \sigma}{\log w}})$ for large alphabet, it follows that the time for answering a query is any $\omega(1)$ for small alphabet and $O(\log \frac{\log \sigma}{\log w})$ for large alphabet.

We assume that α is a frequent character until the end of the section. Handling queries in which α is non-frequent is done by enumerating all the α-nodes in T.

3.3.1 $\text{parent}_\alpha(x)$ query

We consider two cases. If x is an α-node the query is answered as follows.

1. Compute $x' = \text{map}(x)$.
2. Compute $V_\alpha(x')$. Scan the nodes of $V_\alpha(x')$ in reverse order, and check for each node v whether v is an ancestor of x. If an ancestor of x is found, return it.
3. Compute $y' = \text{parent}_4(x')$. If y' does not exist return NULL.

4. Compute $V_\alpha(y')$. Scan the nodes of $V_\alpha(y')$ in reverse order, and check for each node v whether v is an ancestor of x. When an ancestor of x is found, return it.

If x is not an α-node then the query is answered as in He et al. [8].

3.3.2 $\text{depth}_\alpha(x)$ query

1. Let $y = x$ if x is an α-node, and $y = \text{parent}_\alpha(x)$ otherwise.
2. Compute $y' = \text{map}(y)$.
3. Compute $V_\alpha(y')$ and scan the nodes of $V_\alpha(y')$. Count the number of nodes that are ancestors of x, and let i denote this number.
4. Return $i + \text{depth}_4(y') - w_4(y')$.

3.3.3 $\text{level\_ancestor}_\alpha(x, i)$ query

1. Compute $y = \text{parent}_\alpha(x)$. If y does not exist return NULL.
2. If $i = 1$ return y.
3. Compute $y' = \text{map}(y)$.
4. Compute $V_\alpha(y')$. Scan the nodes of $V_\alpha(y')$ in reverse order. For each node v, if v is an ancestor of x, decrease i by 1. If i becomes 0 return v, and otherwise continue with the scan.
5. Compute $z' = \text{level\_ancestor}_4(y', i)$. If z' does not exist return NULL.
6. Set $i \leftarrow i - (\text{depth}_4(\text{parent}(y')) - \text{depth}_4(z'))$.
7. Compute $V_\alpha(z')$. Scan the nodes of $V_\alpha(z')$ in reverse order. For each node v, check whether v is an ancestor of x. If v is an ancestor, decrease i by 1. If i becomes 0 return v, and otherwise continue with the scan.

3.3.4 $\deg_\alpha(x)$ query

1. Compute $x' = \text{map}(x)$. If x' does not exist return 0.
2. If x is not the special node of x', compute $V_\alpha(x')$ and scan the nodes of $V_\alpha(x')$. Return the number of nodes that are children of x.
3. Return $\deg_5(x')$.

We now explain the correctness of the algorithm above. If x is not the special node of x' then all the children of x in T^α are in the tree x'. Thus, it suffices to scan $V_\alpha(x')$. If x is the special node, the set of α-children of x is precisely the set of α-children of the roots of all trees y' such that y' is a child of x' in T^α_L. Therefore, $\deg_\alpha(x) = \deg_5(x')$.

3.3.5 child_rank$_\alpha(x)$ query

If x is an α-node the query is answered as follows.

1. Compute $x' = \text{map}(x)$.
2. Compute $V_\alpha(x')$ and scan the nodes of $V_\alpha(x')$. Count the number of nodes that are left siblings of x, and let i denote this number.
3. Compute $u = \text{parent}(x)$ and $u' = \text{map}(u)$.
4. If u is not the special node of u' return i.
5. Return $i + \text{child\_rank}_5(x') - w_5(x')$.

The case when x is not an α-node is handled as follows.

1. Compute $u = \text{parent}(x)$ and $u' = \text{map}(u)$. If u' does not exist return 0.
2. If u is not the special node of u', compute $V_\alpha(u')$ and scan the nodes of $V_\alpha(u')$. Return the number of nodes that are left siblings of x.
3. Let v be the α-predecessor of x. If v does not exist or if preorder_rank$(v) \leq$ preorder_rank(u) return 0.
4. Compute $v' = \text{map}(v)$.
5. Let w' be the child of u' which is an ancestor of v'.
6. Compute $V_\alpha(w')$ and scan the nodes of $V_\alpha(w')$. Count the number of nodes that are left siblings of x, and let i denote this number.
7. Return $i + \text{child\_rank}_5(w') - w_5(w')$.

3.3.6 child_select$_\alpha(x, i)$ query

1. Compute $x' = \text{map}(x)$. If x' does not exist return NULL.
2. If x is not the special node of x, compute $V_\alpha(x')$ and scan the nodes of $V_\alpha(x')$. For each node v, check whether v is a child of x. Return the i-th child.
3. Compute $y' = \text{child\_select}_5(x', i)$.
4. Set $i \leftarrow i - (\text{child\_rank}_5(y') - w_5(y'))$.
5. Compute $V_\alpha(y')$ and scan the nodes of $V_\alpha(y')$. For each node v, check whether v is a child of x. Return the i-th child.

3.3.7 postorder_select$_\alpha(i)$ query

1. Return bp_select$_{0,1}(i)$, where $w_0(v)$ is assumed to be 0 for all v.

4 Proof of Lemma 5

The representation is a variation of the unweighted tree representation of Navarro and Sadakane [12]. We first review the latter representation. Let T be an unweighted tree, and let P be its balanced parentheses representation. Navarro and Sadakane showed that queries on the tree can be implemented by supporting a set of *base queries* that include (1) the queries deg, child_rank, and child_select, which we call *child queries* (2) the following queries on P and a function $f : \{0, 1\} \to \{-1, 0, 1\}$ from a fixed set of functions $\mathcal{F}$:

$$\begin{aligned}
\text{sum}(P, f, i, j) &= \sum_{k=i}^{j} f(P[k]) \\
\text{fwd_search}(P, f, i, d) &= \min\{j \geq i : \text{sum}(P, f, i, j) = d\} \\
\text{bwd_search}(P, f, i, d) &= \max\{j \leq i : \text{sum}(P, f, i, j) = d\} \\
\text{rmqi}(P, f, i, j) &= \text{argmin}\{\text{sum}(P, f, 1, k) : i \leq k \leq j\} \\
\text{RMQi}(P, f, i, j) &= \text{argmax}\{\text{sum}(P, f, 1, k) : i \leq k \leq j\}
\end{aligned}$$

For example, $\text{level\_ancestor}(v, i) = \text{bwd\_search}(P, \pi, v-1, i)$, where π is a function defined by $\pi(0) = -1$ and $\pi(1) = 1$.

The representation of Navarro and Sadakane is based on partitioning of P into blocks of size $N = D^c$ for some constant integer c, where $D = \Theta(w/(c \log w))$. Each block of P is stored using an aB-tree [13] which is able to support the base queries on the block in constant time. The space of an aB-tree is $O(1)$ bits more than the information-theoretic lower bound. Since P is a binary string, the space of one aB-tree is $N + O(1)$, and the space for all trees is $2n + o(n)$. In order to support base queries on the entire string P, Navarro and Sadakane added additional data-structures. The space of the additional data-structures is $O((n/N) \log^{O(1)} n)$. Thus, by choosing large enough c, the additional space is $o(n)$.

We now describe the aB-trees in more details. An aB-tree over a block Q is a full tree of height c in which each internal node has D children. The N leaves of the tree correspond to the N characters of Q. Each internal node x corresponds to a substring $Q[l_x..r_x]$ of Q, where l_x and r_x are the ranks of the leftmost and rightmost descendant leaves of x in the left-to-right order of the leaves. Each leaf of the aB-tree stores a *value* which is a function of the character corresponding to the leaf. Moreover, each internal node stores a value which is a function of the values stored at the children of the node. The values stored at the nodes of the aB-tree are tuples, and the elements of these tuples will be described below.

Fix a function $f \in \mathcal{F}$. Let E_f be an array defined by $E_f[i] = \text{sum}(Q, f, 1, i)$. In order to support sum, fwd_search and bwd_search on Q and f, the aB-tree stores at each internal node x the values $e_f[x] = E_f[r_x]$, $m_f[x] = \min E_f[l_x..r_x]$, and $M_f[x] = \max E_f[l_x..r_x]$. Answering a $\text{fwd\_search}(Q, f, i, d)$ query is equivalent to finding the smallest index $j \geq i$ such that $\text{sum}(Q, f, 1, j) = d'$, where $d' = \text{sum}(Q, f, 1, i-1) + d$. In order to find j (assuming that j exists), start at the root of the aB-tree and descend toward the j-th leaf. The navigation at a node x is done as follows. Let $x_1, \ldots, x_D$ be the children of x. If x is an ancestor of the i-th leaf of the tree, let i' be the index such that $x_{i'}$ is an ancestor of the i-th leaf. Otherwise, $i' = 0$. We have that the j-th leaf is a descendant of x_k, where $k \geq i'$ is the minimum index such that $m_f[x_k] \leq d' \leq M_f[x_k]$. This property follows from the fact that the image of f is $\{-1, 0, 1\}$. Finding k can be done in constant time using a lookup table. Since each value of m_f or M_f is from $\{-N, \ldots, N\}$, the size of the lookup table is $2^{O(D \log N)} = o(n)$.

In order to support child queries on Q, the aB-tree stores at each internal node x a value $n[x]$ which is the number of times $m_\pi[x]$ appears in $E_\pi[l_x..r_x]$. We now describe how to answer a $\deg(v)$ query. We have that $\deg(v)$ is equal to the number of times the value $\text{depth}(v)$ appears in $E_\pi[v..\text{bp\_close}(v)]$. Moreover, $\text{depth}(v)$ is the minimum value in $E_\pi[v..\text{bp\_close}(v)]$. Let x and x' be the v-th and $\text{bp\_close}(v)$-th leaves of the aB-tree, respectively, and let y be the lowest common ancestor of x and x'. Computing $\deg(v)$ is done by descending from y toward x and toward x', and updating a counter during these descents. During the descent toward x, the counter is updated as follows. Let z be the current node. Let $z_1, \ldots, z_D$ be the children of z, and let z_i be the child which is an ancestor of x. The values $n[z_j]$ are added to the counter for every $j > i$ for which $m_\pi[z_j] = \text{depth}(v)$. This can be performed in constant time using a lookup table of size $o(n)$. The descend toward x' is similar, except that indices j smaller than i are considered. The queries child_rank and child_select are also answered using the $n[x]$ values.

We next describe the additional data-structures needed to answer queries on P. Let $b(i)$ denote the number of the block that contains the i-th character of P.

For supporting fwd_search on P for a function f, trees T_f and T'_f are constructed with weights from $\{0, \ldots, N\}$ on their edges, and weighted ancestor data-structures are built over these trees. The tree T_f is defined as follows. Let m_i (resp., M_i) be the minimum (resp., maximum) value of $\text{sum}(P, f, 1, j)$ for an index j that belongs to the i-th block of P. The nodes of T_f are $\{0, 1, \ldots, n/N\}$. A node $i > 0$ is a child of node j, where j is the minimum index for which $j > i$ and $m_j < m_i$. The edge between these nodes has weight $m_i - m_j$. If no such index exists, i is a child of node 0. The tree T'_f is built analogously from the M_i values. A $\text{fwd\_search}(P, f, i, d)$ query is answered by first checking whether the answer lies in the block $b(i)$ (using the aB-tree that stores the block). If not, a weighted ancestor query on T_f or T'_f finds the block in which the answer lies, and the location inside the block is found using the aB-tree storing this block. In more details, computing $\text{fwd\_search}(P, f, i, d)$ is equivalent to finding the rightmost index $j \geq i$ such that $\text{sum}(P, f, 1, j) = d'$, where $d' = \text{sum}(P, f, 1, i-1) + d$. If the answer is not in block $b(i)$, then it is inside the first block $k > b(i)$ for which $m_k \leq d' \leq M_k$. Again, this follows from the fact that the image of f is $\{-1, 0, 1\}$. Suppose that $d' < m_{b(i)}$. Then, k is the minimum index greater than $b(i)$ such that $d' \geq m_k$. Finding k is done by computing the lowest ancestor of $b(i)$ in T_f whose weighted distance to $b(i)$ is at least $m_{b(i)} - d'$. If $d' > m_{b(i)}$ then k is found by computing the lowest ancestor of $b(i)$ in T'_f whose weighted distance to $b(i)$ is at least $d' - M_{b(i)}$.

Navarro and Sadakane showed that for a tree with n nodes and edge weights from $\{0, \ldots, W\}$, and for every constant integer t, there is a weighted ancestor data-structure that uses $O(n \log n \log(nW) + nW/\log^t(nW))$ bits. Since a tree T_f has $O(n/N)$ nodes and the weights of the edges are at most N, the weighted ancestor data-structure over T_f uses $O((n/N)\log(n/N)\log n + n/\log^t n)$ bits.

In order to support the child queries, additional structures are built over a subset of the nodes of the tree. A node v of T is *marked* if (1) $b(\text{bp\_close}(v)) - b(v) \geq 2$ (2) There is no child v' of v for which $b(v') = b(v)$ and $b(\text{bp\_close}(v')) = b(\text{bp\_close}(v))$. A block k is *contained* in a marked node v if $b(v) < k < b(\text{bp\_close}(v))$ and this inequality is not satisfied for a child v' of v. The representation of T includes the following structures: (1) The rank-select structure of Theorem 3 on a bitmap B of length $2n$ in which $B[i] = 1$ if $P[i] = 1$ and i is a marked node. (2) An array that stores the degrees of the marked nodes, sorted according to postorder. (3) The rank-select structure of Theorem 3 on a bitmap B' defined as follows. For each marked node v generate a bitmap $B_v = 0^{c_1}10^{c_2}1\cdots$, where c_i is the number of children of v whose opening parentheses are in the i-th block that is

contained in v. Then, B' is the concatenation the bitmaps of the marked nodes, ordered according to preorder. (4) For each marked node v, the indices i, j such $B'[i..j] = B_v$. (5) For each marked node v, a list of the blocks contained in v. (6) For each block k, the rank of the block among the blocks that are contained in the same marked node as k.

Given a $\deg(v)$ query, if v is a marked node, the query is answered using the stored value. Otherwise, the parentheses of each child of v are contained either in block $b(v)$ or in block $b(\text{bp\_close}(v))$, except perhaps one child whose opening parenthesis is in the former block and its closing parenthesis is in the latter block. Therefore, the query can be answered using the aB-trees holding these blocks. A $\text{child\_select}(v, i)$ query for a non-marked node can be again answered using the aB-trees of blocks $b(v)$ and $b(\text{bp\_close}(v))$. For a marked node v, if the opening parenthesis of the answer node is in block $b(v)$ or $b(\text{bp\_close}(v))$, the query can be answered using the aB-trees. Otherwise, the bitmap B' is used to find the block that contains the opening parenthesis of the answer node, and then the aB-tree of this block is used to locate the answer.

We now describe our representation for weighted trees. Let T be a weighted tree, and let BP be the balanced parentheses representation of T. Define a string P of length $2n$ in which each character is a tuple of $s + 1$ elements. For an index i, the tuple $P[i]$ is $(w_1(v), w_2(v), \ldots, w_s(v), BP[i])$, where v is the node that corresponds to $BP[i]$. As for the case of unweighted trees, it suffices to support a set of base queries that include (1) the weighted tree queries deg, child_rank, and child_select (2) sum, fwd_search and bwd_search queries on the string P and a function f (from a fixed set of functions $\mathcal{F}$) which has the form

$$\phi_{a,b}(x) = \begin{cases} x_a & \text{if } x_{s+1} = 1 \\ x_b & \text{if } x_{s+1} = 0 \end{cases} \qquad \text{or} \qquad \pi_a(x) = \begin{cases} x_a & \text{if } x_{s+1} = 1 \\ -x_a & \text{if } x_{s+1} = 0 \end{cases}$$

where x_a denotes the a-th coordinate of x. Note that we do not support rmqi and RMQi queries as these queries are needed only to answer tree queries that are not supported by our weighted tree representation (e.g., lca and height queries). Recall that the representation of Navarro and Sadakane relies on the property that the image of the function f is $\{-1, 0, 1\}$, which is not true in our case. Therefore, instead of supporting the fwd_search and bwd_search queries of Navarro and Sadakane, we support the following modified queries

$$\text{fwd\_search}'(P, f, i, d) = \min\{j \geq i : \text{sum}(P, f, i, j) \geq d\}$$
$$\text{bwd\_search}'(P, f, i, d) = \max\{j \leq i : \text{sum}(P, f, i, j) \geq d\}$$

It is easy to verify that the queries of Table 2, can be answered using the modified base queries. For example, $\text{level\_ancestor}_a(v, i) = \text{bwd\_search}'(P, \pi_a, v - 1, i)$.

To support the base queries, P is partitioned into blocks of size $N = D^c$ and each block is stored using an aB-tree. The space of one aB-tree is $N \log(2X^s) + O(1)$. Thus, the total space of the aB-trees is $2n \log(2X^s) + o(n)$. Additionally, since now the image of a function f is $\{-X, \ldots, X\}$ whereas the image is $\{-1, 0, 1\}$ for unweighted trees, the space of the additional data-structures is increased by a factor of at most X. Since $X = O(\log n)$, we can ensure the additional space is $o(n)$ by increasing c by 1.

In the aB-tree of a block Q we store at each node x the values $e_f[x]$ and $m_f[x]$ for every function f, as before. Additionally, for every weight function w_a, we store a value $n_a[x]$ which is equal to the w_a-weight of the set of nodes v for which $l_x \leq \text{bp\_close}(v) \leq r_x$ and $E_\pi[\text{bp\_close}(v)] = m_\pi[x]$. Performing queries on Q is almost the same as in the unweighted case. For a $\text{fwd\_search}'(Q, f, i, d)$ query, during the traversal of the aB-tree we need to compute at a node x the minimum index $k \geq i'$ such that $d' \geq m_f[x_k]$. For a $\deg_a(v)$

query, during the traversal of the aB-tree, the values $n_a[z_j]$ are added to the counter for every $j > i$ (or for every $j < i$) for which $m_\pi[z_j] = \text{depth}(v)$. Since $X = O(\log n)$, the sizes of the lookup tables remain $o(n)$.

For answering fwd_search$'$ queries on P, we build T_f trees. The edge weights of these trees are bounded by NX. Therefore one tree takes $O((n/N)\log(n/N)\log(nX) + nX/\log^t(nX))$ bits (where t is some constant integer), and the total space is $o(n)$. As for the structures for child queries, we now need to store the w_a-degrees of the marked nodes for every weight function w_a. The space is $O((n/N)\log(nX)) = o(n)$ bits. Moreover, we need a rank-select structure on a bitmap B'_a which is defined similarly to B', except that now c_i is the w_a-weight of the children of v whose opening parentheses are in the i-th block that is contained in v. The space of this structure is $(n/N)\log(NX)+O(n/N+NX/\log^t(NX)) = o(n)$ bits.

5 Proof of Lemma 6

The decomposition is based on the decomposition of Farzan and Munro [5]. For completeness, we first describe the latter decomposition. During the run of the algorithm, two kinds of subtrees of T are maintained: temporary and permanent. A permanent tree does not change after it is created, and at the end of the algorithm, the decomposition consists of all permanent trees. A temporary tree has at most L nodes, and a permanent tree has at most $2L$ nodes.

The algorithm uses a procedure pack$(v, u_1, \ldots, u_k)$ that receives a node v and some children $u_1, \ldots, u_k$ of v, where each u_i is a root of a temporary tree. The procedure combines the temporary trees into larger trees as follows.

1. Let u_i be the first unhandled node (initially, $i = 1$). Add v to the temporary tree of u_i.
2. Combine the tree of u_i with the trees of $u_{i+1}, u_{i+2}, \ldots$ (by adding the nodes of the latter trees to the former tree) and stop when the combined tree has at least L nodes, or when there are no more children of v.
3. If $i = 1$ and the size of the combined tree is less than L, declare the tree temporary. Otherwise, declare the tree permanent and go to step 1.

We say that a node v is *heavy* if the number of its descendants (including v) is at least L. The decomposition algorithm works recursively on the tree T, starting from the root. The handling of a node v is done as follows.

1. If v is a leaf, create a temporary tree that consists of v, and stop the recursive call.
2. Otherwise, let $u_1, \ldots, u_k$ be the children of v. Run the algorithm recursively on $u_1, \ldots, u_k$. After the recursive calls, each u_i is a root of a subtree of T.
3. If v has no heavy children, call pack$(v, u_1, \ldots, u_k)$.
4. If v has a single heavy child u_i, then if the subtree of u_i is temporary, call pack$(v, u_1, \ldots, u_k)$. Otherwise, call pack$(v, u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_k)$.
5. If v has at least 2 heavy children, let $u_{i_1}, \ldots, u_{i_d}$ be the heavy children of v. Declare the subtrees of $u_{i_1}, \ldots, u_{i_d}$ permanent. If all the children of v are heavy, create a temporary tree that consists of v. Otherwise, for every $j < d$, call pack$(v, u_{i_j+1}, \ldots, u_{i_{j+1}-1})$.

An example for the decomposition of Farzan and Munro is given in Figure 1a.

We now describe our decomposition. We first construct the decomposition of Farzan and Munro, which satisfies the properties of the lemma, except for property 1. It also satisfies stronger versions of properties 2 and 5: Each tree in the decomposition has size at most $2L$, and the common node of two trees in the decomposition can be only the root of both trees. Moreover, for a tree T' in the decomposition, for every edge (v, w) between a node $v \in T'$ and a node $w \notin T'$, v is the root of T', except perhaps for one edge.

We now change the decomposition as follows (see Figure 1b). First, for every edge (v, w) for which v and w are in different trees, if v is not the root of a tree in the decomposition, add the node w to the unique tree containing v. Note that in this case, the tree also contains the predecessor of v in the preorder, so property 4 is maintained. Otherwise (if v is the root of a tree), add a new tree to the decomposition that consists of the nodes v and w. After the first step is performed, remove from the decomposition all trees that consist of a single node. It is easy to verify that the new decomposition satisfies all the properties of the lemma.

References

[1] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.

[2] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. 20th European Symposium on Algorithms (ESA)*, pages 181–192, 2012.

[3] 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.

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

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

[6] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. of the ACM*, 57(1), 2009.

[7] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[8] M. He, J. I. Munro, and G. Zhou. A framework for succinct labeled ordinal trees over large alphabets. In *Proc. 23rd International Symposium on Algorithms and Computation (ISAAC)*, pages 537–547, 2012.

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

[10] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

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

[12] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

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