A Framework for Succinct Labeled Ordinal Trees over Large Alphabets*

Meng He[1], J. Ian Munro[2], and Gelin Zhou[2]

[1] Faculty of Computer Science, Dalhousie University, Canada.
mhe@cs.dal.ca
[2] David R. Cheriton School of Computer Science, University of Waterloo, Canada.
{imunro, g5zhou}@uwaterloo.ca

Abstract. We consider succinct representations of labeled ordinal trees that support a rich set of operations. Our new representations support a much broader collection of operations than previous work [10, 8, 1]. In our approach, labels of nodes are stored in a *preorder label sequence*, which can be compressed using any succinct index for strings that supports $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations. In other words, we present a framework for succinct representations of labeled ordinal trees that allows alphabets to be large. This answers an open problem presented by Geary et al. [10]. We further extend our work and present the first succinct representation of dynamic labeled ordinal trees that supports several label-based operations including finding the level ancestor with a given label.

1 Introduction

We address the issue of succinct representations of ordinal (or ordered) trees with satellite data over a large alphabet. Much of this is motivated by the needs of large text-dominated databases that store and manipulate XML documents, which can be essentially modeled as ordinal trees in which each node is assigned a tag drawn from a tag set.

Our representations support a much broader collection of operations than previous work [10, 8, 1], particularly those operations that aim at XML-style document retrieval, such as queries written in the XML path language (XPath). Our data structures are succinct, occupying space close to the information-theoretic lower bound, which are essential to systems and applications that deal with very large data sets.

Our approach is based on "tree extraction", that is, constructing subtrees consisting of nodes with appropriate labels (and their parents). The basic idea of tree extraction was introduced by He et al. [14, 15], where it was used to answer queries that are generalizations of geometric queries such as range counting from planar point sets to trees. Here we follow a different approach for a completely different class of operations that originate from text databases. Most of these operations are not required in [14, 15]. In our data structures, an input

* This work was supported by NSERC and the Canada Research Chairs Program.

tree is split according to labels of nodes, such that we can maintain structural information and labels jointly in a space-efficiently way. Previous solutions to the same problem are all based on different ideas [10, 8, 1].

In this paper, we consider the operations listed in Table 1, in which DFUDS denotes *depth-first unary degree sequence* order as defined by Benoit et al. [3]. We list only the labeled versions of these operations. The unlabeled versions simply include all nodes. In other words, the support for unlabeled versions can be reduced to the support for labeled versions by setting the alphabet size to 1. For simplicity, we refer to the labeled versions of operations as α-operations. We call a node that has label α an α-node (hence α-children, α-ancestor, etc). In addition, we define the α-rank of node x in a list to be the number of α-nodes to the left of x in the list.

Operation	Description
$\texttt{depth}_\alpha(x)$	α-depth of x, i.e., number of α-nodes from x to the root
$\texttt{parent}_\alpha(x)$	closest α-ancestor of x
$\texttt{level\_anc}_\alpha(x, i)$	α-ancestor y of x satisfying $\texttt{depth}_\alpha(x) - \texttt{depth}_\alpha(y) = i$
$\texttt{deg}_\alpha(x)$	number of α-children of x
$\texttt{child\_rank}_\alpha(x)$	α-rank of x in the list of children of $\texttt{parent}(x)$
$\texttt{child\_select}_\alpha(x, i)$	i-th α-child of x
$\texttt{nbdesc}_\alpha(x)$	number of α-nodes in the subtree rooted at x
$\texttt{pre\_rank}_\alpha(x)$/$\texttt{pre\_select}_\alpha(i)$	α-rank of x in preorder/i-th α-node in preorder
$\texttt{post\_rank}_\alpha(x)$/$\texttt{post\_select}_\alpha(i)$	α-rank of x in postorder/i-th α-node in postorder
$\texttt{height}_\alpha(x)$	α-height of x, i.e., maximum number of α-nodes from x to its leaf descendant
$\texttt{LCA}_\alpha(x, y)$	lowest common α-ancestor of nodes x and y
$\texttt{dfuds\_rank}_\alpha(x)$/$\texttt{dfuds\_select}_\alpha(i)$	α-rank of x in DFUDS order/i-th α-node in DFUDS order
$\texttt{leaf\_lmost}_\alpha(x)$/$\texttt{leaf\_rmost}_\alpha(x)$	leftmost/rightmost α-leaf in the subtree rooted at x
$\texttt{leaf\_rank}_\alpha(x)$	number of α-leaves to the left of x in preorder
$\texttt{leaf\_select}_\alpha(i)$	i-th α-leaf in preorder
$\texttt{nbleaf}_\alpha(x)$	number of α-leaves in the subtree rooted at node x
$\texttt{insert}_\alpha(x)$	insert an α-node x as an internal node or a leaf
$\texttt{delete}(x)$	delete non-root node x

Table 1. Operations considered in this paper. Here we give only the definitions of the labeled versions of operations.

Geary et al. [10] presented data structures to support in constant time the first group of α-operations in Table 1 and their unlabeled versions. The overall space cost of their data structures is $n(\lg\sigma+2)+O(\frac{n\sigma\lg\lg\lg n}{\lg\lg n})$ bits, which is much more than the information-theoretic lower bound of $n(\lg\sigma+2)-O(\lg n)$ bits when $\sigma = \Omega(\lg\lg n)$. Ferragina et al. [8] and Barbay et al. [1] designed data structures for labeled trees that use space close to the information-theoretic lower bound, but supporting a more restricted set of α-operations. Ferragina et al.'s [8] xbw-based representation supports only $\texttt{child\_select}_\alpha(x, i)$ and $\texttt{deg}_\alpha(x)$ [1], while Barbay et al.'s [1] data structure supports only $\texttt{pre\_rank}_\alpha(x)$, $\texttt{pre\_select}_\alpha(i)$ and $\texttt{nbdesc}_\alpha(x)$. These results are for static labeled trees; to the best of our knowledge, there is no succinct data structure for dynamic labeled ordinal trees with efficient query and update time.

[1] It also supports SubPathSearch queries, which return the number of nodes whose upward paths start with a given query string.

Our results for static and dynamic labeled ordinal trees are summarized in Theorems 1 to 3. First, as a preliminary result, we improve the succinct representation of labeled ordinal trees of Geary et al. [10]. As shown in Theorem 1, the improved representation supports more operations while occupying less space, where PLS_T is the preorder label sequence of T, and H_k is the *k-th order empirical entropy* [16], which is bounded above by $\lg \sigma$. However, this data structure is succinct only if the size of alphabet is very small, i.e., $\sigma = o(\lg \lg n)$.

$\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$ are auxiliary α-operations used in Section 3: $\texttt{deepest}_\alpha(i, j)$ returns a node (there could be a tie) with preorder rank in $[i, j]$ that has the maximum α-depth, and $\texttt{min\_depth}_\alpha(i, j)$ returns an α-node with preorder rank in $[i, j]$ that has the minimum depth (i.e., is closest to the root).

Theorem 1. *Let T be a static ordinal tree on n nodes, each having a label drawn from an alphabet of size $\sigma = o(\lg \lg n)$. Under the word RAM with word size $w = \Omega(\lg n)$, for any $k = o(\log_\sigma n)$, there exists a data structure that encodes T using $n(H_k(PLS_T) + 2) + O(\frac{n(k \lg \sigma + \lg \lg n)}{\log_\sigma n}) + O(\frac{n\sigma \lg \lg \lg n}{\lg \lg n})$ bits of space, supporting the first two groups of α-operations in Table 1 and their unlabeled versions, plus two additional α-operations $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$, in constant time.*

Theorem 2 is the main result in this paper. To achieve this result, we present a framework for succinct representations of labeled ordinal trees, in which an α-operation is reduced to a constant number of well-supported operations on simpler data structures such as bit vectors, preorder label sequences, and unlabeled and 0/1-labeled ordinal trees, where a 0/1-labeled ordinal tree is an ordinal tree over the alphabet $\{0, 1\}$. This creative reduction allows us to deal with large alphabets, and to compress labels of nodes into entropy bounds. Thus our framework answers an open problem proposed by Geary et al. [10].

Theorem 2. *Let T be a static ordinal tree on n nodes, each having a label drawn from an alphabet Σ of size σ. Under the word RAM with word size $w = \Omega(\lg n)$,*

(a) for $\sigma = O(\texttt{polylog}(n))$, T can be encoded using $n(H_0(PLS_T) + 9) + o(n)$ bits of space to support the first two groups of α-operations in Table 1 in constant time;
(b) for general Σ, T can be encoded using $nH_0(PLS_T) + O(n)$ bits of space to support the first two groups of α-operations in Table 1 in $O(\lg \lg \sigma)$ time;
(c) for general Σ and $k = o(\log_\sigma n)$, T can be encoded using $nH_k(PLS_T) + \lg \sigma \cdot o(n) + O(\frac{n \lg \sigma}{\lg \lg \lg \sigma})$ bits to support the first two groups of α-operations in Table 1 in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time.

In addition, these data structures support the unlabeled versions of these α-operations in constant time.

Theorem 3 further extends our work to the dynamic case. Here we only list the result with the fastest query time. One can make use of the dynamic strings in [12, 17] when worst-case update time is desired.

Theorem 3. *Let T be a dynamic ordinal tree on n nodes, each having a label drawn from an alphabet Σ of size σ. Under the word RAM with word size*

$w = \Omega(\lg n)$, T can be represented using $n(H_0(PLS_T)+5)+O(\frac{n(1+H_0(PLS_T))}{\lg^{1-\epsilon} n} + \sigma(\lg\sigma + \lg^{1+\epsilon} n))$ bits of space, for any constant $0 < \epsilon < 1$, such that $\texttt{depth}_\alpha$, $\texttt{parent}_\alpha$, $\texttt{nbdesc}_\alpha$, $\texttt{LCA}_\alpha$, $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$, $\texttt{post\_rank}_\alpha$, $\texttt{post\_select}_\alpha$, and the $\texttt{leaf}$ α-operations can be supported in $O(\frac{\lg n}{\lg\lg n})$ time, $\texttt{level\_anc}_\alpha$ can be supported in $O(\lg n)$ time, and $\texttt{insert}_\alpha$ and $\texttt{delete}$ can be supported in $O(\frac{\lg n}{\lg\lg n})$ amortized time.

This rest of this paper is organized as follows. In Section 2 we review the data structures and the techniques used in this paper. In Section 3 we describe the construction of our data structures for static trees over large alphabets, i.e., the proof of Theorem 2. In Section 4, we sketch our data structures for dynamic trees, i.e., the proof of Theorem 3. Due to space limitations, we omit the proof of Theorem 1.

2 Preliminaries

2.1 Bit Vectors, Strings and the Related Operations

In this subsection, we review bit vectors, strings, and the operations performed on them. Bit vector plays a central role in many succinct data structures. For a bit vector $B[1..n]$, $\texttt{rank}_0(i)$ and $\texttt{rank}_1(i)$ return the numbers of 0-bits and 1-bits in $B[1..i]$, respectively. $\texttt{select}_0(i)$ and $\texttt{select}_1(i)$ return the positions of the i-th 0-bit and the i-th 1-bit in B, respectively. Bit vectors can be generalized to strings, in which characters are drawn from an alphabet Σ of size σ. For a string $S[1..n]$ and $\alpha \in \Sigma$, $\texttt{rank}_\alpha(i)$ returns the number of α's in $S[1..i]$, and $\texttt{select}_\alpha(i)$ returns the position of the i-th α. Another operation studied by researchers is the random access to any substring of length $O(\log_\sigma n)$.

2.2 Tree Extraction

Tree extraction [14, 15], based on the deletion operation of tree edit distance [4], is a technique used to decompose a tree by deleting nodes, moving their children into their positions in the sibling order. A crucial fact is that the ancestor-descendant and preorder/postorder relationships between the remaining nodes are preserved. To support the α-operations related to children, we develop a new space-efficient approach based on tree extraction that is very different from the strategy used in [14, 15], so that the parent-child relationship is preserved.

Let $V(T)$ be the set of nodes in T. For any set $X \subseteq V(T)$ that contains the root of T, we denote by T_X the ordinal tree obtained by deleting all the nodes that are not in X from T, where the nodes are deleted in level order. T_X is called the *X-extraction* of T. It is easy to see that there is a natural one-to-one correspondence between the nodes in X and the nodes in T_X. Lemma 1 captures an essential property of tree extraction. The proof is omitted here.

Lemma 1. *For any two sets of nodes $X, X' \subseteq V(T)$, the nodes in $X \cap X'$ have the same relative positions in the preorder and the postorder traversal sequences of T_X and $T_{X'}$.*

3 Static Trees over Large Alphabets : Theorem 2

For each possible subscript $\alpha \in \Sigma$, we could support the first two groups of operations in Table 1 by relabeling T into a 0/1-labeled tree and indexing the relabeled tree, where a node is relabeled 1 if and only if it is an α-node in T. However, we would have to store σ trees that have $n\sigma$ nodes in total if we simply apply this idea for each $\alpha \in \Sigma$, which we could not afford.

Instead of storing all the n nodes for each $\alpha \in \Sigma$, we store only the nodes that are closely relevant to α, i.e., the α-nodes and their parents, and the ancestor-descendant relationship between these nodes. We apply tree extraction to summarize the information, where the tree constructed for label α is denoted by T_α.

For $\alpha \in \Sigma$, we create a new root r_α, and make the original root of T be the only child of r_α. The structure of T_α is obtained by computing the X_α-extraction of the augmented tree rooted at r_α, where X_α is the union of r_α, the α-nodes in T, and the parents of the α-nodes. The natural one-to-one mapping between the nodes in X_α and T_α determines the labels of the nodes in T_α. The root of T_α is always labeled 0. A non-root node in T_α is labeled 1 if its corresponding node in T is an α-node, and 0 otherwise. Thus, the number of 1-nodes in T_α is equal to the number of α-nodes in T. Let n_α denote both values.

To clarify notation, the nodes in T are denoted by lowercase letters, while the nodes in T_α are denoted by lowercase letters plus prime symbols. To illustrate the one-to-one mapping, we denote by x' a node in T_α if and only if its corresponding node in T is denoted by x. The root of T_α, which corresponds to r_α, is denoted by r'_α. We show how to convert the corresponding nodes in T and T_α using the preorder label sequence of T in Subsection 3.2.

Since the structure of T_α is different from T, we need also store the structure of T and the labels of the nodes in T so that we can perform conversions between the nodes in T and T_α. In addition, to support the leaf α-operations, we store for each $\alpha \in \Sigma$ a bit vector $L_\alpha[1..n_\alpha]$ in which the i-th bit is one if and only if the i-th 1-node in preorder of T_α corresponds to a leaf in T.

Following this approach, our succinct representation consists of four components: (a) the structure of T; (b) PLS_T, the preorder label sequence of T; (c) a 0/1-labeled tree T_α for each $\alpha \in \Sigma$; (d) and a bit vector L_α for each $\alpha \in \Sigma$. The unlabeled versions of the first two groups of operations in Table 1 are directly supported by the data structure that maintains the structure of T. For α-operations, our basic idea is to reduce an α-operation to a constant number of well-supported operations on T, PLS_T, T_α's, and L_α's, for which we summarize the previous work in Subsection 3.5. In the following subsections, we describe our algorithms in terms of T, PLS_T, T_α's and L_α's. For each operation, we specify the component on which it performs as the first parameter. If such a component is not specified in context, then this operation is performed on T.

3.1 $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$ and $\texttt{nbdesc}_\alpha$

By the definitions, we have $\texttt{pre\_rank}_\alpha(x) = \texttt{rank}_\alpha(PLS_T, \texttt{pre\_rank}(x))$ and $\texttt{pre\_select}_\alpha(i) = \texttt{pre\_select}(\texttt{select}_\alpha(PLS_T, i))$. We make use of them to

find the α-predecessor and the α-successor of node x, which are defined to be the last α-node preceding x and the first α-node succeeding x in preorder (both can be x itself).

We support $\texttt{nbdesc}_\alpha(x)$ in the same way as [1]. The descendants of x form a consecutive substring in PLS_T, which starts at index $\texttt{pre\_rank}(x)$ and ends at index $\texttt{pre\_rank}(x) + \texttt{nbdesc}(x) - 1$. We can compute the number of α-nodes lying in this range using $\texttt{rank}_\alpha$ on PLS_T. Providing that x has an α-descendant, we can further compute the first and the last α-descendant of node x in preorder, which are the α-successor of x and the α-predecessor of the node that has preorder rank $\texttt{pre\_rank}(x) + \texttt{nbdesc}(x) - 1$, respectively. Let these α-nodes be u and v. For simplicity, we call $[u, v]$ the α-boundary of the subtree rooted at x.

3.2 Conversion between the nodes in T and T_α

The conversion between node x in T and the corresponding node x' in T_α plays an important role in supporting α-operations. If x is an α-node, then x' must exist in T_α. By Lemma 1, the conversion can be done using $x' = \texttt{pre\_select}_1(T_\alpha, \texttt{rank}_\alpha(PLS_T, x))$, and $x = \texttt{pre\_select}_\alpha(\texttt{pre\_rank}_1(T_\alpha, x'))$.

The other case in which x is not an α-node is more complex, since the node in T_α that corresponds to x may or may not exist. By the definition of T_α, x must have an α-child if x' exists in T_α. In addition, every α-child of x in T must appear as a 1-child of x' in T_α. Using this, we can compute x from x': We first find the first 1-child of x', say $y' = \texttt{child\_select}_1(T_\alpha, x', 1)$, and compute node y in T that corresponds to y' using the equation in the first paragraph. Then x must be the parent of y in T.

Algorithm 1: Compute x' when x is not an α-node

```
1  if x has no α-descendant then return NULL;
2  [u, v] ← the α-boundary of the subtree rooted at x;
3  if x ≠ LCA(u, v) then
4  |   y ← the child of x that is an ancestor of LCA(u, v);
5  |   if y is an α-node then return parent(T_α, y');
6  |   else return NULL;
7  else
8  |   w' = LCA(T_α, u', v');
9  |   if w' corresponds to x then return w';
10 |   else return NULL;
```

Algorithm 1 shows how to compute x' from x when x is not an α-node, and it returns NULL if no such x' exists. We first verify whether x has at least one α-descendant in line 1 using $\texttt{nbdesc}_\alpha$. x' does not exist in T_α if x has no α-descendant. Otherwise, we compute the α-boundary, $[u, v]$, of the subtree rooted at x in line 2. We have two cases, depending on whether x is the lowest common ancestor of u and v. If $x \neq \texttt{LCA}(u, v)$, then x must have a child y that is a common ancestor of u and v. All α-descendants of x must also be descendants of y, or an α-descendant of x may precede u or succeed v in preorder. Thus,

we need only check if y is an α-node, as shown in lines 5 and 6. Now suppose $x = \texttt{LCA}(u, v)$. We claim that x' must be the lowest common ancestor of u' and v' if x' exists in T_α. Thus, we need only compute $w' = \texttt{LCA}(T_\alpha, u', v')$ in line 8, and verify if w' corresponds to x in lines 9 and 10.

3.3 $\texttt{parent}_\alpha$, $\texttt{level\_anc}_\alpha$, $\texttt{LCA}_\alpha$ and $\texttt{depth}_\alpha$

We first show how to compute $\texttt{parent}_\alpha(x)$ in Algorithm 2. The case in which x is an α-node is solved in lines 2 to 3. We simply compute $y' = \texttt{parent}_1(T_\alpha, x')$ and return y. Suppose x is not an α-node. We compute u, the α-predecessor of x in preorder, in line 4. We claim that x has no α-parent if x has no α-predecessor in preorder, since the ancestors of x precede x in preorder. If such u exists, we take a look at $v = \texttt{LCA}(u, x)$ in line 6. We further claim that there is no α-node on the path between v and x (excluding v), because u would not be the α-predecessor if such an α-node exists. In addition, we know that v has at least one α-descendant because of the existence of u. We return v if v is an α-node. Otherwise, we compute the first α-descendant, w, of v. It is clear that there is no α-node on the path between w and v (excluding w). Thus, the α-parent of w, being computed in line 9, must be the α-parent of both v and x.

Algorithm 2: $\texttt{parent}_\alpha(x)$

```
1  if x is an α-node then
2  |   y' ← parent_1(T_α, x');
3  |   return y;
4  u ← the α-predecessor of x in preorder of T;
5  if x has no α-predecessor then return NULL;
6  v ← LCA(u, x);
7  if v is an α-node then return v;
8  w ← the first α-descendant of v in preorder;
9  y' ← parent_1(T_α, w');
10 return y;
```

Then we make use of $\texttt{parent}_\alpha(x)$ to support $\texttt{level\_anc}_\alpha(x, i)$: We first compute $y = \texttt{parent}_\alpha(x)$, where y must be an α-node or NULL. We return y if $y = \texttt{NULL}$ or $i = 1$. Otherwise, we compute $z' = \texttt{level\_anc}_1(T_\alpha, y', i - 1)$ and return z.

$\texttt{LCA}_\alpha$ and $\texttt{depth}_\alpha$ can also be easily supported using $\texttt{parent}_\alpha$. $\texttt{LCA}_\alpha(x, y)$ is equal to $\texttt{LCA}(x, y)$ if the lowest common ancestor of x and y is an α-node; otherwise it is equal to $\texttt{parent}_\alpha(\texttt{LCA}(x, y))$. To compute $\texttt{depth}_\alpha(x)$, let $y = x$ if x is an α-node, or $y = \texttt{parent}_\alpha(x)$ if x is not. It is then clear that $\texttt{depth}_\alpha(x) = \texttt{depth}_\alpha(y) = \texttt{depth}_1(T_\alpha, y')$, since each α-ancestor of y in T corresponds to a 1-ancestor of y' in T_α.

3.4 $\texttt{child\_rank}_\alpha$, $\texttt{child\_select}_\alpha$ and $\texttt{deg}_\alpha$

We can support $\texttt{child\_select}_\alpha(x, i)$ and $\texttt{deg}_\alpha(x)$ using the techniques shown in Subsection 3.2. We first try to find x', the node in T_α that corresponds to

x. If such x' does not exist, then x must have no α-child. Thus, we return NULL for $\texttt{child\_select}_\alpha(x, i)$ and return 0 for $\texttt{deg}_\alpha(x)$. Otherwise, we compute $y' = \texttt{child\_select}_1(T_\alpha, x', i)$ and return y for $\texttt{child\_select}_\alpha(x, i)$, as well as return $\texttt{deg}_1(T_\alpha, x')$ for $\texttt{deg}_\alpha(x)$.

Algorithm 3: $\texttt{child\_rank}_\alpha(x)$

```
1 if x is an α-node then return child_rank_1(T_α, x');
2 u ← parent(x);
3 if u has no α-child then return 0;
4 v ← the α-predecessor of x in preorder;
5 if x has no α-predecessor or pre_rank(v) ≤ pre_rank(u) then return 0;
6 compute u' and v', the nodes in T_α that correspond to u and v;
7 w' ← the child of u' that is an ancestor of v';
8 return child_rank_1(T_α, w');
```

The algorithm to support $\texttt{child\_rank}_\alpha(x)$ is shown as Algorithm 3. The case in which x is an α-node is easy to handle, as shown in line 1. We consider only the case in which the label of x is not α. In lines 2 to 3, we compute node u that is the parent of x, and verify if u has an α-child using $\texttt{deg}_\alpha$. We return 0 if u has no α-child. Otherwise, we compute the α-predecessor, v, of x in preorder. If such v does not exist, or v is not a proper descendant of u, then x has no α-sibling preceding it and we can return 0, since an sibling preceding x occurs before x in preorder. Suppose v exists as a proper descendant of u. We can find u' and v', the nodes in T_α that correspond to u and v, since both u and v are α-nodes. In addition, we find the child, w', of u' that is an ancestor of v'. We claim that each α-child of u in T that precedes x corresponds to a 1-child of u' in T_α that precedes w'. Otherwise, v would not be the α-predecessor. Finally, we return $\texttt{child\_rank}_1(T_\alpha, w')$ as the answer.

3.5 Completing the Proof of Theorem 2

Due to space limitations, the support for the other static α-operations is omitted here. In the current state, we have σ 0/1-labeled trees and σ bit vectors. To reduce redundancy, we merge T_α's into a single tree $\mathcal{T}$, and merge L_α's into a single bit vector $\mathcal{L}$. We list the characters in Σ as $\alpha_1, \cdots, \alpha_\sigma$. Initially, $\mathcal{T}$ contains a root node $\mathcal{R}$ only, on which the label is 0. Then, for $i = 1$ to σ, we append r'_{α_i}, the root of T_{α_i}, to the list of children of $\mathcal{R}$. Let n_α be the number of α-nodes in T. For $\alpha \in \Sigma$, T_α has at most $2n_\alpha + 1$ nodes, since each α-node adds a 1-node and at most one 0-node into T_α. In addition, the T_α that corresponds to the label of the root of T has at most $2n_\alpha$ nodes, since the root does not add an 0-node to T_α. Hence, $\mathcal{T}$ has at most $2n+\sigma$ nodes in total. By the construction of $\mathcal{T}$, the preorder/postorder traversal sequence of T_α occurs as a substring in the preorder/postorder traversal sequence of $\mathcal{T}$. Also, the DFUDS traversal sequence of T_α with r'_α removed occurs as a substring in the DFUDS traversal sequence of $\mathcal{T}$.

In addition, we append L_{α_i} to $\mathcal{L}$, which is initially empty, for $i = 1$ to σ. The

length of $\mathcal{L}$ is clearly n. It is not hard to verify that the reductions described in early subsections can still be performed on the merged tree $\mathcal{T}$ and the merged bit vector $\mathcal{L}$. The following lemma generalizes the discussion.

Lemma 2. *Let T be an ordinal tree on n nodes, each having a label drawn from an alphabet Σ of size σ. Suppose that there exist*

- *a data structure $\mathcal{D}_1$ that represents a unlabeled ordinal tree on n nodes using $\mathcal{S}_1(n)$ bits and supports the unlabeled versions of the first two groups of α-operations in Table 1;*
- *a data structure $\mathcal{D}_2$ that represents a string S using $\mathcal{S}_2(S)$ bits and supports $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ for $\alpha \in \Sigma$;*
- *a data structure $\mathcal{D}_3$ that represents a 0/1-labeled ordinal tree on n nodes using $\mathcal{S}_3(n)$ bits and supports the first two groups of α-operations in Table 1 and their unlabeled versions, plus two additional α-operations $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$;*
- *and a data structure $\mathcal{D}_4$ that represents a bit vector of length n using $\mathcal{S}_4(n)$ bits and supports $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ for $\alpha \in \{0, 1\}$.*

Then there exists a data structure that encodes T using $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_3(2n + \sigma) + \mathcal{S}_4(n)$ bits of space, supporting the first two groups of α-operations in Table 1 and their unlabeled versions using a constant number of operations mentioned above on $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$ and $\mathcal{D}_4$.

Proof. The unlabeled versions are supported by $\mathcal{D}_1$ directly. The reductions for the α-operations are shown in Subsections 3.1 to 3.4, and more details are omitted due to space limitations. Here we consider the space cost only. We maintain the structure of T, PLS_T, $\mathcal{T}$, and $\mathcal{L}$ using $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$, respectively. The overall space cost is $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_3(2n + \sigma) + \mathcal{S}_4(n)$ bits. □

With the following three lemmas the proof of Theorem 2 follows.

Lemma 3 ([9, 11, 2]). *Let S be a string of length n over an alphabet of size σ. Under the word RAM with word size $w = \Omega(\lg n)$,*

(a) for $\sigma = O(\texttt{polylog}(n))$, S can be represented using $nH_0(S) + o(n)$ bits of space to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in $O(1)$ time;
(b) for general Σ, S can be represented using $nH_0(S) + O(n)$ bits of space to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in $O(\lg \lg \sigma)$ time;
(c) for any $k = o(\log_\sigma n)$, S can be represented using $nH_k(S) + \lg \sigma \cdot o(n) + O(\frac{n \lg \sigma}{\lg \lg \lg \sigma})$ bits to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time.

Lemma 4 ([5]). *A bit vector of length n can be represented in $n + o(n)$ bits to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in constant time, for $\alpha \in \{0, 1\}$.*

Lemma 5 ([13, 6, 7, 17]). *Let T be an ordinal tree on n nodes. T can be represented using $2n + o(n)$ bits such that the unlabeled versions of the first two groups of α-operations in Table 1 can be supported in constant time.*

Proof (Theorem 2). Applying Lemma 5, one of Lemma 3 (a,b,c), Theorem 1, and Lemma 4 for $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$, respectively, we obtain the conclusion. □

4 Dynamic Trees that Support Level-Ancestor Operations : Theorem 3

Our succinct representation of dynamic trees also consists of four components: the structure of T, PLS_T, T_α's and L_α's. The construction of T_α is different in this scenario in order to facilitate update operations. For each $\alpha \in \Sigma$, we still add a new root r_α to T, and compute the X_α-extraction of the augmented tree rooted at r_α. However, X_α contains r_α and the α-nodes in T only, and we do not assign labels to the nodes in T_α. Finally, we still merge T_α's into a single tree $\mathcal{T}$, and merge L_α's into a single bit vector $\mathcal{L}$. $\mathcal{T}$ has exactly $n+\sigma+1$ nodes. We omit the details of supporting operations due to space constraints.

References

1. Barbay, J., Golynski, A., Munro, J.I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theor. Comput. Sci. 387(3), 284–297 (2007)
2. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
3. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
4. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
5. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage (extended abstract). In: SODA. pp. 383–391 (1996)
6. Farzan, A., Munro, J.I.: A uniform approach towards succinct representation of trees. In: SWAT. pp. 173–184 (2008)
7. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: ICALP (1). pp. 451–462 (2009)
8. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM 57(1) (2009)
9. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
10. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
11. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA. pp. 368–373 (2006)
12. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: SPIRE. pp. 334–346 (2010)
13. He, M., Munro, J.I., Rao, S.S.: Succinct ordinal trees based on tree covering. In: ICALP. pp. 509–520 (2007)
14. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: ISAAC. pp. 140–149 (2011)
15. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: ESA. pp. 575–586 (2012)
16. Manzini, G.: An analysis of the burrows-wheeler transform. J. ACM 48(3), 407–430 (2001)
17. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA. pp. 134–149 (2010)