

Succinct ordinal trees with level-ancestor queries

RICHARD F. GEARY AND RAJEEV RAMAN

University of Leicester, UK

and

VENKATESH RAMAN

Institute of Mathematical Sciences, Chennai, India

We consider *succinct* or space-efficient representations of trees that efficiently support a variety of navigation operations. We focus on static *ordinal* trees, i.e., arbitrary static rooted trees where the children of each node are ordered. The set of operations is essentially the union of the sets of operations supported by previous succinct representations (Jacobson, *Proc. 30th FOCS*, 549-554, 1989; Munro and Raman, *SIAM J. Comput.* **31** (2001), 762-776; and Benoit et. al *Proc. 6th WADS*, LNCS 1663, 169-180, 1999), to which we add the *level-ancestor* operation.

Our representation takes $2n + o(n)$ bits to represent an n -node tree, which is within $o(n)$ bits of the information-theoretic minimum, and supports all operations in $O(1)$ time on the RAM model. These operations also provide a mapping from the n nodes of the tree onto the integers $\{1, \dots, n\}$. In addition to the existing motivations for studying such data structures, we are motivated by the problem of representing XML documents compactly so that XPath queries can be supported efficiently.

1. INTRODUCTION

Trees are a fundamental structure in computing. They are used in almost every aspect of modelling and representation for explicit computation. Their specific uses include searching for keys, maintaining directories, primary search structures for graphs, and representations of parsing – to name just a few. Explicit storage of trees, with a pointer per child as well as other structural information, is often taken as a given, but can account for the dominant storage cost.

This cost can be prohibitive (see [4, 9, 15, 16] for examples). Our focus is on static *ordinal* trees, i.e., arbitrary static rooted trees where the children of each node are ordered. Following the early work of Jacobson [15] there have been some more recent papers on *succinct* representations of static ordinal trees. These representations require

This research was supported in part by UISTRF grant 2001.04/IT and an EPSRC Doctoral Training Account grant.

Authors' addresses: Author1&2: Department of Computer Science, University of Leicester, Leicester LE1 7RH, UK. {r.geary, r.raman}@mcs.le.ac.uk. Author 3: Institute of Mathematical Sciences, Chennai, India 600 113. vraman@imsc.res.in.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2001 ACM 1073-0516/01/0300-0034 \$5.00

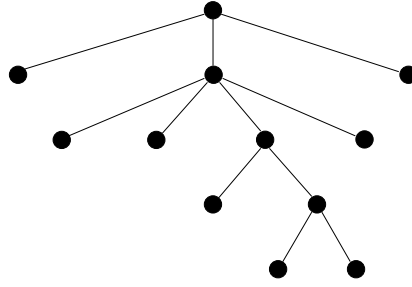
only $2n + o(n)$ bits to represent an n -node tree, but permit a reasonable class of primitive operations to be performed quickly (in $O(1)$ time). Since the information-theoretic lower bound on the space for representing an n -node ordinal tree is $\lceil \lg(\binom{2n+1}{n} / (2n+1)) \rceil = 2n - O(\lg(n))$ bits¹, this space bound is optimal to within lower-order terms. In this paper, we consider the following set of primitive operations:

- $\text{RANK}_z(x)$: return the position of node x in z -order, for $z \in \{pre, post\}$;
- $\text{SELECT}_z(i)$: return the i -th node in z -order, for $z \in \{pre, post\}$;
- $\text{CHILD}(x, i)$: for $i \geq 1$ return the i -th child of node x ;
- $\text{DEG}(x)$: return the number of child nodes of x ;
- $\text{CHILDRANK}(x)$: return i such that x is the i -th child of its parent;
- $\text{DEPTH}(x)$: return the depth of x , the length of the unique path from the root to x ;
- $\text{DESC}(x)$: return the number of descendants of x , including x itself (also called the *subtree size*); and
- $\text{ANC}(x, i)$: for $i \geq 0$ return the i -th ancestor of node x .

Representing a tree to support ANC queries is known as the *level-ancestor* problem, which has a number of algorithmic applications (see [7, 11, 12]). A number of $O(1)$ -time, $\Theta(n \lg n)$ -bit representations have been devised for this problem [1, 3, 6, 11].

Previous tree representations using $2n + o(n)$ bits only supported a proper subset of these operations. Jacobson's LOUDS (level order unary degree sequence) representation [15] represents a node of degree d as a string of d **1**s followed by a **0**; these nodes are then represented in a level order traversal of the tree. The parenthesis representation of Munro and Raman [16] represents each node as a pair of parentheses, with the children of each node represented recursively inside each pair. The parenthesis representation was augmented with additional operations by Chiang, Lin and Lu [7], and in recent independent work, by Munro and Rao [17]. The DFUDS (depth first unary degree sequence) representation [4, 5] is similar to LOUDS in that it represents each node of degree d as a sequence of d opening parentheses followed by a single closing parenthesis, but the sequence of nodes is represented in a depth first order. An example of each different representation is shown in Figure 1.1, and a summary of all the operations that they support is given in Table 1.1. It is worth noting that the LOUDS data structure supports an additional operation that allows enumeration of all nodes at a level, however, it is difficult to see how LOUDS could support e.g., DESC because the descendants of any one node are not kept contiguously.

¹ Throughout this paper we use " $\lg x$ " to denote " $\log_2 x$ ".



LOUDS: 1 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 0 0 1 1 0 0 0
 Parenthesis: (() (() () (() (() ())) ()) ())
 DFUDS: (((())) (((()))) (()) (()))))

Figure 1.1 Example ordinal tree of n nodes, with three different $2n$ -bit representations.

	$\{\text{RANK, SELECT}\}_{pre}$	$\{\text{RANK, SELECT}\}_{post}$	CHILD	DEG	CHILDRANK	DEPTH	DESC	ANC
LOUDS [15]	✗	✗	✓	✓	✓	✗	✗	✗
Parenthesis [16]	✓	✓	✗	✗	✗	✓	✓	✗
Parenthesis-a [8]	✓	✓	✗	✓	✗	✓	✓	✗
Parenthesis-b [17]	✓	✓	✗	✗	✗	✓	✓	✓
DFUDS [4, 5]	✓	✗	✓	✓	✓	✗	✓	✗
New	✓	✓	✓	✓	✓	✓	✓	✓

Table 1.1 Functionality of previous representations. An entry of ✓ or ✗ indicates whether the representation can or cannot support the operation in $O(1)$ time. All the above representations can find the parent in $O(1)$ time, and hence report $\text{ANC}(x, i)$ in $O(i)$ time. The parentheses representation can support $\text{CHILD}(x, i)$ in $O(i)$ time.

We give a $2n + O(n \lg \lg \lg n / \lg \lg n)$ -bit representation that supports all the above operations. As can be seen from Table 1.1 we provide all the functionality of the previous representations while adding the ANC operation. We assume, as do [4, 5, 16, 17], the unit-cost RAM model with word size $O(\lg n)$ bits². There is a natural mapping between a balanced string of $2n$ parentheses, where the first (opening) parenthesis matches the last (closing) parenthesis, and an n -node tree [16]. By applying this transformation, we are able to support natural operations on the parenthesis string, such as finding the closing parenthesis corresponding to an opening one, or finding the i -th outer enclosing parenthesis, extending and generalising the operations of [16].

One of the key ingredients in our result is a way of ‘partitioning’ an arbitrary ordinal tree into equal-sized (to within a constant factor) connected subtrees of polylogarithmic size each. Of course, this is not always possible – consider a tree comprising a root and

² Jacobson’s original results were stated as requiring $O(\lg n)$ probes in the *bit-probe* model.

$n - 1$ children – so we obtain a family of subtrees that *cover* the original tree and may intersect only at their common roots. For binary trees, one can actually partition the tree, and there is a standard transform that converts any ordinal tree into a binary tree with the same number of nodes (see e.g. [16]). Unfortunately this transformation does not preserve either distances or pre- and post- orders.

We also consider succinct representations of ordinal trees where each node in the tree is labelled with a symbol from an alphabet Σ . We would like to support “labelled” versions of each of the previously defined operations, for example, $\text{ANC}(x, \sigma, i)$ chooses the i -th ancestor of x that is labelled with σ , and $\text{SELECT}_{pre}(\sigma, i)$ chooses the i -th node in pre-order that is labelled with σ , for any $\sigma \in \Sigma$. The information-theoretic space lower bound for such trees is $n \lg |\Sigma| + 2n - O(\lg n)$ bits. We show how to modify the above representation so that using $n (\lg |\Sigma| + 2) + O(|\Sigma| n \lg \lg n / \lg \lg n)$ bits, it can support the labelled operations in $O(1)$ time as well. Thus, if $|\Sigma|$ is a constant the labelled operations are also handled optimally, and the space used is $n(\lg |\Sigma| + 2) + o(n(\lg |\Sigma| + 2))$, i.e., optimal to within lower-order terms, whenever $|\Sigma| = o(\lg \lg n)$.

This result is a first step towards the space-efficient representation of (large, static) XML documents. The correspondence between XML documents and ordinal trees is obvious and well understood (see Figure 1.2 for a simplified example). The XML document object model (DOM) [14] provides a framework to access XML documents using the query language XPath [10]. Unfortunately, an implementation of the DOM (using explicit storage of the tree with several pointers per node) can take up many times more memory space than the equivalent raw XML file (which is already verbose). This ‘XML bloat’ significantly impedes scalability of current XML query processors [2].

Our data structure gives an XML representation that supports the *axis specifiers* in XPath *location path expressions* (LPE). An XPath LPE refers to a node, or an ordered node set in the document tree. At any given point when evaluating an XPath LPE, we have an ordered set S of nodes matching some part of the LPE. The next step in the query could involve updating S to include specific nodes in the document that are located along one of several *axis specifiers* relative to any node $x \in S$ (x is called the *context node*). Some example axis specifiers are:

child	All children of the context node.
descendant	All descendants of the context node.
ancestor	All ancestors of the context node.
preceding	All nodes before the context node in pre-order, excluding any ancestors of the context node.

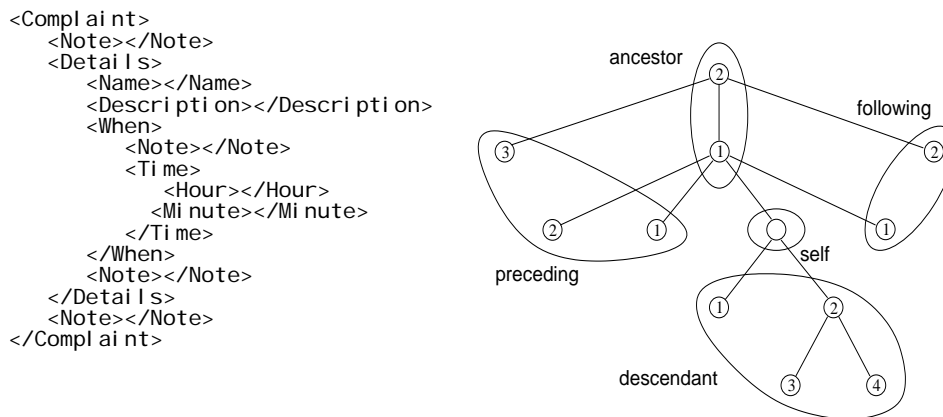


Figure 1.2 Small XML fragment (only tags shown) and corresponding tree representation. Five axis specifiers are shown with nodes numbered according to their position along the given axis. `self` denotes the current context node.

Most axis specifiers return nodes in pre-order, except ones such as `ancestor` and `preceding`, which return nodes in reverse pre-order. These axis specifiers may be followed by a label, which indicates the label of the node sought, and a number, which indicates the position of the node sought along the given axis (e.g., `child:a[10]` selects the 10th child of the context node that is labelled `a`).³ Using the labelled tree operations, we can select along each of the axis specifiers in $O(1)$ time. Evaluating XPath LPEs efficiently is central to XML query processing, but we are not aware of any other $O(1)$ -time implementations for these crucial subroutines.

The rest of this paper is organised as follows. Section 2 deals with preliminaries, including the tree-covering algorithm. Section 3 gives a $2n + o(n)$ -bit representation that supports essentially only ANC queries. Section 4 shows how to combine and extend many the ideas in Sections 2 and 3 to give the final result. Section 5 extends the results to labelled trees and fleshes out the connection to XML documents, and Section 6 concludes with some open problems.

2. PRELIMINARIES

2.1 A tree covering procedure.

We now show how, given an integer parameter, $M \geq 2$, and an ordinal tree, T , we can cover the vertices of T with a number of connected subtrees, all of which are of size

³ This is a simplified view of axis specifiers that ignores text, comment nodes etc.; see Section 5 for more.

$\Theta(M)$, except possibly one subtree containing the root of T , which could be of size $O(M)$. Additionally, two subtrees are either disjoint or intersect only at their common root. An example of the kind of cover we propose is given in Figure 2.1.

To achieve this, we consider each node $x \in T$ in turn in post-order and call the procedure $\text{GROUP}(x)$ described below. This procedure either creates a set of new *cover elements* (i.e. subtrees that will form part of the cover) rooted at x , or designates a set of currently uncovered nodes, including x , as a *partially completed subtree* (PCS). As will become apparent below, a PCS is a connected subgraph of T , which is too small to be designated as a cover element. Since we visit nodes in post-order, when we call $\text{GROUP}(x)$, all of x 's children have been visited; in general, some of x 's children will be roots of PCSs. The effect of $\text{GROUP}(x)$ is to coalesce these PCSs into larger connected subtrees that include x .

In what follows, all cover elements and PCSs are represented by sets of vertices, and the algorithm uses the notation $\bigcup S = \bigcup_{X \in S} X$, whenever S is a set of sets:

$\text{GROUP}(x)$:

1. If x is a leaf, $\{x\}$ becomes a PCS; return.
2. Otherwise, x has a number of children, each of which is either the root of a cover element or the root of a PCS. Denote by $Y = \{S_1, \dots, S_p\}$ the set of all "child" PCSs of x (numbered so that the root of S_i comes before the root of S_{i+1}).
 - a. If $|\bigcup Y| < M - 1$ then make $\{x\} \cup (\bigcup Y)$ into a PCS and return.
 - b. Otherwise, perform the following steps:
 - i. Create a new cover element $Z = \bigcup \{\{x\}, S_q, S_{q+1}, \dots, S_r\}$, where S_q is the leftmost PCS in Y , and r is the smallest index such that $|Z| \geq M - 1$.
 - ii. Set $Y = Y - \{S_q, S_{q+1}, \dots, S_r\}$.
 - iii. If $|\bigcup Y| < M - 1$, then set $Z = Z \cup (\bigcup Y)$, output Z as a cover element and return. Otherwise, output Z as a cover element and go to (i).

After the last step of the algorithm, there may be a PCS at the root of T , which is then made into a cover element as well (see Figure 2.1). Clearly, two cover elements are either disjoint or intersect only at their common root, and the sizes of cover elements are also bounded as claimed before:

LEMMA 2.1. Suppose the above procedure is run on an ordinal tree with some parameter $M \geq 2$. Then, for any cover element A in the generated cover, $|A| \leq 3M - 4$. Also, unless A contains the root of the tree, $|A| \geq M$.

Proof. At any step in the processing, a given node x will have $k \geq 0$ partially completed child subtrees, $\{S_1, S_2, \dots, S_k\}$. If $|S_1| + \dots + |S_k| \leq M-2$, then the algorithm will combine all the subtrees with x to form another PCS of size $\leq M-1$ and return; otherwise it will create a number of cover elements.

If $|S_1| + \dots + |S_k| \geq M-1$, the algorithm will select the first p PCSs such that $|S_1| + \dots + |S_p| \geq M-1$. These PCSs will be combined with x to form a cover element A of size $\geq M$. By definition, $|S_1| + \dots + |S_{p-1}| < M-1$, and $|S_p| \leq M-1$, so $|S_1| + \dots + |S_p| \leq 2M-3$. Including x , $|A| \leq 2M-2$, which is the maximum size of a cover element that is generated when the test at the start of step b(iii) fails.

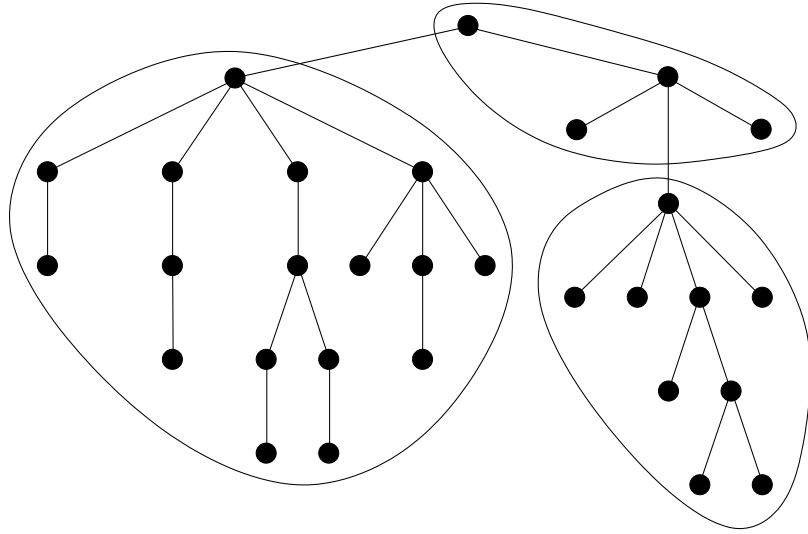


Figure 2.1 Example tree subdivided using our tree cover algorithm with parameter $M = 7$.

This process is continued for all the PCSs, $\{|S_{p+1}|, \dots, |S_k|\}$. If at some point the size of the remaining PCSs is insufficient to produce another cover element with size $\geq M$ (i.e. the test of b(iii) succeeds) then up to $M-2$ elements may be added to the last cover element created, giving a cover element of size at most $2M-2 + M-2 = 3M-4$. \square

2.2 Tree cover decomposition

We cover the given tree T using the above procedure, choosing $M = \max(\lceil (\lg n)^4 \rceil, 2)$. We call the resulting cover a *tier 1* cover, and use the term *mini-trees* to denote the elements of this cover. We then apply the above procedure to each mini-tree with parameter $M' = \max(\lceil (\lg n) / 24 \rceil, 2)$, obtaining a *tier 2* cover formed of *micro-trees*.

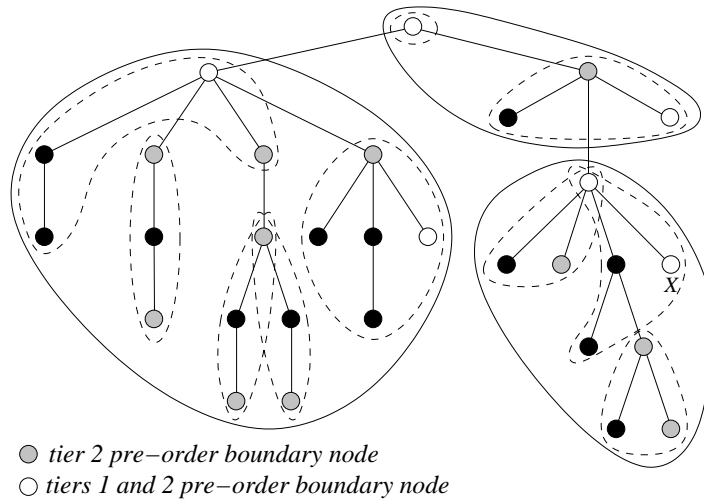


Figure 2.2 Two level tree cover decomposition with $M = 7$ and $M' = 3$. Solid curves enclose mini-trees while dashed curves enclose micro-trees.

For $i = 1, 2$, a tier i pre-order *boundary* node is either the first node in pre-order (i.e., the root) of a tier i subtree or the last node in pre-order of a tier i subtree. We define tier i post-order boundary nodes analogously. Since any micro-tree is entirely contained within a mini-tree, and all mini-trees are covered by micro-trees, it follows that all tier 1 pre- and post-order boundary nodes are also tier 2 pre- and post-order boundary nodes, respectively (see Figure 2.2); in particular a mini-tree root is also a micro-tree root.

Let $pre(x)$ denote the depth-first pre-order position of node x in a tree T . When A denotes a mini- (or micro-) tree, let $root(A)$ denote the root of the tree and let $last(A)$ and $second(A)$ denote nodes such that, for all $x \in A$, if $x \neq root(A)$, then $pre(second(A)) \leq pre(x) \leq pre(last(A))$. Given two cover elements A and B within the same tier, say that $A \prec B$ if either $pre(root(A)) < pre(root(B))$ or $root(A) = root(B)$ and $pre(second(A)) < pre(second(B))$.

We now prove the following property of the tree cover:

LEMMA 2.2. Let σ be a sequence of nodes that appear consecutively in a pre-order traversal of the given tree T , such that σ contains no tier i pre-order boundary nodes, for $i \in \{1, 2\}$. Then all nodes in σ appear consecutively in a pre-order traversal of a single tier i subtree. The above also holds if we replace ‘pre-order’ by ‘post-order’.

We begin by proving the following proposition:

PROPOSITION 2.3. Given any two cover elements A and B in a tree T , such that $A \prec B$, one of the two cases below holds:

- if $root(A) \neq root(B)$, then for all $x \in T$ such that $pre(root(B)) \leq pre(x) \leq pre(last(B))$, x is not in A .
- if $root(A) = root(B)$, then $pre(last(A)) \leq pre(second(B))$.

Proof. First assume $root(A) \neq root(B)$, so A and B must be disjoint. Suppose that $root(B)$ is a descendant of $root(A)$ (Case 1, Figure 2.3). Let S_B denote the subtree rooted at $root(B)$. By definition, the sequence of $|S_B|$ nodes in the pre-order listing of T that starts at $root(B)$ is a complete list of nodes that are in S_B . Since A is connected, S_B contains no nodes in A , but every node in B is contained within S_B , so (i) holds in this case. If $root(B)$ is not a descendant of $root(A)$, then (Case 2, Figure 2.3) every node in A is visited before the first node in B is visited, hence the sequence of nodes between $root(B)$ and $last(B)$ does not contain any node in A , so (i) holds in this case as well.

Now suppose that $z = root(A) = root(B)$ (Case 3, Figure 2.3). Let a be the rightmost child of z that belongs to A , and let b be the leftmost child of z that belongs to B . By definition of the cover algorithm, $pre(a) < pre(b)$. If S_a and S_b are the subtrees rooted at a and b respectively, we know that all nodes in S_a (including $last(A)$) come before all nodes in S_b (including $second(B)$) in the pre-order. \square

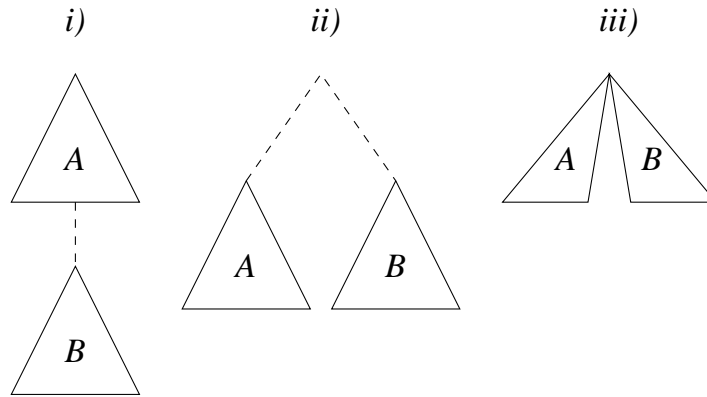


Figure 2.3 Proposition 2.3 (i) case 1; (ii) case 2; (iii) case 3.

Proof (of Lemma 2.2). It is enough to show that given two consecutive nodes, x followed by y , in a depth-first pre-ordering of a tree T , that belong to different cover elements, P and Q respectively, one of the nodes, x or y , must be a boundary node. The proof for post-order boundary nodes is similar.

Since roots of cover elements are boundary nodes, assume that neither x nor y is the root of the cover element to which it belongs. We consider three cases: (a) $pre(root(P)) > pre(root(Q))$ (b) $pre(root(P)) < pre(root(Q))$, or (c) $root(P) = root(Q)$:

- (a) By Proposition 2.3, the sequence of nodes between $root(P)$ and $last(P)$ does not contain any nodes that are in Q ; thus we must have $x = last(P)$, so x is a boundary node.
- (b) By Proposition 2.3, the sequence of nodes between $root(Q)$ and $last(Q)$ cannot contain any nodes that are in P . We already know that $y \neq root(Q)$, so x is in between two nodes in Q ; this is a contradiction, so case (b) cannot occur.
- (c) By Proposition 2.3, either all non-root nodes in P must precede all non-root nodes in Q , or vice-versa. As x precedes y , the second alternative is impossible. Thus, $x = last(P)$, and so x is a boundary node. \square

2.3 Node names.

From an external viewpoint, we assume the name of a node in the given tree T is just its pre-order number (this also provides the promised mapping from the nodes of T to $\{1, \dots, n\}$). Suppose that we have determined tier 1 and 2 covers of T as above. Then, we number the mini-trees t_1, t_2, \dots in a way that ensures that $t_1 \prec t_2 \prec \dots$. For each mini-tree t_i , we number the micro-trees within it as $\mu_{\langle i,1 \rangle}, \mu_{\langle i,2 \rangle}, \dots$ in the same way, and we also number each node in a micro-tree consecutively in pre-order starting from 1. This allows us to refer to a node x (for internal purposes) by an alternate name $\tau(x) = \langle \tau_1(x), \tau_2(x), \tau_3(x) \rangle$, meaning that x is the $\tau_3(x)$ -th node in pre-order in micro-tree number $\mu_{\langle \tau_1(x), \tau_2(x) \rangle}$. For example, $\langle 3, 2, 4 \rangle$ would correspond to the node marked X in Figure 2.2. Since mini- and micro- trees are not disjoint, a root of a micro- or mini- tree may have more than one τ -name; we will refer to the lexicographically smallest τ -name for a node as its *canonical* name, and the copy of a node with the canonical name as a *canonical* copy. Note that by Lemma 2.1, there are at most $t_1 = \lfloor n / M \rfloor + 1$ mini-trees, $t_2 = \lfloor 3M / M' \rfloor + 1$ micro-trees within a mini-tree and $t_3 = 3M'$ nodes in a micro-tree, so a τ -name can be viewed as a $\lceil \lg t_1 \rceil + \lceil \lg t_2 \rceil + \lceil \lg t_3 \rceil = \lg n + O(1)$ -bit integer.

2.4 Previous results used

A key data structure which we use in the paper is a *bitvector*, which represents a sequence of n bits $b_1 b_2 \dots b_n$. A bitvector supports the operations $RANK0(i)$ and $RANK1(i)$, which report the number of 0 bits and 1 bits respectively in $b_1 \dots b_i$, as well as supporting the operations $SELECT0(i)$ and $SELECT1(i)$, which report the indices of the i -th 0 bit and 1 bit respectively. Theorem 2.1 below is from [18]:

THEOREM 2.1. A bitvector of length n can support $RANK0(i)$, $RANK1(i)$, $SELECT0(i)$ and $SELECT1(i)$ in $O(1)$ time, using $\lg \binom{n}{m} + O(n \lg \lg n / \lg n)$ bits, where m is the number of bits that have value 1.

We remark that since $\lg \binom{n}{m} \leq n$, the space used by the data structure is never more than $n + o(n) = O(n)$ bits. A space bound of $n + o(n)$ bits was previously obtained by [9,15] using a simpler data structure. We also use the following results. Theorem 2.2 describes the DFUDS (depth first unary degree sequence) representation; Theorem 2.3 gives simple, rather than the strongest, results from [1, 3].

THEOREM 2.2. [4, 5] There is a $2n + o(n)$ bit representation of an n -node ordinal tree that provides a mapping from the nodes of the tree to $\{1, \dots, n\}$ and supports DEG, CHILD, CHILDANK, DESC, and finding the parent of a node, in constant time.

THEOREM 2.3. [1, 3] There is an $O(n \lg^2 n)$ bit representation of an n -node ordinal tree in which ANC queries can be answered in constant time. The nodes of the tree can be named using arbitrary distinct integers of $\lg n + O(1)$ bits each.

We will frequently represent micro-trees and other suitably small objects using *implicit* representations. An implicit representation of an object o would contain some short “header” information, followed by an integer $r(o)$ that represents the object in (close to) the information-theoretic minimum number of bits. For example, if o were an ordinal tree with i nodes, its implicit representation might comprise of a header that contains the integer i , and an integer $r(o)$, which could be a sequence of $2i$ bits that encode the tree in a parenthesis encoding. To compute $f(o, x_1, \dots, x_k)$ for some function f , we index a pre-computed table that contains the value of f for all possible values of its arguments, using an index formed out of $r(o), x_1, \dots, x_k$. This is the “four Russians” trick, and is used in many contexts within this paper and other papers dealing with succinct data structures. In all cases of interest to us, the lookup table takes $o(n)$ bits of space.

3. A $2n + o(n)$ -BIT REPRESENTATION SUPPORTING ANCESTOR AND DEPTH QUERIES

We begin our description with a structure that uses $2n + o(n)$ bits to represent an n -node tree and supports ANC(x, i) queries. We assume for now that x is specified by its τ -name, and that we return the τ -name of ANC(x, i).

We first argue that the tier 1 and 2 covers described in Section 2.2 can be represented using $2n + o(n)$ bits. We represent each micro-tree μ , $|\mu| = i$, that comprises the number i in a fixed-size field of $\lceil \lg \lg n \rceil$ bits, and a further $2i$ bits that specify the structure of μ using (say) the parenthesis representation. Since micro-trees can only have root nodes in common, the total size of all $O(n / \lg n)$ micro-trees is $n + O(n / \lg n)$, and the implicit representations of these micro-trees take up $2n + O(n \lg \lg n / \lg n)$ bits in all.

Each mini-tree has at most $3M$ nodes (Lemma 2.1), or $3M/M'$ nodes after the compression of each micro-tree by its index. Hence each mini-tree can be represented (using the standard pointer representation) using $O((M \lg M)/M')$ bits which is $O((\lg n)^3 \lg \lg n)$ bits for each mini-tree for a total of $O((n \lg \lg n)/\lg n)$ bits. The (at most n/M) pointers connecting mini-trees will take even less space.

In addition to the tier 1 and 2 covers we will have a *macro tree* consisting of *macro nodes*. As in [1], a node in the given tree T is called a macro node if it is either the root of T (denoted $root(T)$) or its depth is a multiple of M and it has at least M descendants. We obtain the macro tree, T' , from T by deleting all but the macro nodes, and adding edges from a macro node x to its nearest macro node ancestor in T ; the macro tree T' consists of $O(n/M)$ nodes.

We will represent the macro tree T' using Theorem 2.3; this takes $O((n (\lg n)^2)/M) = O(n/\lg n)$ bits and supports ANC queries in $O(1)$ time. Essentially the macro tree helps us to jump in $O(1)$ time to within M of the required ancestor. To find ancestors at a distance of at most M , we will keep some auxiliary pointers with the root of every mini tree, which we call *skip pointers*.

A (Δ, l) skip pointer list at a node x which is at a depth d from the root, is a 2-dimensional array A , where $A[i, j]$, for $0 \leq i \leq l-1$ and $1 \leq j \leq \Delta$ contains the τ -name of the $\min\{d, j\Delta^i\}$ -th ancestor of x . Hence, the $A[0, *]$ entries contain ancestors at distances $1, 2, \dots, \Delta$ from x , the $A[1, *]$ entries contain ancestors at distances $\Delta, 2\Delta, \dots, \Delta^2$ from x , and so on. With the root of every mini-tree t , we keep

- its depth, denoted by $d(t)$,
- the τ -name of its closest macro node ancestor $m(t)$, along with its distance from $root(t)$, and
- a $(\lceil \sqrt{M} \rceil, 2)$ skip pointer list, s_t .

These take up $O(\sqrt{M})$ values each of length $O(\lg n)$ bits with every mini-tree root for a total of $O(n \lg n / \sqrt{M}) = o(n)$ bits. Similarly, with the root of every micro-tree μ within t , we keep

- its distance from the root of the mini-tree it is in, denoted $d(\mu)$ and
- a $(\lceil \sqrt{\lg n} \rceil, 8)$ skip pointer list s_μ , essentially as above, but with no pointer pointing beyond $root(t)$.

Note that $d(\mu) = O(M) = O((\lg n)^4)$ and hence takes $O(\lg \lg n)$ bits. Since all the skip pointers point to nodes in the same mini-tree, their τ_1 -names are the same as that of the

root, and we don't need to store their τ_1 -names. Hence each of these names takes only $O(\lg \lg n)$ bits for a total of $O(\sqrt{\lg n} \lg \lg n)$ bits for each micro-tree, or a total of $O(n \lg \lg n / \sqrt{\lg n})$ bits for all micro-trees.

To support ancestor queries whose answers lie within a micro-tree, we use table look-up. The table has, for every possible micro tree of up to $3M'$ nodes, and for every node x in such a micro-tree, two items:

- its distance $d(x)$ from the micro-tree root,⁴ and
- for every distance i up to the height of the micro-tree, the τ_3 value of $\text{ANC}(x, i)$ (the τ_1 and the τ_2 values are the same as that of x).

The table is indexed using the implicit representation of the micro-tree, together with $\tau_3(x)$ and the argument i to the ANC query. Each entry in this table is clearly $O(\lg n)$ bits, and the number of entries is at most $\sum_{i=1}^{3M'} i^2 \cdot 2^{2i} \leq 18 (M')^2 \cdot 2^{6M'}$, so the table has size $O(n^{3/4} (\lg n)^3) = o(n)$ bits. We now argue that this data structure supports ANC and DEPTH queries in $O(1)$ time:

THEOREM 3.1. There exists a data structure to represent an n -node tree taking $2n + O(n \lg \lg n / \sqrt{\lg n})$ bits that supports ANC and DEPTH queries in constant time.

Proof. The description of the data structure is as above. We first focus on (the straightforward) task of computing $\text{DEPTH}(x)$ for a node x in a micro-tree μ within a mini-tree t . If $x = \text{root}(t)$ then $d(t)$ is the answer. Otherwise if $x = \text{root}(\mu)$ then $\text{DEPTH}(x) = d(t) + d(\mu)$. Otherwise, we obtain $d(x)$, the distance of x from $\text{root}(\mu)$ from the table, and $\text{DEPTH}(x) = d(t) + d(\mu) + d(x)$. Note that the τ -name of a node indicates whether or not it is a micro- or mini-tree root (see Figure 3.1).

The algorithm to compute $\text{ANC}(x, i)$ is given in Figure 3.1. The correctness of the algorithm is immediate. Now we argue that it runs in $O(1)$ time. In Case 1, the algorithm either stops or a recursive call takes us to the root of a micro-tree (Case 2).

In Case 2, the algorithm stops or a recursive call takes us to the root of a mini-tree (case 3) or we move upwards in the mini-tree by a factor of $\sqrt{\lg n}$ each time (since $s < k \leq 8$). So in at most 8 steps (each of which may involve going through a step in case 1), we get the answer or move to the mini-tree root.

⁴ Note that we use the same symbol d for the depth of a node from its micro-tree root, the (relative) depth information we keep at micro-tree roots, and the (true) depth stored at mini-tree roots.

```

Begin
  case 1:   $x$  is not a root of a micro-tree (i.e.,  $\tau_3(x) > 1$ ).
            $d \leftarrow d(\tau_3(x))$  (obtained from a table look-up).
           If  $i \leq d$  find  $a = \text{ANC}(x, i)$  from the table representation of its micro-tree
            $\tau_3(x)$  and return  $\langle \tau_1(x), \tau_2(x), a \rangle$ .
           Else ( $i > d$ ) set  $i \leftarrow i - d$ ,  $x \leftarrow \langle \tau_1(x), \tau_2(x), 1 \rangle$  (the root of its micro-tree);
           return (recursively)  $\text{ANC}(x, i)$ .

  case 2:   $x$  is the root of a micro-tree  $\mu = \tau_2(x)$ , but not a root of a mini-tree (i.e.,
            $\tau_3(x) = 1, \tau_2(x) > 1$ ).
           Set  $\delta \leftarrow \lceil \sqrt{\lg n} \rceil$  and  $d \leftarrow d(\mu)$  (Note:  $d \leq \delta^8$ ).
           If  $i > d$ , then  $x \leftarrow \langle \tau_1(x), 1, 1 \rangle$  and  $i \leftarrow i - d$  and return (recursively)
            $\text{ANC}(x, i)$ .
           Else ( $i \leq d$ ) (answer lies within the mini-tree, and  $\delta^k \geq d \geq i$  for some
           integer  $k \leq 8$ ):
           find  $r, s$  for some  $1 \leq r \leq \delta$  and  $0 \leq s < k \leq 8$  such that  $r\delta^s \leq i < (r+1)\delta^s$ .
           Let  $x' \leftarrow \langle \tau_1(x), s_\mu[s, r] \rangle$ .
           Then  $x'$  is an ancestor of  $x$  and  $\text{distance}(x, x') \leq i < \text{distance}(x, x') + \delta^s$ .
           If  $s=0$ , then return  $x'$  else  $x \leftarrow x'$  and  $i \leftarrow i - r\delta^s$  and return  $\text{ANC}(x, i)$ .

  case 3:   $x$  is a root of the mini-tree  $t = \tau_1(x)$  (i.e.,  $\tau_2(x) = \tau_3(x) = 1$ ).
            $\delta \leftarrow \lceil \sqrt{M} \rceil$ .
           If  $i \leq M$ , then find  $r, s$  such that  $1 \leq r \leq \delta$  and  $s \in \{0, 1\}$  such that
            $r\delta^s \leq i < (r+1)\delta^s$ . Let  $x' \leftarrow s_t[s, r]$ .
           Then  $x'$  is an ancestor of  $x$  and  $\text{distance}(x, x') \leq i < \text{distance}(x, x') + \delta^s$ .
           If  $s = 0$ , then return  $x'$  else  $x \leftarrow x'$  and  $i \leftarrow i - r\delta^s$  and return  $\text{ANC}(x, i)$ .
           Else ( $i > M$ ), find the closest macro node ancestor  $m(t)$  and its distance
            $d'$  from the root of  $t$ .
           Let  $x' = \text{ANC}(m(t), \lfloor (i - d') / M \rfloor)$  in the macro tree  $T'$ .
           Return  $\text{ANC}(x', (i - d') \bmod M)$ .

end

```

Figure 3.1 Code for $\text{ANC}(x, i)$. The argument x is given as its τ -name $\langle \tau_1(x), \tau_2(x), \tau_3(x) \rangle$; the output is the τ -name of the i -th ancestor of x .

In Case 3, first, by using the level-ancestor structure in the macro-tree, if necessary, we reach an ancestor node whose distance to the answer is at most M . Then, using the skip pointers, we reach a node whose distance to the answer is at most \sqrt{M} . Then perhaps after an execution of a step in each of case 1 and case 2, we get the answer from

the skip pointers at the first level in the mini-tree root. So overall the running time of the algorithm is a constant. \square

4. A COMPLETE $2n + o(n)$ -BIT REPRESENTATION

This section has three parts. Subsection 4.1 describes a modified tree cover, addressing the difficulty that, while the representation described in Section 3 is adequate for ANC queries, it does not support the remaining queries. CHILD queries, for example, seem difficult to support in $O(1)$ time using the representation of Section 3 since the children of a node may be spread over several micro- or mini-trees. Subsection 4.2 describes how the modified tree cover helps to support all operations bar RANK, SELECT and DESC in $O(1)$ time, provided that nodes are referred to by τ -names. Subsection 4.3 shows how to convert between τ -names and pre-order/post-order numbers, thus allowing us to support RANK, SELECT and DESC as well, and proving our main result.

4.1 A modified tree cover

We start with the cover created in Section 2. Nodes in the original cover are called *original* nodes. In addition we will *extend* each micro-tree to possibly include a number of *promoted* nodes, which are copies of original nodes in other micro-trees. Nodes are promoted as follows:

- For every node x that is the root of a micro-tree, we promote x into the extended micro-tree to which the parent of x , $p(x)$, belongs. If $p(x)$ belongs to more than one micro-tree then we promote x into one particular extended micro-tree as follows ($p(x)$ is the root of each of these micro-trees):
 - let y be the rightmost sibling to the left of x that is not the root of a micro-tree. We promote x to the extended micro-tree that y is in.
 - if no such y exists, then we promote x to the left-most extended micro-tree of which $p(x)$ is the root.

PROPOSITION 4.1. Except for the roots of micro-trees, all other nodes have the property that (at least a promoted copy of) each of their children is in the same extended micro-tree as themselves.

Proof. If the child x of a node y is not in the same micro-tree as y , then x must be the root of a micro-tree. By the above, x is promoted to at least one of the micro-trees containing y . If y is not the root of a micro-tree, then y belongs to only one micro-tree, and all its children are promoted to the same micro-tree as y . \square

The τ -name of a vertex x remains the same as before the promotion; i.e., we ignore promoted nodes when counting $\tau_3(x)$. We now discuss the representation of extended

micro-trees (which can be very large!). We say that an extended micro-tree is of *type 1* if its size is $\leq \frac{1}{4} \lg n$ and *type 2* otherwise. We first note that:

PROPOSITION 4.2. The number of type 2 extended micro trees is $O(n / (\lg n)^2)$ and the number of original nodes in all type 2 extended micro trees put together is $O(n / \lg n)$.

Proof. Each type 2 extended micro tree has at most $3M' - 4$ original nodes, and so at least $\frac{1}{4} \lg n - 3M' + 4$ promoted nodes. Since $M' = \max(\lceil (\lg n) / 24 \rceil, 2)$, each type 2 extended micro-tree has $\Omega(\lg n)$ promoted nodes. There are $O(n / \lg n)$ promoted nodes in all, so the first part follows. Now the second part is immediate. \square

4.1.1 *The base representation.* We now describe the representation of the micro-trees. We will sometimes talk about bitvectors that indicate whether a particular node in a tree satisfies a particular property. In this case, we will by default assume that the i -th element of the bitvector refers to the i -th node of the tree in pre-order, unless specified otherwise. Bitvectors are represented according to Theorem 2.1; if the strong space bound of the Theorem is indeed necessary, we say that a *compressed* representation is used, in order to make this explicit. Otherwise, we will merely use the linear upper bound given in the remark following Theorem 2.1, to simplify calculations.

We first describe the basic data structures associated with each extended micro-tree, and introduce others as and when needed.

4.1.2 *Type 1 extended micro-trees.* The principal data we store for a type 1 extended micro-tree μ_i (comprising p_i promoted nodes and o_i original nodes) are:

1. $O(\lg \lg n)$ -bit header information needed to parse the following information, including the type of the micro-tree; numbers p_i and o_i ; start/end positions of various sub-parts.
2. A bitstring (called *tree_i*) containing a $2(p_i + o_i)$ -bit implicit representation of μ_i .
3. A bistring (called *nodetypes_i*) that contains an implicit representation (using $\lceil \lg \binom{p_i + o_i}{p_i} \rceil$ bits) of the subset of all nodes in μ_i that are promoted.
4. A representation (*edges_i*) of all edges that leave μ_i :
 - a. A string of p_i bits which specifies whether a promoted node is in the same mini-tree as μ_i ;
 - b. Two arrays that contain τ_2 -names and τ_1 -names, respectively, of the original copy of the promoted nodes of μ_i , depending on whether the original copy of the promoted node is in the same mini-tree as μ_i . The sizes of the arrays are p'_i and $(p_i - p'_i)$ respectively, where p'_i is the number of promoted nodes that belong to the same mini-tree as μ_i .

4.1.3 *Type 2 extended micro-trees.* The principal data we store for a type 2 extended micro-tree i (comprising p_i promoted nodes and o_i original nodes) are:

5. $O(\lg n)$ -bit header information needed to parse the following information, including the type of the extended micro-tree; numbers p_i and o_i ; start/end positions of various sub-parts etc.
6. A representation ($tree_i$) of the tree structure consisting of:
 - a. An $O(p_i + o_i)$ -bit DFUDS representation of the extended micro-tree (Theorem 2.2);
 - b. A $2o_i$ -bit implicit representation of the micro-tree (excluding promoted nodes).
7. A bitvector ($nodetypes_i$) of length $p_i + o_i$ that specifies if a node in μ_i is original (denoted 1) or promoted (denoted 0).
8. A representation ($edges_i$) of all edges that leave μ_i , as in (4) above.

4.1.4 *Roots of extended micro-trees.* Let μ_i be an extended micro-tree and let $r_i = \text{root}(\mu_i)$. Suppose that r_i has d_i children, which may be in one or more mini- or micro-trees. We store the following data, associated only with the canonical copy of r_i :

9. Two bitvectors ($child_i$) — one where the first bit is 1, and for $j = 2, \dots, d_i$, the j -th bit is 1 if the j -th child of r_i is in a different mini-tree from the $(j - 1)$ -st child; the other is analogous but for micro-trees.
10. Like (4, 8) τ_1 or τ_2 names of the other micro- and mini-trees of which r_i is the root ($edges_i$).

We now calculate the space used so far. Since there are $O(n / \lg n)$ and $O(n / (\lg n)^2)$ extended micro-trees of Types 1 and 2 respectively, the space used by items (1) and (5) is $o(n)$ bits. We use the following facts extensively:

(a) For nonnegative integers a_i, b_i , $\prod_i \binom{a_i}{b_i} \leq \binom{\sum_i a_i}{\sum_i b_i}$;

(b) $\lg \binom{O(n)}{O(n/\lg n)} = O(n \lg \lg n / \lg n)$;

(c) Let $k = O(n / \lg n)$. If $x_1, \dots, x_k \geq 2$ and $\sum_i x_i = O(n)$, then $\sum_i x_i \lg \lg x_i / \lg x_i = O(n \lg \lg \lg n / \lg \lg n)$.

The last one of these follows from the concavity of the function $f(x) = x \lg \lg x / \lg x$. Letting $s = \sum_i x_i$, by Jensen's inequality [19], the sum $\sum_i x_i \lg \lg x_i / \lg x_i$ is maximised when all the x_i 's are equal. Hence, $\sum_i x_i \lg \lg x_i / \lg x_i \leq s \lg \lg (s / k) / \lg (s / k)$. It is easy to check that $s \lg \lg (s / k) / \lg (s / k)$ is maximised when both s and k are as large as they can be, i.e., when $s = \Theta(n)$ and $k = \Theta(n / \lg n)$, thus giving (c).

Let S_j with $j \in \{1, 2\}$ consist of the indices i such that each μ_i is an extended micro-tree of type j . We note the following ((iii) restates Proposition 4.2):

- (i) $\sum_{i \in S_1} p_i$ and $\sum_{i \in S_2} p_i$ are both $O(n / \lg n)$;
- (ii) $\sum_{i \in S_1} o_i = n + O(n / \lg n)$;
- (iii) $\sum_{i \in S_2} o_i = O(n / \lg n)$.

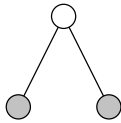
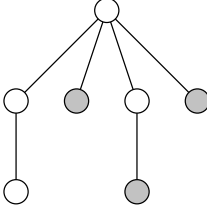
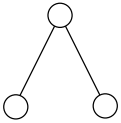
micro-tree τ -name	$\langle 1, 1 \rangle$	$\langle 2, 1 \rangle$	$\langle 3, 1 \rangle$
tree structure			
type (1, 5)	1	2	1
p_i (1, 5)	2	3	0
o_i (1, 5)	1	4	3
$tree_i$ (2, 6.a)	((()))	((((((((()))))))))	(((())))
(6.b)	n/a	(((())))	n/a
$nodetypes_i$ (3, 7)	0 0	1 1 1 0 1 0 0	Λ
$edges_i$ (4.a, 8.a)	0 1	1 1 1	Λ
(τ_2) (4.b, 8.b)	[2]	[2, 3, 5]	[]
(τ_1) (4.b, 8.b)	[2]	[]	[]
d_i (9)	2	4	4
$child_i$ (mini) (9)	1 0	1 0 0 0	1 0 0 0
(micro) (9)	1 0	1 0 0 0	1 0 1 0
$edges_i$ (τ_1) (10)	[]	[]	[]
(τ_2) (10)	[]	[]	[2]

Figure 4.1 Example representation of three selected extended micro-trees from the tree in Figure 2.2. White denotes original nodes while grey denotes promoted nodes. For illustrative purposes we assume that type 1 extended micro-trees have at most 6 nodes. n/a denotes data that is not stored (due to tree type) and λ denotes an empty bitstring. Numbers in parentheses in the left-hand column refer to the data description above.

From these facts, we obtain that the number of bits used by $tree_i$ summed over all type 1 extended micro-trees is:

$$\sum_{i \in S_1} 2(p_i + o_i) \leq 2n + O(n / \lg n),$$

and summed over all type 2 extended micro-trees is

$$\sum_{i \in S_2} O(p_i + o_i) = O(n / \lg n).$$

The space used by $nodetypes_i$ (for type 1 extended micro-trees) is

$$\sum_{i \in S_1} \left[\lg \binom{p_i + o_i}{p_i} \right] \leq \lg \left(\sum_{i \in S_1} p_i + o_i \right) + |S_1| \leq \lg \binom{O(n)}{O(n/\lg n)} + O(n/\lg n) = O(n \lg \lg n / \lg n);$$

for type 2 extended micro-trees, it is $O(n / \lg n)$ as before. Thus, the space used so far is $2n + O(n \lg \lg n / \lg n)$ bits.

We now add up the space required by the data structures $child_i$ and $edges_i$. In (4) and (8) there are bitvectors of size $O(p_i)$; these add up to $O(n / \lg n)$ bits as before. In (9) the bitvectors are stored in compressed format; for the i -th root the bitvector for micro-trees takes $\lceil \lg \binom{d_i}{m_i} \rceil + O(d_i \lg \lg d_i / \lg d_i)$ bits by Theorem 2.1, where m_i is the number of micro-trees in which the children of r_i lie, and d_i is the degree of r_i . Since $\sum_i m_i = O(n / \lg n)$, as before, the first term adds up to $O(n \lg \lg n / \lg n)$ bits. For the second term, we first note that there are at most $O(n / \lg n)$ micro-trees, and hence only $O(n / \lg n)$ micro-trees whose roots have degree ≥ 2 . Applying fact (c) above, the second term sums up to $O(n \lg \lg \lg n / \lg \lg n)$ bits over micro-tree roots with degree ≥ 2 ; for micro-trees with degree 1 this term anyway sums up to $O(n / \lg n)$ bits. A similar argument is used for the mini-tree bitvector.

Next, note that each of the full $O(\lg n)$ -bit τ -names stored in (4, 8, 10) represents an edge between two mini-trees and each such edge is represented at most twice (once each with a root of an extended micro-tree and the extended micro-tree itself). Thus, storing these full τ -names takes $O((n / M) \lg n) = O(n / \lg^3 n) = o(n)$ bits in all. Similarly, storing the τ_2 names corresponding to edges between micro-trees takes $O(n \lg \lg n / \lg n) = o(n)$ bits overall as well.

4.2 τ -name operations.

We now describe how to compute ANC, DEPTH, CHILD, CHILDRANK and DEG in $O(1)$ time, but refer to nodes using their τ -names; these names take $\lg n + O(1)$ bits.

4.2.1 ANC and DEPTH operations. We augment the structure above (which takes $2n + o(n)$ bits) with the auxiliary data structures (macro tree, skip pointer lists, depth information, and tables) for ANC queries from the previous section, which take $o(n)$ bits as well. In order to support DEPTH and ANC queries in $O(1)$ time, it suffices to support them in $O(1)$ time within an extended micro-tree. Both of these are readily accomplished using table lookup — either on the bitvector that is the concatenation of $tree_i$ and $nodetypes_i$ (which has size at most $\frac{3}{4} \lg n$) for a type 1 extended micro-tree i , or on the implicit representation of the original nodes in $tree_i$ for a type 2 extended micro-tree i (recall that promoted nodes are not counted for the purpose of τ_3 names).

4.2.2 *CHILD operations.* We now discuss how to compute $\text{CHILD}(c, i)$ for a node c with τ -name $\langle x, y, z \rangle$. If $z \neq 1$ we proceed as follows, making use of Proposition 4.1. First the i -th child c' of c is found. If c' is not a promoted node, it is returned, but if it is, we find the rank of c' among the other promoted nodes. For type 1 nodes, both these steps take $O(1)$ time using table lookup. For type 2 nodes, c' is found using the DFUDS representation, and the type (and rank) of c' is found using the bitvector nodetypes_x ; again both steps take $O(1)$ time. Using RANK and SELECT on the bitvector in edges_i we can then easily access the τ -name of c' from the two arrays of τ -names in $O(1)$ time.

If $c = \text{root}(y)$, we use child_y to determine the micro-tree y' in which the i -th child lies (this is done by a RANK query on each of the bitvectors in child_y and looking up edges_y) and the number i' of children of c that are in micro-trees before y' (a SELECT operation on the bitvectors in child_y). This takes $O(1)$ time, after which we look for the $(i - i')$ -th child of the $\text{root}(y')$ (which is of course a copy of c) which can be found from the DFUDS representation or by a table look-up.

4.2.3 *CHILDRANK operations.* Next, we now describe how to compute $\text{CHILDRANK}(c)$ for a node c with τ -name $\langle x, y, z \rangle$. If $z \neq 1$ and $\text{parent}(c) \neq \text{root}(\mu_{\langle x, y \rangle})$ we first find the parent p of c . Using Proposition 4.1, we know a copy of each of the children of p is in y , which enables us to compute CHILDRANK as follows. For type 1 nodes, both these steps take $O(1)$ time using table lookup. For type 2 nodes, we look at the DFUDS representation of the extended micro-tree (the mapping between τ -names and the nodes in the extended micro-tree is handled as for the CHILD operation) and from this we can determine the parent of c , and, using the DFUDS representation's functionality, $\text{CHILDRANK}(c)$ as well; again, both steps take $O(1)$ time.

If $\text{parent}(c) = \text{root}(y)$ the siblings of c are not necessarily in the same extended micro tree y (the micro-tree y may overlap with other mini-/micro-trees at the root). We must store the following additional data, which is of size $o(n)$:

- a. with each mini-tree, t , we store crank_t , an $O(\lg n)$ bit integer representing the number of siblings to the left of $\text{second}(t)$;
- b. with each micro-tree, μ , we store:
 - i. crank_μ , an $O(\lg \lg n)$ bit integer representing the number of siblings to the left of $\text{second}(\mu)$ within the current mini-tree; and
 - ii. isroot_μ , a single bit (true/false) indicating if the root of μ is also the root of a mini-tree.

We can now calculate CHILDRANK as follows: first, we compute, as above, $\text{cr}(c)$, the rank of c among the set of c 's siblings that lie within the same micro-tree as c . Now, if

if $isroot_\mu = \text{true}$, then $\text{CHILDRANK}(c) = \text{crank}_t + \text{crank}_\mu + \text{cr}(c)$, otherwise $\text{CHILDRANK}(c) = \text{crank}_\mu + \text{cr}(c)$.

Finally, if $c = \text{root}(y)$ then there is a promoted copy of c in another micro-tree. We just need to navigate to that micro-tree (by performing an $\text{ANC}(c, 1)$ query) and then do a CHILDRANK operation on the promoted copy of c within that micro-tree, using one of the two methods above.

4.2.4 DEG operations. It is easy to compute the degree, $\text{DEG}(c)$, of a given node, c , with τ -name $\langle x, y, z \rangle$. If $c \neq \text{root}(y)$ then from Proposition 4.1, all children of c are in y . If c is in a type 1 micro-tree, then we can compute $\text{DEG}(c)$ using table lookup, otherwise we can determine it from the DFUDS representation of the type 2 micro-tree. Each of these takes $O(1)$ time.

If $c = \text{root}(y)$, (note that $\langle x, y, z \rangle$ is the canonical τ -name of c), then the degree, d_i , of c is stored explicitly at the root of the micro-tree (Section 4.1.4).

We have now shown the following:

THEOREM 4.1. There is a representation of an n -node ordinal tree which occupies $2n + O(n \lg \lg \lg n / \lg \lg n)$ bits and supports the operations ANC , CHILD , CHILDRANK , DEPTH and DEG in $O(1)$ time. The representation labels the nodes of the tree using integers of $\lg n + O(1)$ bits each.

4.3 Pre- and Post-order Number Operations

In this section we now assume that nodes are referred to by their pre-order number (or their post-order number). This not only gives us the promised labelling of nodes from $\{1, \dots, n\}$, but also makes RANK_{pre} and SELECT_{pre} trivial. To preserve the validity of Theorem 4.1 we must first provide conversion methods between τ -names and pre-/post-order numbers. We then show that we are able to support the operations RANK_z , SELECT_z (for $z \in \{pre, post\}$) and DESC in $O(1)$ time.

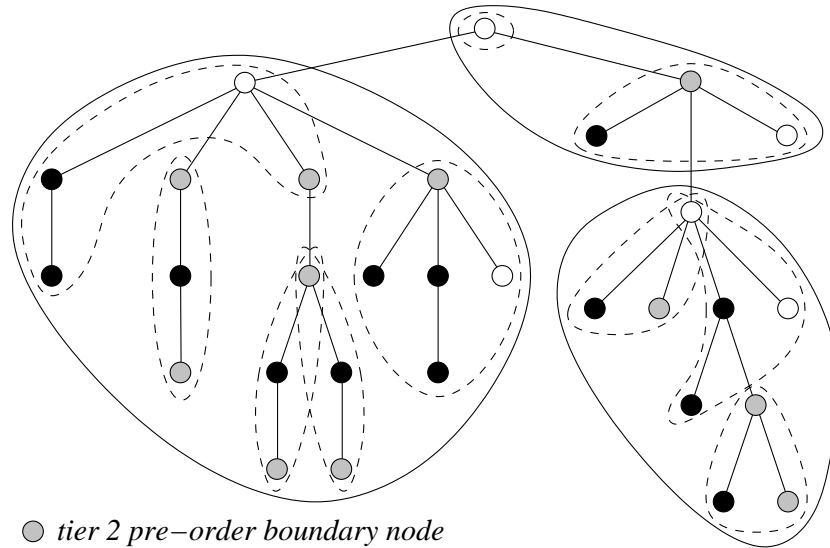
4.3.1 Converting pre-/post-order numbers to τ -names. We store a compressed bitvector B_1 of length n , whose i -th bit is 1 if the i -th node in pre-order is a tier 1 pre-order boundary node (in what follows, we temporarily drop the qualifier ‘pre-order’). Another bitvector B_2 marks the tier 2 boundary nodes. By Theorem 2.1, the bitvectors occupy $O(n \lg \lg n / \lg n) = o(n)$ bits.

We also store two arrays C_1 and C_2 , as follows. Let $1 = \text{root}(T) = x_1 < x_2 < \dots < x_{n_1} = n$ be the tier 1 boundary nodes. By Lemma 2.2, all nodes x such that $x_i < x < x_{i+1}$ belong to the same mini-tree, say t_j ; we let $C_1[i] = j$. (If no

such node x exists, then if x_i is the root of a micro-tree we let $C_1[i] = \tau_1(x_i)$, otherwise, we let $C_1[i] = \tau_1(x_{i+1})$. As a τ_1 -name is $O(\lg n)$ bits long, and the array C_1 contains $O(n/M)$ entries, C_1 occupies $O(n / (\lg n)^3) = o(n)$ bits in all.

Likewise, we store in C_2 , for each sequence of nodes that belong to the same micro-tree μ , the τ_2 -name of this micro-tree. A little care must be taken, however, to ensure that the data stored allows τ_3 -names to be computed correctly. For this, entries in C_2 are of the form $\langle q, r, b \rangle$, where q and r are data for calculating τ_2 and τ_3 names, respectively, and b is a 0/1 value. The details are as follows.

Let x_i be the i -th tier 2 boundary node. If x_i is the root of some micro-tree we set $C_2[i] = \langle \tau_2(x_i), 1, 1 \rangle$. Otherwise, let y be the last node (if any) before x_i such that we have not yet visited all nodes in y 's micro-tree, and such that y is in the same mini-tree as x_i . If y exists, we set $C_2[i] = \langle \tau_2(y), \tau_3(y), 0 \rangle$. Otherwise, we set $C_2[i] = \langle -, -, 0 \rangle$ (see Figure 4.2). As a τ_2 -name and a τ_3 -name are both $O(\lg \lg n)$ bits long, and the array C_2 contains $O(n / \lg n)$ entries, C_2 occupies $O(n \lg \lg n / \lg n) = o(n)$ bits in all.



- tier 2 pre-order boundary node
- tiers 1 and 2 pre-order boundary node

node type	○	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
B_1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
B_2	1	1	0	0	1	0	1	1	1	0	1	0	1	1	0	0	0	1	1	0	1
C_1	τ_1	1	2													1			3		1
C_2	τ_2	1	1	2	1	-	3	4	-	5						-	2	1	2	3	2
	τ_3	1	1	1	3	-	1	1	-	1						-	1	1	1	1	3
	root?	1	1	1	0	0	1	0	0	1						0	1	1	0	1	0

Figure 4.2 Pre-order node sequence of the example tree; lines divide the sequence into mini/micro tree subsequences, while corresponding values in the arrays B_1 , B_2 , C_1 and C_2 are shown below it. (- denotes values that will never be accessed.)

If x is a non-boundary node, we compute $\tau(x)$ in $O(1)$ time as follows. We perform RANK1 queries on B_1 and B_2 and use the results to index into C_1 and C_2 , respectively, obtaining some values p and $\langle q, r, b \rangle$. Then $\tau_1(x) = p$, $\tau_2(x) = q$ and $\tau_3(x) = r + (x - x')$, where x' is the pre-order number of the tier 2 boundary node we used to index into C_2 (which can be found by a SELECT1 query on B_2). Now suppose x is a boundary node. If x is a boundary node that is the root of a micro-tree (the final value stored in C_2 lets us determine this), the above procedure works. If not, let p and $\langle q, r, b \rangle$ denote the last entries of C_1 and C_2 before the entries corresponding to x . Then $\tau_1(x) = p$, $\tau_2(x) = q$ and $\tau_3(x) = r + (x - x')$, where x' is the pre-order number of the last tier 2 boundary node before x .

To convert from post-order numbers to τ -names, we use an analogous approach, storing compressed bitvectors B_3, B_4 and arrays C_3, C_4 .

4.3.2 *Converting τ -names to pre-/post-order numbers.* Let $in(t)$ for some mini- or micro-tree t be defined as $pre(second(t)) - 1$ (the pre-order number of the node before the second node in t). If C_t is the set of all children of $root(t)$ that have an (original or promoted) node inside t , then let $out(t)$ be defined as the largest pre-order number among the descendants of C_t . With the root of each mini-tree t we store these numbers $in(t)$ and $out(t)$. As in and out are $\Theta(\lg n)$ -bit numbers, storing them explicitly for mini-trees costs $o(n)$ bits, but we cannot afford to do the same for micro-trees. We now discuss how to store $O(\lg \lg n)$ bits per micro-tree root and still compute in and out in $O(1)$ time for a micro-tree root, focussing on the in values; out is very similar.

Intuitively, with the root of each micro-tree μ_i , contained within a mini-tree t , we store an $O(\lg \lg n)$ -bit “pointer” to an appropriate mini-tree root r , whose in or out value is within $(\lg n)^{O(1)}$ of $in(\mu_i)$, together with the difference in pre-order between r and $root(\mu)$. In more detail, we consider two cases. In a pre-order traversal, the sequence s of nodes between $root(t)$ and $second(\mu_i)$ may either contain nodes in other mini-trees (case 1) or it may not (case 2).

Case 1: There are a number, $j \geq 1$, of mini-trees t_1, \dots, t_j contained in s such that the parent of the root of each mini-tree is in t . Let t_j be the mini-tree in this set with the highest τ_1 number (and therefore the last of these mini-trees to be visited before $second(\mu_i)$ in a pre-order traversal). We store:

- a. $last_i$, the $\langle \tau_2, \tau_3 \rangle$ -name of $parent(root(t_j))$;
- b. $lsib_i$, the $\langle \tau_2, \tau_3 \rangle$ -name of the first sibling to the right of $root(t_j)$, if it exists as an original node within t , and NULL otherwise; and
- c. $ldist_i$, the difference in pre-order between $out(t_j)$ and $in(\mu_i)$.

Case 2: We store:

- a. $last_i = \text{NULL}$;
- b. $lsib_i = \text{NULL}$; and
- c. $ldist_i$, the difference in pre-order between $in(t)$ and $in(\mu_i)$.

We can now calculate $in(\mu_i)$ for a micro-tree μ_i in $O(1)$ time in one of three ways:

- a. If $last_i = \text{NULL}$ then $in(\mu_i) = in(t) + ldist_i$; otherwise
- b. If $lsib_i = \text{NULL}$ then we find t_j by performing the query $\text{CHILD}(last_i, \text{DEG}(last_i))$, and calculate $in(\mu_i) = out(t_j) + ldist_i$; otherwise
- c. we find t_j by performing the query $\text{CHILD}(last_i, \text{CHILDRANK}(lsib_i) - 1)$, and calculate $in(\mu_i) = out(t_j) + ldist_i$.

In a similar manner, to enable us to calculate $out(\mu_i)$, we do the following. Let t' be the first mini-tree (if any), the parent of whose root is in t , that is visited after $root(\mu_i)$. We store $next_i$, which equals the $\langle \tau_2, \tau_3 \rangle$ -name of $parent(root(t'))$ if t' exists, and NULL otherwise; $nsib_i$, the $\langle \tau_2, \tau_3 \rangle$ -name of the first sibling to the left of $root(t_1)$, if it exists in t , and NULL otherwise; and $ndist_i$, which equals the difference in pre-order between $out(\mu_i)$ and $in(t')$, if t' exists, and the difference between $out(\mu_i)$ and $in(t)$, otherwise. We can now calculate $out(\mu_i)$ for a micro-tree μ_i in $O(1)$ time as above.

Since $last_i$, $lsib_i$, $next_i$ and $nsib_i$ are in t , we can safely omit their τ_1 components. The value of $ldist_i$ and $ndist_i$ is never more than the number of nodes in a mini-tree. Therefore, we are able to store this data in $O(\lg \lg n)$ bits.

mini-tree τ -name	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle 3 \rangle$
$in(t)$	1	2	21
$out(t)$	30	18	29

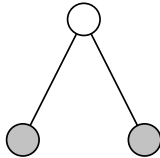
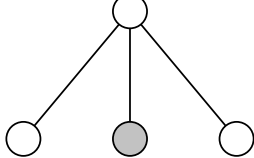

micro-tree τ -name	$\langle 1, 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 2, 4 \rangle$
tree structure			
$last_i$	NULL	$\langle 1, 1 \rangle$	NULL
$lsib_i$	NULL	$\langle 2, 1 \rangle$	NULL
$ldist_i$	0	1	7
$next_i$	NULL	NULL	NULL
$nsib_i$	NULL	NULL	NULL
$ndist_i$	0	0	5

Figure 4.3 Example representation of τ -name to post order conversion data stored with the three mini-trees and three selected extended micro-trees from the tree in Figure 2.2. White denotes original nodes while grey denotes promoted nodes.

Next, we show how to calculate the pre-order number of a given original node x with $\tau(x) = \langle i, j, k \rangle$. We consider 3 possible cases.

Case1: There exists a promoted node in the extended micro-tree $\langle i, j \rangle$ with pre-order number less than x . Let x' be the last promoted node in $\langle i, j \rangle$ before x in pre-order and d be the difference in pre-order between x and x' . These can be found by table lookup for type 1 extended micro-trees. For type 2 extended micro-trees, we look at $nodetypes_{\langle i, j \rangle}$; x' is the $\text{SELECT0}(\text{RANK0}(\text{SELECT1}(k)))$ -th node and $d = \text{SELECT1}(k) - \text{SELECT0}(\text{RANK0}(\text{SELECT1}(k)))$. We find the τ -name of x' as before (Section 4.3.1), determining the (micro- or mini-) tree $\mu_{j'}$ that x' is a root of. The pre-order number of x is now $out(\mu_{j'}) + d$.

Case 2: No such node x' exists and $\langle i, j \rangle$ is the **first** micro-tree rooted at $\langle i, j, 1 \rangle$. The pre-order number of x is simply $in(\mu_i) + d$.

Case 3: No such node x' exists and $\langle i, j \rangle$ is not the first micro-tree rooted at $\langle i, j, 1 \rangle$. We need to find the last micro-tree $\mu_{j''}$ rooted at $\langle i, j, 1 \rangle$ that comes before $\langle i, j \rangle$. If $j \neq 1$, then $\mu_{j''} = \langle i, j - 1 \rangle$, otherwise we need to find the micro-tree $\mu_{j'''}$ containing the canonical copy of $\langle i, j, 1 \rangle$ (one way to do this would be to convert $in(\mu_j)$ to a τ -name). We then do a RANK query and an array access respectively on the bitvector $child_i$ (micro) and array $edges_i$ (τ_2) associated with $\mu_{j'''}$ to obtain $\mu_{j''} = \langle i - 1, j' \rangle$, where $j' = edges_i(\text{RANK1}_{child_i}(\text{CHILDRANK}(\langle i, j, 1 \rangle))) - 1$. The pre-order number of x is now $out(\mu_{j''}) + d$.

A related approach allows us to convert between post-order numbers and τ -names in $O(1)$ time using similar additional data structures.

4.3.3 RANK, SELECT and DESC operations. RANK and SELECT operations in constant time now become trivial: we simply convert to pre- (or post-) order names. Finally, for a node x , $\text{DESC}(x) = 1 + \text{RANK}_{post}(x) - \text{RANK}_{pre}(x) + \text{DEPTH}(x)$. We have thus shown:

THEOREM 4.2. There is a representation of an n -node ordinal tree which occupies $2n + O(n \lg \lg n / \lg \lg n)$ bits and supports the operations ANC, CHILD, CHILDRANK, $\text{RANK}_z/\text{SELECT}_z$ for $z \in \{pre, post\}$, DESC, DEPTH and DEG in $O(1)$ time. The representation labels the nodes of the tree using integers from $\{1, \dots, n\}$.

5. OPERATIONS ON LABELLED TREES AND REPRESENTING XML DOCUMENTS

5.1 Operations on Labelled Trees

The previous section described a data structure to represent an ordinal tree. We now consider representing an ordinal tree where each node in the tree is labelled with a symbol from an alphabet Σ . We would like to support ‘labelled’ versions of each of the previously defined operations:

- $\text{RANK}_z(x, \sigma)$: return the number of nodes that have label σ and precede x in z -order, for $z \in \{pre, post\}$;
- $\text{SELECT}_z(i, \sigma)$: return the i -th node in z -order that has label σ , for $z \in \{pre, post\}$;
- $\text{CHILD}(x, \sigma, i)$: for $i \geq 1$ return the i -th child of node x ;
- $\text{DEG}(x, \sigma)$: return the number of child nodes of x labelled σ ;
- $\text{CHILDRANK}(x, \sigma)$: return the number of siblings of x with label σ that precede x ;
- $\text{DEPTH}(x, \sigma)$: return the number of ancestors of x that have label σ ;
- $\text{DESC}(x, \sigma)$: return the number of descendants of x that have label σ , including x itself; and
- $\text{ANC}(x, \sigma, i)$: for $i \geq 0$ return the i -th ancestor of node x that has label σ .

The information-theoretic space lower bound for these kinds of labelled trees is easily seen to be $n(\lg |\Sigma| + 2) - O(\lg n)$. Suppose that the representation of Theorem 4.2 uses $2n + f(n)$ bits. We sketch a fairly easy modification of this representation, that uses $n(\lg |\Sigma| + 2) + O(|\Sigma|f(n))$ bits and supports the labelled operations in $O(1)$ time as well. Thus, if $|\Sigma|$ is a constant the labelled operations are also handled optimally. Indeed, in Theorem 4.2, we noted that $f(n) = O(n \lg \lg n / \lg \lg n)$, so the representation uses optimal space, to within lower-order terms, for $|\Sigma| = o(\lg \lg n)$.

We sketch the modifications required (assuming $|\Sigma| = o(\lg \lg n)$). Firstly, we reduce M' (the size of micro-trees) to $\max\left(\left\lceil \frac{\lg n}{24 \lg |\Sigma|} \right\rceil, 2\right)$ (and similarly change the threshold for type 1/2 micro-trees); this lets all labels of a type 1 micro-tree μ be stored along with its tree structure in $\lceil |\mu| \lg |\Sigma| \rceil + 2|\mu|$ bits whilst allowing table lookup. For type 2 micro-trees we store $|\Sigma|$ more bitvectors, one for each $\sigma \in \Sigma$, that indicate if the i -th node is labelled σ . At the (canonical copy of the) root r_i of a micro-tree μ_i , in addition to $child_i$, we store several bitvectors as follows. If r_i 's children are in l different micro-trees, and for $j = 1, \dots, l$ there are $a^{\sigma_j} \geq 0$ children labelled σ in the j -th micro-tree, then the sequence $0^{a^{\sigma_1}} 1 0^{a^{\sigma_2}} 1 \dots 0^{a^{\sigma_l}} 1$ is stored as a compressed bitvector (there are $2|\Sigma|$ such bitvectors, one for

each $\sigma \in \Sigma$ and one each for the partition of a node's children induced by micro- and mini-trees). If the degree of r_i is d_i , then the total sizes of all bitvectors (including $child_i$) is bounded by $\lg \binom{O(d_i + l|\Sigma|)}{O(l|\Sigma|)}$ plus lower-order terms; this sums up to $o(n)$ over all roots.

Finally, we store analogues of $last_i, in_i, out_i$ for each $\sigma \in \Sigma$. It is easy to verify that this information suffices to handle the labelled versions of all operations except ANC.

For ANC we make the following observations. By storing separate skip pointers for each label, we can find labelled ancestors within mini- and micro- trees in $O(1)$ time. These skip pointers use $o(n)$ space. The major difference, however, is in the macro tree, where we need to find the nearest macro node to the ancestor that we seek. In the unlabelled case, successive macro nodes in the macro tree are separated by exactly M real nodes. However, in the labelled version, there may be a variable number of nodes of a particular colour lying between any two successive macro nodes. We store $|\Sigma|$ copies of the macro tree, one for each $\sigma \in \Sigma$. On each copy, we now have to solve a weighted version of the level-ancestor problem, where the weight of an edge between two macro nodes for some $\sigma \in \Sigma$ gives the number of nodes labelled with σ between the two macro nodes. Specifically, we wish to answer the query $succ(x, d)$, which returns the last (highest) node z on the path from x to the root, such that the sum of weights on the path between x and z is less than d . The distance between two successive macro nodes is $O((\lg n)^4)$, so the weight of an edge in the macro tree is also $O((\lg n)^4)$. As the weights are polylogarithmic in n , Alstrup and Holm [1] can solve this problem in $O(1)$ time (in contrast to the $O(\lg \lg n)$ solutions proposed by [11, 12]). We have thus shown:

THEOREM 5.1. There is a representation of an n -node ordinal tree, where each node is labelled with some $\sigma \in \Sigma$, $|\Sigma| = o(\lg \lg n)$, which occupies $(\lg |\Sigma| + 2)n + o((\lg |\Sigma| + 2)n)$ bits and supports the labelled versions of the operations ANC, CHILD, CHILDANK, RANK _{z} /SELECT _{z} for $z \in \{pre, post\}$, DESC, DEPTH and DEG in $O(1)$ time. The representation names the nodes of the tree using integers from $\{1, \dots, n\}$.

5.2 Representing XML documents

As noted in the introduction, an XML tree is an ordinal tree where each node is labelled with a tag name σ from a set of tags T . We consider the following *location step* problem, which appears to be an important subtask for evaluating Location Path Expressions (LPEs) in XPath. A single location step takes as input a context node c , and an *axis*

specifier, and a *node test*, to select the appropriate nodes(s) along these axis specifiers. There are 13 axis specifiers, but we only consider the non-trivial ones below⁵:

child	All children of the context node.
parent	The parent of the context node.
descendant	All descendants of the context node.
ancestor	All ancestors of the context node.
following	All nodes after the context node in pre-order, excluding any descendants of the context node.
preceding	All nodes before the context node in pre-order, excluding any ancestors of the context node.
following-sibling	All siblings that come after the context node.
preceding-sibling	All siblings that come before the context node.

To a first approximation, a node test is simply a label $\sigma \in T$, and an LPE of the form $axis : \sigma$ ensures that only nodes with label σ along the appropriate axis are considered. Of particular interest to us is the ability to randomly index into the set of selected nodes along a given axis. Thus, a LPE of the form $axis : \sigma[i]$ refers to the i -th node labeled σ along *axis*. (e.g., in Figure 1.2, the LPE $descendant : NOTE[4]$, evaluated at the root, returns the root's rightmost child). If c is the context node, then selecting the i -th node labelled σ along the above axes may be implemented as follows:

$child(c, \sigma, i)$	$CHILD(c, \sigma, i)$
$parent(c, \sigma)$	$ANC(c, \sigma, 1)$
$descendant(c, \sigma, i)$	$SELECT_{pre}(RANK_{pre}(c, \sigma) + i, \sigma)$, if $i \leq DESC(c, \sigma)$
$ancestor(c, \sigma, i)$	$ANC(c, \sigma, i)$
$following(c, \sigma, i)$	$SELECT_{pre}(RANK_{pre}(c, \sigma) + DESC(c, \sigma) + i, \sigma)$
$preceding(c, \sigma, i)$	$SELECT_{post}(RANK_{post}(c, \sigma) - DESC(c, \sigma) - i, \sigma)$
$following-sibling(c, \sigma, i)$	$CHILD(ANC(c, \sigma, 1), \sigma, CHILDRANK(c, \sigma) + i)$
$preceding-sibling(c, \sigma, i)$	$CHILD(ANC(c, \sigma, 1), \sigma, CHILDRANK(c, \sigma) - i)$

To apply this labelled tree representation to XPath however, requires a little care. A node test, in reality, is not just a tag name. XPath considers each node to be one of 7 types: root, element (tag), attribute, comment, namespace, processing instruction or text.⁶ If we have a LPE of the form $axis : \sigma[i]$, σ does not have to be a specific tag name, and instead could be one of the following:

⁵ The omitted axis specifiers are `self`, `ancestor-or-self`, `descendant-or-self`, `attribute` and `namespace`.

⁶ However, attributes and namespaces are not considered to be actual nodes in the ordinal tree representing the XML document.

*	selects from the set of element nodes;
text()	selects from the set of text nodes;
comment()	selects from the set of comment nodes;
processing-instruction()	selects from the set of processing instruction nodes; and
node()	selects from the complete set of all nodes.

Therefore we wish the set of labels to be $\Sigma = T \cup \{*, \text{text}(), \text{comment}(), \text{processing-instruction}(), \text{node}()\}$. The addition of the node sets $*$ and $\text{node}()$ create a problem, because they are not disjoint from the other labels ($* = \bigcup T$ and $\text{node}() = \bigcup \{*, \text{text}(), \text{comment}(), \text{processing-instruction}()\}$). We need to create auxiliary data structures for $*$ and $\text{node}()$ like those for the other labels. Because each of these auxiliary data structures uses $o(n)$ bits, and there are only two extra ones to create, the space bound stated in Theorem 5.1 is still valid for a data structure supporting XPath queries.

6. CONCLUSION

We have described a $2n + o(n)$ bit representation of an n -node static ordinal tree, which supports the operations of RANK_z , SELECT_z , for $z \in \{\text{pre}, \text{post}\}$; CHILD , DEG , CHILDRANK , DEPTH , DESC and ANC in $O(1)$ time on the RAM model of computation. This set of operations is essentially the union of the sets of operations supported by previous succinct representations, to which we added the ANC operation, and it is also relevant to the processing of XML documents. We use the approach of covering the given ordinal tree with a number of small, connected trees. There appears to be no particular reason why this approach can support precisely the set of operations we consider, and no other operations. It would be interesting to characterise which kinds of operations are intrinsically supported by representations based on this approach.

Finally, we modified the ordinal tree representation to allow nodes to be labelled with symbols from an alphabet, while supporting labelled versions of the above operations. Our representation of labelled trees uses optimal space, to within lower-order terms, but only for very small alphabet sizes. Nevertheless, it is a first step towards the succinct representation of XML documents. An obvious question is whether a labelled tree representation can be found that remains space-efficient for larger alphabets, and [13] is a very recent advance in this direction.

REFERENCES

1. ALSTRUP, S. AND HOLM, J. Improved algorithms for finding level ancestors in dynamic trees. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Proceedings*. Lecture Notes in Computer Science 1853 Springer 2000, pp. 73-84.
2. APACHE SOFTWARE FOUNDATION. Apache Xindice: Frequently Asked Questions. <http://xml.apache.org/xindice/faq.html>, October 2005.
3. BENDER, M.A. AND FARACH-COLTON, M. The level ancestor problem simplified. In *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Proceedings*. Lecture Notes in Computer Science 2286 Springer 2002, pp. 508-515.
4. BENOIT, D.A., DEMAINE, E.D., MUNRO, J.I., RAMAN, R., RAMAN, V., AND RAO, S.S. Representing trees of higher degree. *Algorithmica* **43** (2005), pp. 275-292.
5. BENOIT, D.A., DEMAINE, E.D., MUNRO, J.I., AND RAMAN, V. Representing trees of higher degree. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Proceedings*. Lecture Notes in Computer Science 1663 Springer 1999, pp. 169-180.
6. BERKMAN, O. AND VISHKIN, U. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, **48** (1994), pp. 214-230.
7. CHAZELLE, B. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, **2** (1987), pp. 337-361..
8. CHIANG, Y.-T., LIN, C.-C. AND LU, H.-I. Orderly spanning trees with applications. *SIAM Journal on Computing*, **34** (2005), pp. 924-945. Preliminary version in *SODA 2001*.
9. CLARK, D. AND MUNRO, J.I. Efficient suffix trees on secondary storage (extended Abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, 1996, pp. 383-391.
10. CLARK, J. AND DEROSE, S. XML path language (XPath) Version 1.0. W3C Recommendation. <http://www.w3.org/TR/xpath>, *W3C Consortium*, 1999.
11. DIETZ, P.F. Finding level-ancestors in dynamic trees. In *Algorithms and Data Structures, 2nd Workshop WADS '91, Proceedings*. Lecture Notes in Computer Science 519 Springer 1991, pp. 32-40.
12. FARACH, M. AND MUTHUKRISHNAN, S. Perfect hashing for strings: formalization and algorithms. In *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Proceedings*. Lecture Notes in Computer Science 1075 Springer 1996, pp. 130-140.
13. FERRAGINA, P., LUCCIO, F., MANZINI, G AND MUTHUKRISHNAN, S.. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 184-196, 2005.
14. LE HORS, A., LE HÉGARET, P., WOOD, L., NICOL, G., ROBIE, J., CHAMPION, M. AND BYRNE, S. Document object model (DOM) level 2 core specification Version 1.0. W3C Recommendation. <http://www.w3.org/TR/DOM-Level-2-Core>, *W3C Consortium*, 2000.
15. JACOBSON, G. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*. pp. 549-554, 1989.
16. MUNRO, J.I. AND RAMAN, V. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, **31** (2001), pp. 762-776.
17. MUNRO, J.I. AND RAO, S.S. Succinct representations of functions. In *Proceedings of the 31st International Colloquium on Automata, Languages and Computation, LNCS 3142*, pp. 1006-1015, 2004.
18. RAMAN, R., RAMAN, V., AND RAO, S.S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, 2002, pp. 233-242.
19. RUDIN, W. *Real & Complex Analysis, 3rd ed.* McGraw-Hill, 1987.