Representing Trees of Higher Degree*

David Benoit† Erik D. Demaine‡ J. Ian Munro§ Rajeev Raman¶
Venkatesh Raman‖ S. Srinivasa Rao§

Abstract

This paper focuses on space efficient representations of rooted trees that permit basic navigation in constant time. While most of the previous work has focused on binary trees, we turn our attention to trees of higher degree. We consider both cardinal trees (or k-ary tries), where each node has k slots, labelled $\{1, \ldots, k\}$, each of which may have a reference to a child, and ordinal trees, where the children of each node are simply ordered. Our representations use a number of bits close to the information theoretic lower bound and support operations in constant time. For ordinal trees we support the operations of finding the degree, parent, ith child and subtree size. For cardinal trees the structure also supports finding the child labeled i of a given node apart from the ordinal tree operations. These representations also provide a mapping from the n nodes of the tree onto the integers $\{1, \ldots, n\}$, giving unique labels to the nodes of the tree. This labelling can be used to store satellite information with the nodes efficiently.

1 Introduction

Trees are a fundamental structure in computing. They are used in almost every aspect of modeling 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

*Preliminary versions of different portions of this paper appeared in [2], [3] and [30]. Work supported in part by the Canada Research Chairs Programme, EPSRC grant GR L/92150, NSERC grant RGPIN8237-0220646 and UK-India Science and Technology Research Fund project number 2001.04/IT.

†Invio Bioinformatics, Inc., 1721 Lower Water St., Halifax, NS B3J 2S5, Canada, benoit@invio.ca

‡MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, edemaine@mit.edu

§School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {imunro, ssrao}@uwaterloo.ca

¶Department of Computer Science, University of Leicester, Leicester LE1 7RH, UK, r.raman@mcs.le.ac.uk

‖Institute of Mathematical Sciences, Chennai, India 600 113, vraman@imsc.ernet.in

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. For example, suffix trees on binary alphabets (which are indeed binary trees) were developed for the purpose of indexing large files to permit full text search. That is, a suffix tree permits searches in time bounded by the length of the input query, and in that sense is independent of the size of the database. However, assuming our query phrases start at the beginning of words and that words of text are on average 5 or 6 characters in length, we have an index of about 3 times the size of the text. That the index contains a reference to each word of the text accounts for less than a third of this overhead. Most of the index cost is in storing its tree structure. Indeed this is the main reason for the proposal [14, 19] of simply storing an array of references to positions in the text rather than the valuable but costly structure of the tree.

These and many other applications deal with large static trees. The representation of a tree is required to provide a mapping from the n nodes to $\{1, \ldots, n\}$. Any information that needs to be stored for the application (e.g., the location of a word in the database) is found through this mapping. The suffix tree applications used binary trees, though trees of higher degree, for example degree 256 for text or 4 for DNA sequences, might be better. Trees of higher degree, i.e. greater than 2, are the focus of this paper.

Starting with Jacobson [16, 17] some attention has been focused on succinct representation of trees—that is, on representations requiring close to the information theoretic number of bits necessary to represent objects from the given class, but on which a reasonable class of primitive operations can be performed quickly. Such a claim requires a clarification of the model of computation. The information theoretic lower bound on space is simply the logarithm to the base 2 (denoted lg) of the number of objects in the class. The number of binary (cardinal) trees on n nodes is $C_n \equiv \binom{2n+1}{n}/(2n+1)$ [15]; $\lg C_n = 2n - \Theta(\lg n)$. Jacobson's goal was to navigate around the tree with each step involving the examination of only $O(\lg n)$ bits of the representation. As a consequence, the bits he inspects are not necessarily close together. If one views a word as a sequence of $\lg(n+1)$ consecutive bits, his methods can be shown to involve inspecting $\Theta(\lg\lg(n+1))$ words. We adopt the model of a random access machine with a $\lg(n+1)$ (or so) bit word. Basic operations include the usual arithmetics and shifts. Fredman and Willard [12, 13] call this a *transdichotomous model* because the dichotomy between the machine model and the problem size is crossed in a reasonable manner.

Clark and Munro [6, 7] followed the model used here and modified Jacobson's approach to achieve constant time navigation. They also demonstrated the feasibility of using succinct representations of binary trees as suffix trees for large-scale full-text searches. Their work emphasized the importance of the subtree size operation, which indicates the number of matches to a query without having to list all the matches. As a consequence, their implementation was ultimately based on a different, $3n$ bit representation that included subtree size but not the ability to move from child to parent. Munro and Raman [24] essentially closed the issue for *binary* trees by achieving a space bound of $2n+o(n)$ bits, while supporting the operations of finding the parent, left child, right child, and subtree size in constant time. Recently, Munro et al. [26] have given a dynamic binary tree representation taking the same amount of space.

Trees of higher degree are not as well studied. There are essentially two forms to study, which we call ordinal trees and cardinal trees. An *ordinal tree* is a rooted tree of arbitrary degree in which the children of each node are ordered, hence we speak of the ith child. The one-to-one mapping between these trees and binary trees is a well known undergraduate example [20, p. 333], and so about $2n$ bits are necessary for representation of such a tree. Jacobson [16, 17] gave a $2n + o(n)$ bit structure to represent ordinal trees and efficiently support queries for the degree, parent or ith child of a node. The improvement of Clark and Munro [7] leads to constant execution for these operations. However, determining the size of a subtree essentially requires a traversal of the subtree. In contrast, Munro and Raman [24] implement parent and subtree size in constant time, but take $\Theta(i)$ time to find the ith child, and $\Theta(d)$ time to find the degree d of a given node. Their representation was augmented by Chiang et al. [5] to support the degree operation in $O(1)$ time. The structure presented here performs all four operations in constant time, in the same optimal space bound of $2n + o(n)$ bits.

By a *cardinal tree* (or trie) of degree k, we mean a rooted tree in which each node has k positions for an edge to a child. Each node has up to k children and each child of a given node is labeled by a unique integer from the set $\{1, 2, \ldots, k\}$. A binary tree is a cardinal tree of degree 2. Since there are $C_n^k \equiv \binom{kn+1}{n}/(kn+1)$ cardinal trees of degree k [15], $\lg C_n^k = (k \lg k - (k-1)\lg(k-1))n - O(\lg(kn))$ bits is a lower bound on the space required to store a representation of an arbitrary k-ary cardinal tree, for fixed k and n increasing. If k is viewed as a (slowly-growing) function of n, then this bound approaches $(\lg k + \lg e)n$ bits. Our techniques answer queries asking for parent, the child with label i and subtree size in constant time. The structure requires $(\lceil \lg k \rceil + 2)n + o(n) + O(\lg\lg k)$ bits. This can be written to more closely resemble the lower bound as $(\lceil \lg k \rceil + \lceil \lg e \rceil)n + o(n) + O(\lg\lg k)$ bits.

Our result is related to, but more precise in terms of model than, that of [8]. More recently, building on the results of this paper, some of us have given a representation of k-ary that supports all the above operations *except* the subtree size operation in $O(1)$ time, but uses $\lg C_n^k + o(n + \lg k)$ bits [29].

The rest of this paper is organized as follows. Section 2 describes previous encodings of ordinal trees. These techniques are combined in Section 3 to achieve an ordinal tree encoding supporting all the desired operations in constant time. Section 4 extends this structure to support cardinal trees. Finally, in Section 5, we tune our results to the particular case of degree 4, the size of the alphabet describing DNA sequences.

2 Previous Work

First we outline two ordinal tree representations that use $2n + o(n)$ bits, but do not support all of the desired operations in constant time.

2.1 Jacobson's Ordinal Tree Encoding

Jacobson's [16] encoding of ordinal trees represents a node of degree d as a string of d **1**s followed by a **0**, which we denote $\mathbf{1}^d\mathbf{0}$. Thus the degree of a node is represented by a simple binary prefix code, obtained from terminating the unary encoding with a **0**. These prefix codes are then written in a level-order traversal of the entire tree. This method is known as the level-order unary degree sequence representation (which we abbreviate to LOUDS), an example of which is given in Fig. 1(b). Using auxiliary structures for the so-called *rank* and *select* operations (see Section 2.1.1), LOUDS supports, in constant time, finding the parent, the ith child, and the degree of any node.

Every node in the tree, except the root node, is a child of another node, and therefore has a **1** associated with it in the bit-string. The number of **0**s in the bit-string is equal to the number of nodes in the tree, because the description of every node (including the root node) ends with a **0**. Jacobson introduced the idea of a 'superroot' node which simply prefixes the representation with a **1**. This satisfies the idea of having 'one **1** per node', thus making the total length of the bit-string $2n$. Unfortunately, the LOUDS representation is illsuited to computing the subtree size, because in a level-order encoding, the information dealing with any subtree is likely to spread throughout the encoding.

2.1.1 Rank and Select

The operations performed on Jacobson's tree representation require the use of two auxiliary structures: the rank and select structures [6, 17, 23, 24]. These structures support the following operations, which are used extensively, either directly or implicitly, in all subsequent work, including this paper:

Rank: $rank_1(j)$ returns the number of **1**s up to and including position j in an n bit string. One can support this operation in $O(1)$ time by augmenting the bit string with an auxiliary $o(n)$ bit structure [17, 23]. $rank_0(j)$ is the analogous function counting the **0**s.

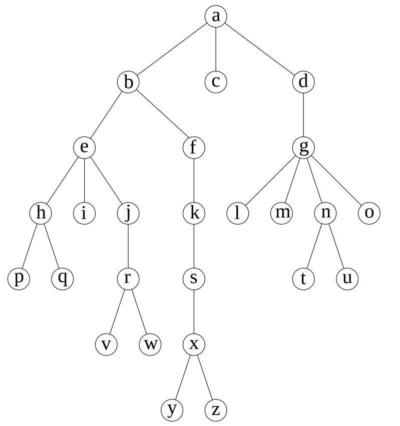
Select: $select_1(j)$ returns the position of the jth **1**. It also requires an auxiliary $o(n)$ bit structure [17]. Jacobson's method takes more than constant time, but inspects only $O(\lg n)$ bits. The modification by Munro [23] reduces this to $\Theta(1)$ time, on RAM with word-size $\Theta(\lg n)$ bits. $select_0(j)$ is the analogous function locating a **0**.

The auxiliary structures for rank [6, 17] are constructed as follows:

- Conceptually break the array into blocks of length $\lceil(\lg n)^2\rceil$. Keep a table containing the number of **1**s up to the last position in each block.
- Conceptually break each block into sub-blocks of length $\left\lceil\frac{1}{2}\lg n\right\rceil$. Keep a table containing the number of **1**s within the block up to the last position in each sub-block.
- Keep a table giving the number of **1**s up to every possible position in every possible distinct sub-block. Since there are only $O(\sqrt{n})$ distinct possible sub-blocks and $O(\lg n)$ positions, this takes $O(\sqrt{n}\lg n \lg\lg n) = o(n)$ bits of space.

A rank query, then, is simply the sum of three values, one from each table. For select, the approach is a bit more complicated, though similar in spirit [6, 25].

Traversals on Jacobson's encoding are performed using rank and select as follows. To compute the degree of a node given the position in the bit-string, p, at which its description begins, simply determine the number of **1**s up to the next **0**. This can be done using $rank_0$ and $select_0$ by taking $select_0(rank_0(p)+1) - p$. To find the parent of p, $select_1(rank_0(p)+1)$ turns out to find the **1** in the description of the parent of p that corresponds to p. Thus, searching backwards to the previous zero (using $rank_0$ and $select_0$ operations) finds the bit before the beginning of the description of the parent. Note that the '+1' term is because of the superroot. Inverting this formula, the ith

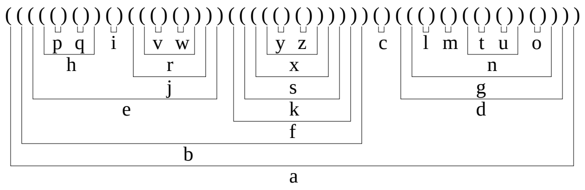


(a) The Ordinal Tree

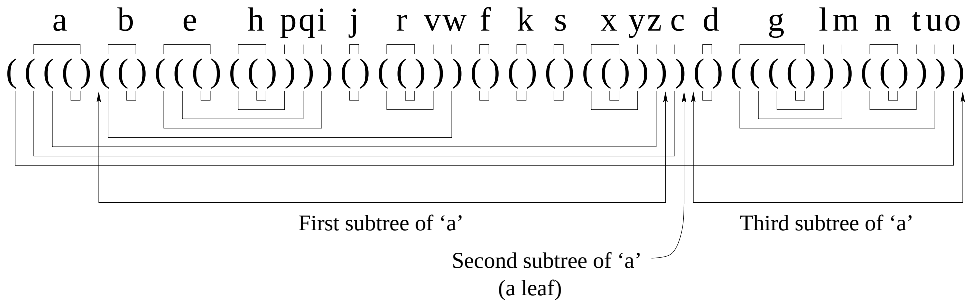
1110,110,0,10,1110,10,11110,110,0,10,10,

0,0,110,0,0,0,110,10,0,0,0,0,110,0,0

(b) Jacobson's LOUDS Representation (without the superroot). The commas have been added to aid the reader.



(c) Munro and Raman's Balanced Parentheses Representation



(d) Our *DFUDS* Representation

Figure 1: Three encodings of an ordinal tree

child is computed by $select_0(rank_1(p+i-1)-1)$. Of course, we must first check that i is at most the degree of the node.

2.2 Balanced Parentheses Representation

The binary tree encoding of Munro and Raman [24] is based on the isomorphism with ordinal trees, reinterpreted as balanced strings of parentheses. Our work is based upon theirs and we also find it more convenient to express the rank and select operations in terms of operating on parentheses. We therefore equate: $rank_{open}(j) \equiv rank_1(j)$, $rank_{close}(j) \equiv rank_0(j)$, $select_{open}(j) \equiv select_1(j)$ and $select_{close}(j) \equiv select_0(j)$. The following operations, defined on strings of balanced parentheses, can be performed in constant time [24]:

$findclose(i)$: find the position of the close parenthesis matching the open parenthesis in position i.

$findopen(i)$: find the position of the open parenthesis that matches the closing parenthesis in position i.

$excess(i)$: find the difference between the number of open and closing parentheses before position i.

$enclose(i)$: given a parenthesis pair whose open parenthesis is in position i, return the position of the open parenthesis corresponding to the closest matching parenthesis pair enclosing i.

The balanced parenthesis representation is derived from a depth-first traversal of the tree, writing a left (open) parenthesis on the way down, and writing a right (close) parenthesis on the way up. In this encoding of ordinal trees as balanced strings of parentheses, the key point is that the nodes of a subtree are stored contiguously. The size of the subtree, then, is implicitly given by the begin and end points of the encoding. Using the $findopen(i)$ and $findclose(i)$ operations, one can determine the subtree size by taking half the difference between the positions of the left and right parentheses that enclose the description for the subtree of the node. The parent of a node is also given in constant time using the $enclose()$ operation. An example of this encoding is given in Fig. 1(c).

The problem with this representation is that finding the ith child takes $\Theta(i)$ time. However, it provides an intuitive method of finding the size of any subtree. Indeed, we will use the balanced parenthesis structure in the next section for our ordinal tree representation.

3 Our Ordinal Tree Representation

Munro and Raman's representation is able to give the size of the subtree because the representation is created in depth-first order, and so each subtree is described as a contiguous balanced string of parentheses. Jacobson's representation allows access to the ith child in constant time because there is a simple relationship between a node and its children based on rank and select.

To combine the virtues of these two methods, we write the unary degree sequence of each node but in a depth-first traversal of the tree, creating what we call a depth-first unary degree sequence representation (*DFUDS*). The representation of each node contains essentially the same information as in LOUDS, written in a different order. This creates a string of parentheses which is almost balanced; there is one unmatched closing parenthesis. We will add an artificial opening parenthesis at the beginning of the string to match the closing parenthesis (like Jacobson's superroot). We use the redefinitions of rank and select in terms of strings of parentheses and the operations described in Section 2.2. An example of our encoding is given in Fig. 1(d).

Theorem 3.1 *The DFUDS representation of an ordinal tree on n nodes is a string of balanced parentheses, of length $2n$, for $n > 0$.*

Proof: The validity of the construction follows by induction and the following observations:

1. If the root has no children, then the representation is '()' (of length 2).

2. Assume that the method produces p strings, $R_1, R_2, \ldots, R_p$, of balanced parentheses for p different trees, whose total length is $2n - 2$. We must prove that the method will produce a string of balanced parentheses of length $2n$ when all p 'subtrees' are made children of a single root node (note that it would not make sense for any of these trees to be null, as they would not be included as 'children' of the new root node).

   By definition, we start the representation, R_n, of the new tree with a leading '(' followed by p '('s and a single ')' representing that the root has p children. So far, R_n is '$(({}^{p})$' meaning that there are p '('s which have to be matched.

   Next, for each i from 1 to p, strip the leading (artificial) '(' from R_i, and append the remainder of R_i to R_n. First, note that R_n gives the DFUDS representation of the new tree. Because $R_1, \ldots, R_p$ were strings of balanced parentheses, we stripped the leading '(' from each, and appended them to a string starting with p unmatched '(', the string is balanced. The total length of the representation can be easily seen to be $2n$. □

3.1 Operations

This section details how the navigation operations are performed on this representation. This leads to our main result for ordinal trees.

Theorem 3.2 *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 $\{1, \ldots, n\}$ and permits finding the degree, parent, ith child and subtree size in constant time.*

Proof: We describe procedures for performing the operations on the *DFUDS* representation of the ordinal tree. A node (with degree d) is referred to by the position of the first of the $(d + 1)$

parentheses that represent it. This gives a numbering of the nodes using integers from 1 to $2n$, which is easily converted to a number from 1 to n by means of a $rank_{close}$ operation.

Degree. The degree of a node is equal to the number of opening parentheses that are listed before the next closing parenthesis, starting from the beginning of the description of the current node. This can be found using the $rank_{close}$ and $select_{close}$ operations. More precisely, the degree of a node p is given by the expression

$$select_{close}(rank_{close}(p) + 1) - p. \tag{1}$$

***i*th Child.** From the beginning of the description of the current node:

- Find the degree d of the current node. If $i > d$, then child i cannot be present; abort.
- Jump forward $d - i$ positions. This places us at the left parenthesis whose matching right parenthesis immediately precedes the description of the subtree rooted at child i.
- Find the right parenthesis that matches the left parenthesis at the current position. The encoding of the child begins after this position.

More precisely, assuming that the node p has at least i children, the description of its ith child is a sequence of parentheses beginning at position

$$findclose(select_{close}(rank_{close}(p) + 1) - i) + 1. \tag{2}$$

Parent. From the beginning of the description of the current node:

- Find the opening parenthesis that matches the closing parenthesis that comes before the current node. (If the parenthesis before the current node is an opening parenthesis, we are at the root of the tree, which has no parent, so we abort.) We are now within the description of the parent node.
- To find the beginning of the description of the parent node, jump backwards to the first preceding closing parenthesis and the description of the parent node is after this closing parenthesis. If there are no closing parentheses before the given position, then the parent is the root of the tree (and its description starts at the beginning of the representation of the tree).

More precisely, the description of the parent of a node p is a sequence of parentheses beginning at position

$$select_{close}(rank_{close}(findopen(p - 1))) + 1. \tag{3}$$

Note that, this is correct even when the parent is the root node, because $rank_{close}(i)$ returns 0 if there are no closing parentheses up to position i in the string, and $select_{close}(0)$ returns 0.

Subtree Size. From the beginning of the description for the current node, the number of items in the subtree of the current node is equal to the number of open parentheses in the string of balanced parentheses that describes the subtree. If the current node is a leaf, then the answer is trivial. Otherwise, the algorithm is as follows:

- Find the innermost set of parentheses that enclose the current position.
- The subtree size is the difference between the number of open parentheses up to the closing parentheses of those found above, and those before the current position.

Equivalently, the size of the subtree rooted at node p is

$$rank_{open}(findclose(enclose(p))) - rank_{open}(p) + 2. \tag{4}$$

Alternatively, we can find the subtree size as half the number of characters in the string which describes the subtree, or

$$(findclose(enclose(p)) - p)/2 + 1. \tag{5}$$

□

Next we use this data structure as a component to represent cardinal trees.

4 Our Cardinal Tree Encoding

A simple cardinal tree encoding can be obtained by a slight modification to a binary tree encoding by Jacobson [16, 17]. The modified encoding for k-ary trees simply encodes a node by k bits, where the ith bit specifies whether child i is present. We call this the *bitmap representation* of a node. It can be applied to any tree ordering such as level order or, for our present purposes, depth-first order. An example of this encoding, for $k = 4$, is given in Fig. 2.

This encoding has the major disadvantage of taking kn bits, far from the lower bound of roughly $(\lg k + \lg e)n$. This section describes our method for essentially achieving 'the lower bound with ceilings added', i.e., $(\lceil \lg k \rceil + \lceil \lg e \rceil)n + o(n + \lg k)$ bits. We use as a component the succinct encoding of ordinal trees from the previous section, which takes the $\lceil \lg e \rceil n = 2n$ $(+o(n))$ term of the storage bound. The remaining storage allows us to use, for each node, 'essentially' $d\lceil \lg k \rceil$ bits to encode

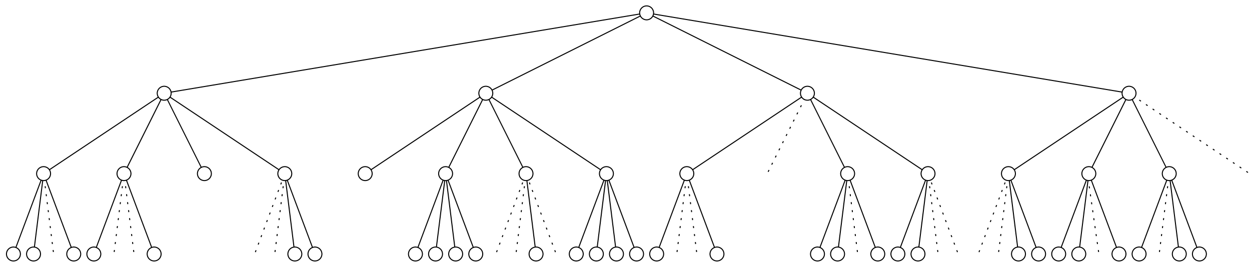


Figure 2: Generalized Jacobson Encoding of a 4-ary Tree: 1111 1111 1111 1011 1110 1101 1001 0000 0011 0000 1111 0010 1111 1001 1101 1100 0011 1101 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000—200 bits.

which children are present, where d is the number of children of that node (the qualification in the space bound is clarified below).

The ordinal tree representation in Section 3 gives, in constant time, four of the five major operations we wish to perform on cardinal trees: subtree size, ith child, parent and the degree. In a cardinal tree, we also want to perform the operation 'go to the child with label j' as opposed to 'go to the ith child'. This can be done by storing additional information at each node, which encodes the labels of the present children, and efficiently supports finding the ordinal number of the child with a given cardinal label. More precisely, when asked for the child with label j, we determine whether it exists, and if so how many children are listed before child j, i.e., its *rank* r among the children of the parent node. The desired child is thus the rth child of the node, which can be found in constant time using the ordinal tree structure.

This leads us to define the *dictionaries with rank* problem, which is to represent a subset S of a finite universe U so that the following operation can be supported in $O(1)$ worst-case time:

rank(x)**:** Given $x \in U$, return -1 if $x \notin S$ and $|\{y \in S | y < x\}|$ otherwise.

Note that if U is a range of integers, the $rank_1$ operation on the characteristic vector of S is related to *rank* above: $rank_1$ and *rank* are similar on the **1**s, but *rank* returns an uninteresting value on the **0**s. However, the data structure for *rank* must use no more than $O(|S| \log |U|)$ bits. Under this constraint, one cannot support $rank_1$ in $O(1)$ time, as we would be solving the fixed-universe predecessor problem, to which stronger lower bounds apply [1]. Thus, the weaker functionality of *rank* is essential. In our application of cardinal k-ary tree encoding, $U = \{0, \ldots, k-1\}$ is the set of child positions of a node, and the size of S is the degree d of the node.

If $k \leq (1-\varepsilon)\lg n$ for some constant $\varepsilon > 0$, there is a simple solution for this problem. As the ordinal tree representation gives us the number of children, d, of a given node, we only need to distinguish the set of labels stored at this node from among the $\binom{k}{d}$ possible subsets of size d from $\{0, \ldots, k-1\}$. Hence, we represent the set of labels *implicitly* as a $b = \lceil \lg \binom{k}{d} \rceil$-bit number which gives the index of this set in some fixed enumeration of all possible subsets of size d from $\{0, \ldots, k-1\}$. Since $b \leq d\lceil \lg k \rceil$, we can store this number in a field of $d\lceil \lg k \rceil$ bits, filling the field out with leading zeros if need be. We answer *rank* queries by using this b-bit number together with the argument to the *rank* query to index into a pre-computed table that contains the answer to the query. The pre-computed table, which is common to all nodes in the tree, has $O(k2^b)$ entries of $O(\lg k)$ bits each. Since $b \leq k \leq (1-\varepsilon)\lg n$, the table is of size $o(n)$ bits; equally, the index into this table also fits in a word of $\lg n$ bits. This approach is easily extended to the case $k = O(\lg n)$, but larger values of k need more work; this is the focus of the rest of this section.

4.1 Static Dictionary with Rank

A dictionary with rank is a generalization of a *static dictionary*, which only supports (yes/no) membership queries on S. The most space-efficient static dictionary is due to Pagh [27] and requires $\lceil \lg \binom{k}{d} \rceil + o(d) + O(\lg\lg k) = d\lg k - d\lg d + O(d + \lg\lg k)$ bits of space. Pagh's approach is based on *minimal perfect hashing* [11, 22, 31] and does not maintain the ordering of elements of S.

We begin by noting that one can add $d\lceil \lg d \rceil$ bits of explicit rank information to Pagh's approach, giving a dictionary with rank that takes $d\lg k + O(d + \lg\lg k)$ bits. In turn, we then show how to remove the additional terms and arrive at a dictionary with rank that requires $d\lceil \lg k \rceil$ bits. As it is known that $\Omega(\lg\lg k)$ bits are needed to represent minimal perfect hash functions [22], we cannot remove the $\lg\lg k$ term in the space bound for a single node using this approach. However, as the k-ary tree representation requires the storage of several dictionaries (one for each node), we share components of the hash functions across dictionaries. This reduces the space bound to $d\lg k + O(d)$ bits per node, plus shared information that adds up to $o(n) + O(\lg\lg k)$ bits over all nodes. The space per node is improved to $d\lceil \lg k \rceil$ bits by storing the rank information approximately and reconstructing the exact values during a query.

A common extension to the dictionary problem is that every element of the set S is associated with *satellite* data from a set V. A membership query '$x \in S$?' should then return the satellite data associated with x if $x \in S$. In what follows, if f is a mapping from a finite set X to a finite totally ordered set Y, by $||f||$, we mean $\max\{f(x) : x \in X\}$, and for integer $m \geq 1$, $[m]$ denotes the set $\{1, \ldots, m\}$. For $y \geq x \geq 1$, and $z \geq 1$, $c > 0$, let $g_c(x, y, z) = x(\lg y - \lg x + \lg z + c)$. If x, y, z are all positive powers of 2, g_c represents a 'nearly' space-optimal cost of storing x keys from $[y]$ with satellite information from $[z]$.

The following lemma will be used in Theorem 4.1 to augment a dictionary supporting just membership (along with some satellite information) to also support the rank operation with little extra space. The main trick is to explicitly store some partial rank information as satellite information (except for a sparse number of elements for whom the full rank information is implicitly stored). This saves a linear number of bits.

Lemma 4.1 *Let N, M be integers such that $M \geq N \geq 1$, and let $A \subseteq [M]$ with $|A| = N$. Suppose there is a constant $c > 0$ such that a dictionary for an arbitrary $A' \subseteq A$, $|A'| \geq N/2$, along with satellite information from $[s]$ for any $s \geq 1$ can be stored using at most $g_c(|A'|, M, s)$ bits to support membership queries on A' in $O(1)$ time. Then there is a dictionary with rank that stores A using at most $N \lceil \lg M \rceil$ bits that supports rank() queries in $O(1)$ time.*

Proof: Suppose without loss of generality that $N \geq 2^{c+4}$ (otherwise just list A explicitly). Let $x_0 < \ldots < x_{N-1}$ be the elements of set A. Let $4 \leq r \leq N$ be an integer and let $N' = N - \lceil N/r \rceil$.

We write down r using $\lceil \lg N \rceil$ bits and explicitly write down the keys $B = \{x_0, x_r, x_{2r}, \ldots\}$ in sorted order using $(N - N') \lceil \lg M \rceil$ bits. Finally we store $A' = A \setminus B$ in the assumed dictionary, where the key $x_i \in A'$ is stored along with satellite information $i \bmod r$. Note that $|A'| = N' \geq N/2$.

To answer *rank*(x) queries, we do a binary search to find the predecessor of x in B. If $x \in B$ we are done, as x's rank in A is easily calculated. Otherwise, we locate x in the dictionary for A', and if x is found, we take the associated satellite data and x's rank in B, and thereby calculate its rank in A.

We choose $r = \lfloor N/2^{c+2} \rfloor$, and note that $r \geq 4$. The set A' together with its satellite information is stored in at most $N'(\lg M - \lg N' + \lg N - (c+2) + c) \leq N'(\lg M - 1)$ bits (recall that $\lg N \leq \lg N' + 1$). Adding in the $\lceil \lg N \rceil$ bits that represent r, this is still less than $N' \lceil \lg M \rceil$ bits, since $N \geq 4$. Thus, the overall data structure takes at most $N \lceil \lg M \rceil$ bits as desired. Since $|B| = \lceil N/r \rceil = O(2^c)$, the binary search on B takes $O(c) = O(1)$ time, and all other operations on this data structure also take $O(1)$ time. $\square$

Theorem 4.1 *Let $N, M \geq 1$ be integers, let $S \subseteq [M]$ be a set of size $n \leq N$. Suppose we have access to two functions $h_S, q_S : [M] \mapsto \mathbb{N}$, satisfying the following conditions:*

1. *h_S is 1-1 on S.*

2. *h_S and q_S can be evaluated in $O(1)$ time and from $h_S(x)$ and $q_S(x)$ one can uniquely reconstruct x in $O(1)$ time.*

3. *$||h_S||$ is $O(N^2)$ if $n > \sqrt{\lg N}$ and $||h_S||$ is $(\lg N)^{O(1)}$ otherwise.*

4. *$\lceil \lg ||h_S|| \rceil + \lceil \lg ||q_S|| \rceil$ is $\lg M + O(1)$.*

Then we can represent S using $n \lceil \lg M \rceil$ bits plus a precomputed table of size $o(\sqrt{N})$ bits that depends only upon $||h_S||$, if $n \leq \sqrt{\lg N}$. Assuming a word size of at least $\lg \max\{M, N\}$ bits, and that $||h_S||$ and $||q_S||$ are known, we can support rank in $O(1)$ time.

Proof: Let n_0 be a sufficiently large constant, to be determined later. Depending upon the value of n, we apply one of three approaches. If $n \leq n_0$, we explicitly write down the elements of S. If $n > \sqrt{\lg N}$, we represent $h_S(S)$ using a perfect hash function; since n is not too small, the space for the hash function can be seen to be $O(n)$ bits. Finally, in the intermediate range we represent $h_S(S)$ implicitly and operate upon it by table lookup (cf. the paragraph before the start of Section 4.1) In both the latter cases, additive $O(n)$ terms are hidden by using Lemma 4.1.

Case I ($n_0 \leq n < \sqrt{\lg N}$): We begin by showing that any $T \subseteq S$, $|T| \geq |S|/2$, can be stored along with satellite data from $[\ell]$ in at most $g_c(|T|, M, \ell)$ bits for some constant $c > 1$ and for any integer ℓ.

Let $T = \{x_1, \ldots, x_{n'}\}$. As h_S is 1–1 on T, let π be the permutation such that $y_i = h_S(x_{\pi(i)})$, for $i = 1, \ldots, n'$ and $y_1 < y_2 < \ldots < y_{n'}$. The data structure consists of the following three components:

1. An implicit representation of the set $T' = \{y_1, \ldots, y_{n'}\}$;

2. An array *sat* of size n' such that $sat[i]$ contains the satellite information associated with $x_{\pi(i)}$;

3. An array Q of size n' such that $Q[i] = q_S(x_{\pi(i)})$, i.e, $Q[i]$ is the quotient corresponding to y_i.

Given a query key x, we first find the index i such that $y_i = h_S(x)$, if such an i exists. Since the size of the representation of T' is $\left\lceil \lg \binom{||h_S||}{n'} \right\rceil = O(n' \lg ||h_S||) = o(\lg N)$, one can use this representation to index into a table of size $o(\sqrt{N})$ and accomplish this in $O(1)$ time. If no such i exists, then $x \notin T$. If such an i exists, then $x_{\pi(i)}$ is the only candidate for a match in T. Next, we reconstruct this key from y_i and $Q[i]$, and if it matches we return $sat[i]$.

The size of the representation of T is $\left\lceil \lg \binom{||h_S||}{n'} \right\rceil + n'(\lceil \lg ||q_S|| \rceil + \lceil \lg \ell \rceil)$; since $\lg \binom{||h_S||}{n'} = n'(\lg ||h_S|| - \lg n') + O(n')$, and $\lceil \lg ||h_S|| \rceil + \lceil \lg ||q_S|| \rceil$ is $\lg M + O(1)$ by assumption, the space used is $n'(\lg M - \lg n' + \lg \ell) + O(n')$, which is expressible as $g_c(|T|, M, \ell)$ bits for some constant $c > 0$, as required.

Now, by applying Lemma 4.1, we can represent S using $n \lceil \lg M \rceil$ bits and support *rank* queries in $O(1)$ time.

Case II ($n \geq \sqrt{\lg N}$): We again show that any $T \subseteq S$, $|T| \geq |S|/2$ can be stored along with satellite data from $[\ell]$ in at most $g_{c'}(|T|, M, \ell)$ bits for some constant $c' > 0$. Again, let $T = \{x_1, \ldots, x_{n'}\}$ and $T' = \{h_S(x) \mid x \in T\}$. Our approach is essentially as above, except that we use observations from [27] in place of item (1) in Case I to store T' (see Remark 2 after proof).

Recall that a function $f : [||h_S||] \rightarrow [n']$ is said to be *perfect* for T' if it is 1-1 on T'. Schmidt and Siegel [31] have shown that such a function f can be represented in $O(n' + \lg \lg ||h_S||) = O(n + \lg \lg N) = O(n)$ bits, and can be evaluated in $O(1)$ time for any argument in its range. For $i \in [n']$ we let $f^{-1}(i)$ denote the unique $y \in T'$ such that $f(y) = i$. A first cut at the data structure consists of:

1. A function f as above;

2. An array Y of size n' such that $Y[i]$ contains $f^{-1}(i)$;

3. An array *sat* of size n' such that *sat*$[i]$ contains the satellite information associated with $f^{-1}(i)$ and

4. An array Q of size n' such that $Q[i] = q_S(f^{-1}(i))$.

As observed by Pagh [27, Proposition 2.2] we can save space by storing not the full key $f^{-1}(i)$ in location $Y[i]$, but only the quotient information that distinguishes it from the part of $[||h_S||]$ that is mapped to i under f (Knuth [21, p. 525] attributes the general concept to Butler Lampson). By doing this, we can store Y in $n'(\lg ||h_S|| - \lg n') + O(n')$ bits and still support membership queries in T' in $O(1)$ time.

Given a query key x, we determine if $h_S(x) \in T'$ by calculating $i = f(h_S(x))$ and inspecting $Y[i]$. If so, we determine if $x \in T$ by inspecting $Q[i]$, and if so, we return *sat*$[i]$. The size of the representation of T is clearly $n'(\lg M - \lg n' + \lg \ell) + O(n') = g_{c'}(|T|, M, \ell)$ bits, as desired. Again, by Lemma 4.1, we can represent S using $n \lceil \lg M \rceil$ bits and support rank queries in $O(1)$ time.

CASE III ($n < n_0$): We write down the elements of S in sorted order, using $n \lceil \lg M \rceil$ bits. Note that n_0 can be chosen to be $2^{\max\{c,c'\}+3}$. $\square$

Remarks:

1. The function h_S is essentially a 'range reduction' commonly used in perfect hashing. Indeed, the first two steps of the FKS scheme [11] show the existence of the function h_S with range of $O(|S|^2)$ and $q_S(x)$ is simply the quotient information required to recover x given $h_S(x)$. However, for a small set S, the space to represent the function h_S can become dominant. The solution to this is to use the same function for several small sets. This is why h_S has a relaxed range of $O((\lg N)^c)$, rather than the minimum range of $O(\lg N)$. We get to these details in the next lemma.

2. The argument for Case II could be shorter if we could use [27, Theorem 6.1], and store the $q_S()$ values as additional satellite data. Unfortunately, the condition $s = m^{O(1)}$ at the start of [27, Section 6] may not hold in our case.

Lemma 4.2 *Let $N, M \geq 1$ be integers, and let $0 < i_1 < i_2 < \ldots < i_s < N$ be a sequence of integers. Let $S_{i_1}, S_{i_2}, \ldots, S_{i_s}$ all be subsets of $[M]$ such that $\sum_{j=1}^{s} |S_{i_j}| \leq N$. Then there exist functions $h_{S_{i_j}}$ and $q_{S_{i_j}}$ for $j = 1, \ldots, s$ that satisfy the conditions of Theorem 4.1, and that can be represented in $o(N) + O(\lg \lg M)$ bits.*

Proof: Let $n_{i_j} = |S_{i_j}|$ and let $S^* = \cup_{j=1}^{s} S_{i_j}$. We first define a function f, which is a 'global' range reduction.

If $M \leq N^2$ define f as $f(x) = x$. Otherwise, if $M > N^2$, then we find a hash function f given by $f(x) = (ax \bmod p) \bmod N^2$ for some prime $p \leq N^2 \lg M$ and $1 \leq a \leq p-1$ which maps S^* bijectively into the set $[N^2]$. The existence of such a function is guaranteed by FKS [11]. Let a^{-1} be the inverse of a modulo p, i.e. the unique integer $x, 1 \leq x \leq p-1$ such that $ax \bmod p = 1$. The function f, along with a^{-1} is represented using $O(\lg N + \lg \lg M)$ bits.

Note that choosing $h_{S_{i_j}} = f$ suffices if $n_{i_j} \geq \sqrt{\lg N}$, but if S_{i_j} is small, i.e., $n_{i_j} < \sqrt{\lg N}$ we need to reduce the range even further. We form $N/\lg N$ groups of sets, where the ℓth group consists of the sets $\{S_{i_j} | \ell \lceil \lg N \rceil + 1 \leq i_j < (\ell+1) \lceil \lg N \}\rceil$. Let S_ℓ be the union of all elements in the small sets in the ℓth group. For the ℓth group, find a prime p_ℓ such that the the function $g_\ell(x) = f(x) \bmod p_\ell$ is 1-1 on the set $f(S_\ell)$. Such a p_ℓ whose value is at most $O(|S_\ell|^2 \lg N)$ exists, since $||f||$ is $O(N^2)$ [11, 31]. Since $|S_\ell| \leq (\lg N)^{3/2}$ we can represent p_ℓ using $O(\lg \lg N)$ bits.

We store these primes, indexed by their group number in a separate table. Each prime is stored in a field of $b = \Theta(\lg\lg N)$ bits. If S_ℓ is empty (i.e. there is no small set in the ℓ-th group) then the table contains a string of b 0s in the entry corresponding to that group. The total space required by this table is $o(N)$ bits. For a small set S_{i_j} the function $h_{S_{i_j}}$ which is required by Theorem 4.1 is defined by $h_{S_{i_j}}(x) = g_\ell(f(x))$ if S_{i_j} belongs to the ℓ-th group.

Next we go on to describe the functions q_S required in Theorem 4.1. To recover the original element x from $g_\ell(x)$, we need to store the following 'quotient' value:

$$q_\ell(x) = \Big((x \text{ div } p)\left\lceil p/N^2\right\rceil + \Big((ax \bmod p) \text{ div } N^2\Big)\Big)\left\lceil N^2/p_\ell\right\rceil + f(x) \text{ div } p_\ell.$$

From $g_\ell(x)$, $q_\ell(x)$ and using p and p_ℓ (and a^{-1}, if needed) one can obtain x in constant time. Observe that $\lg||h_{S_{i_j}}|| + \lg||q_{S_{i_j}}|| \leq \lg M + 4$, which satisfies the hypothesis of Theorem 4.1. □

Theorem 4.2 *Let $S_1, S_2, \ldots, S_s$ all contained in $[k]$ be given sets with S_i containing d_i elements, such that $\sum_{i=1}^{s} d_i = n$. Then this collection of sets can be represented using $n\lceil\lg k\rceil + o(n) + O(\lg\lg k)$ bits, supporting $rank(x, S_i)$ operations in constant time. Here $rank(x, S_i)$ returns the rank of the element x in set S_i, if $x \in S_i$ and returns -1 otherwise. We assume that we have access to a constant time oracle which returns the starting position of the representation of each dictionary.*

Proof: This follows almost immediately from Theorem 4.1, applied with $N = n$ and $M = k$. The first thing to note is that there are only polylogarithmically many tables for operations on small sets, thus the space required by all tables put together is $o(n)$. The second thing is that in order to easily find the start of the representation of a set S_i, the representations of all sets S_i would need to be padded out to precisely $|S_i|\lceil\lg k\rceil$ bits if necessary. □

4.2 Putting things together

To summarize, the construction of the representation of a k-ary cardinal tree is a two-stage process; see Figure 3 for an example. First we store the representation of the ordinal tree (the cardinal tree without the labels), and any auxiliary structures required for those operations, in $2n + o(n)$ bits. Next, to facilitate an easy mapping from the ordinal representation to the cardinal information, we traverse the tree in depth-first order (as in creating the ordinal-tree representation) and store the child information in a separate array. Besides a global three words, each of $\lceil 2\lg n + \lg\lg k\rceil$ bits (used for global range reduction), each child-information structure is written using $d\lceil\lg k\rceil$

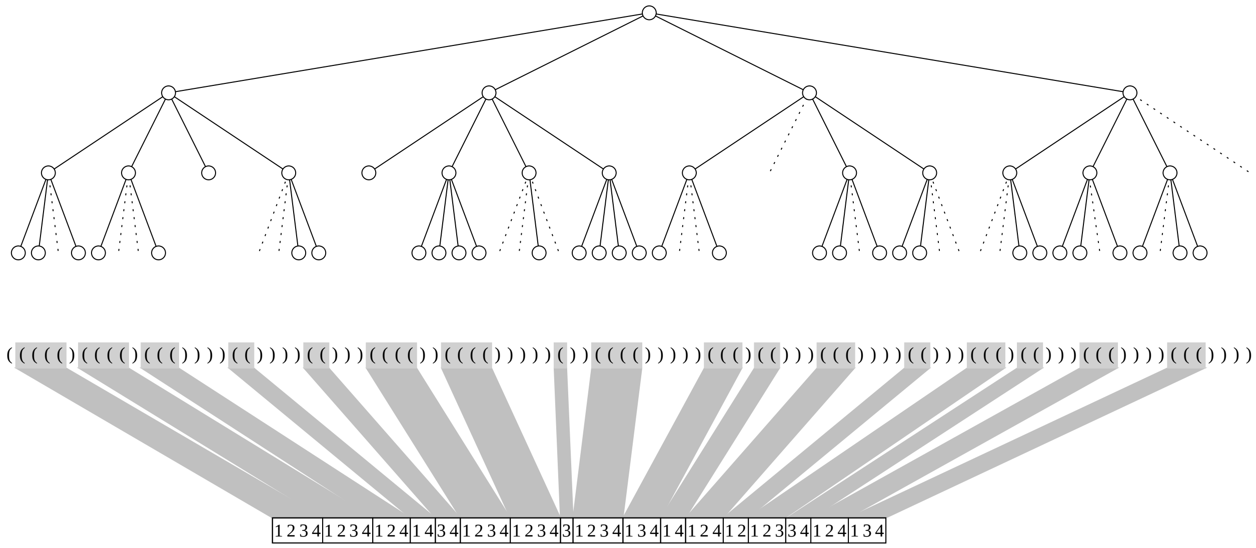


Figure 3: Our cardinal tree encoding of the tree in Fig. 2

bits (using Theorem 4.2), and if less space than that is required, the structure is padded to fill the entire $d \lceil \lg k \rceil$ bits. To find the child-information structure for a node starting at position p in the ordinal tree encoding, we compute the number of open parentheses strictly before position p, $rank_{open}(p-1)$, and look at position $rank_{open}(p-1) \lceil \lg k \rceil$ in the child-information array. Thus we have:

Theorem 4.3 *There is an $n(\lceil \lg k \rceil + 2) + o(n) + O(\lg \lg k)$ bit representation of a k-ary cardinal tree on n nodes that provides a mapping from the nodes of the tree to $\{1, \ldots, n\}$ and supports the operations of finding the parent of a node, the child with label j and the size of the subtree rooted at any node, in constant time.*

Compared with the lower bound of approximately $\lg k + \lg e$ bits per node, there is a difference of $0.557305 + o(1)$ bits per node plus the effects of the ceiling on the $\lg k$ term.

5 Representations for Fixed k

The above structures are intended for situations in which k is very large and not viewed as a constant. In most applications of cardinal trees (e.g., B-trees or tries over the Latin alphabet), k is given a priori. It is a matter of 'data structure engineering' to decide what aspects of our solution for 'asymptotically large' k are appropriate when k is 256 or 4. While it may be a matter of debate

1111111 1111111 111110 00 00 00 1100 00 00 00 10000 00 00 1111111 00 1111111 00 00 00 00 011
00 1111111 00 00 00 00 111101 1100 00 00 111110 00 00 00 11101 00 00 1111110 10000 00 00
111110 00 00 00 111101 00 00 00—158 bits.

Figure 4: Prefix Encoding of the Tree in Figure 2.

as to the functional relationship between 256 and k, it is generally accepted that 4, the cardinality of the alphabet describing genetic codes, is a (reasonably) small constant.

A naïve encoding of a cardinal tree of degree 4 is to represent each node as three nodes in a binary tree, resulting in an encoding that uses $6n + o(n)$ bits. Theorem 4.3 improves this bound to $4n + o(n)$ bits. Up to a ceiling and $o(n)$ term, this matches the information theoretic lower bound of $(4 \lg 4 - 3 \lg 3)n = 3.24511\ldots$ bits. Simplifying and tuning our prior discussions, we show how to obtain a more succinct encoding:

Corollary 5.1 *There exists a* $\left(3 + \frac{5}{12}\right) n + o(n) = 3.41667\ldots n + o(n)$ *bit representation of a* 4*-ary cardinal tree on* n *nodes that provides a mapping from the nodes of the tree to* $\{1, \ldots, n\}$ *and supports the operations of finding the parent of a node, the child of a node with label* j*, and the size of the subtree rooted at any node, all in constant time.*

The basic idea is that some types of cardinal nodes are more common than other types in a single tree. More precisely, there are 2^k *types* of cardinal nodes, represented by a characteristic vector of which children are present, but trees necessarily bias towards certain node types. For example, assuming for the moment that there are no degree-one nodes, half of the nodes are leaves, and hence half the nodes have type $00 \cdots 0$. Thus if we use fewer bits to represent leaves than other nodes, we should be able to achieve an overall improved space bound.

To achieve this goal, we use a *prefix code* to encode nodes, that is, an assignment of bit-strings to node types such that no bit-string is a prefix of another. The particular prefix code for $k = 4$ is shown in Table 1.

The overall organization of the data structure is as follows. We first store a bit-string obtained by concatenating the prefix codes of the nodes of the given 4-ary cardinal tree in depth-first order of the tree (see Fig. 4 for an example). This requires at most $(3 + 5/12)n$ bits. We augment this bit-string with $o(n)$ additional bits to allow the bit-string to act as an array, i.e., to permit access to the ith element (prefix code) in constant time. This is done by combining Jacobson's technique for building an index into Huffman coded files [18] with the ideas used in improvong the

Node Degree	Node	Code(s)	Bits per Node
0 (leaf)	0000	00	
1	0001	010	
	0010	011	$3\frac{1}{4}$
	0100	101	
	1000	1001	
2	0011	10000	
	0101	10001	
	1001	1100	$\frac{4\frac{2}{3}+1\cdot 2}{2} = 3\frac{1}{3}$
	0110	1101	
	1010	11100	
	1100	11101	
3	0111	111100	
	1011	111101	$\frac{6\frac{1}{4}+2\cdot 2}{3} = 3\frac{5}{12}$
	1101	111110	
	1110	1111110	
4	1111	1111111	$\frac{7+3\cdot 2}{4} = 3\frac{1}{4}$

Table 1: Prefix Code for 4-ary Trees. The bits per node are calculated by including the cost of representing any leaves in the subtree of the node, using the fact that $d-1$ leaves can be associated to a node with degree d in a one-to-one manner. Note that these are set-to-set mappings of node arrangements to codes; we permute the encodings so that the shorter code is chosen for more frequently occurring nodes of a given degree.

time complexity of the select structure [6].

Let D be the balanced parenthesis sequence obtained by writing the degrees of nodes in unary in the depth-first order of the tree, as in Section 3. We store the $o(n)$ bit auxiliary structures described in Section 3 that are used to support all the navigational operations on the tree, without storing the sequence D itself. In order to run in $O(1)$ time, the navigational operations need, in addition to these auxiliary structures, access to a constant number of segments of $O(\lg n)$ bits from D. Thus, if we can reconstruct a segment of D of length $O(\lg n)$ starting at a given position in constant time, then we can support all the operations supported by the DFUDS representation in $O(1)$ time as well. Together with the prefix code sequence, we can support the operation of finding a child with a given label (this also follows from the fact that the maximum degree of a node is 4). Thus, we now describe how to support the operation of finding a segment of D of length $O(\lg n)$ starting at a given position in constant time, using the prefix code sequence. For this purpose, we store the following auxiliary structures:

- an array A of size $O(n/(\lg n)^2)$ that, for every position $i \leq n$ in D which is a multiple of $(\lg n)^2$, stores: (a) the index j of the node, in the depth-first order of the tree, to which the ith parenthesis in D corresponds (i.e., it appears in the unary degree representation of that node), and (b) the offset of the position i from the beginning of the representation of the corresponding node (which is a number between 0 and 4).

- an array B of size $O(n/\lg n)$ that, for every position $i \leq n$ in D which is a multiple of $(\lg n)/2$, stores: (a) the value $j \bmod (\lg n)^2$ where j is the index of the node that corresponds to the ith parenthesis in D, and (b) the offset of the position i from the beginning of the representation of the corresponding node.

- a $o(n)$ bit precomputed table that, for every possible sequence of $\varepsilon \lg n$ prefix codes (for some $\varepsilon < 1/8$, as the longest prefix code is of length 7), stores the parenthesis sequence corresponding to this, and also its length.

Now, to find a segment of length $O(\lg n)$ starting at position i in D, we first find, using A, an index j and a value $i' \leq (\lg n)^2$ such that the index j corresponds to the position $\lfloor i/(\lg n)^2 \rfloor (\lg n)^2$ in D, and $(i - i')$ is the number of parentheses in the prefix codes 1 to $j - 1$. Again, using B, we find another index j' and a value $i'' \leq (\lg n)/2$ such that the index j' corresponds to the position $\lfloor 2i/\lg n \rfloor (\lg n)/2$ in D, and $(i - i'')$ is the number of number of parentheses in the prefix codes from 1 to j'. Now, by doing a linear scan from the j'th prefix code (in the prefix code sequence) and using the precomputed table, we can output the required sequence in constant time. Here we use the fact that we can read (write) prefix codes (unary degree sequences) of $O(\lg n)$ nodes in $O(1)$ time.

6 Conclusion

We have given succinct representations for ordinal and cardinal trees that require space within a lower-order term of the information theoretic lower bound in many cases, and are always within a additive linear number of bits of the lower bound. Our representations support all basic navigational operations in constant time, and also support the subtree size operation in constant time. En route we gave a representation of a static dictionary that supports membership and rank queries of present elements in constant time.

References

[1] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 295–304, Atlanta, Georgia, May 1999.

[2] David A. Benoit. Compact tree representations. MMath thesis, University of Waterloo, 1998.

[3] David A. Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In F. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 169–180, Vancouver, Canada, August 1999.

[4] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[5] Y.-T. Chiang, C.-C. Lin, and Hsueh-I Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 506–515, January 2001.

[6] David Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[7] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, Atlanta, Georgia, January 1996.

[8] John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a compact representation of trees. *Software—Practice and Experience*, 23(3):277–291, March 1993.

[9] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM Journal on Computing*, 22(1):1–10, February 1993.

[10] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.

[11] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[12] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

[13] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994.

[14] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice Hall, 1992.

[15] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*, Addison-Wesley, 1989.

[16] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Research Triangle Park, North Carolina, October–November 1989.

[17] Guy Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, 1989.

[18] Guy Jacobson and Vo Kiem-Phong. Heaviest Increasing/Common Subsequence Problems. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, Udi Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 52–66, Tucson, Arizona, USA, April–May 1992.

[19] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[20] Donald E. Knuth. *The Art of Computer Programming vol. 1: Fundamental Algorithms, 2nd ed.* Addison-Wesley, 1974.

[21] Donald E. Knuth. *The Art of Computer Programming vol. 3: Sorting and Searching, 2nd ed.* Addison-Wesley, 1998.

[22] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

[23] J. Ian Munro. Tables. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42, Hyderabad, India, December 1996.

[24] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. Preliminary version in *Proceedings of the 38th Annual IEEE Computer Society Conference on Foundations of Computer Science*, pages 118–126, Miami Beach, Florida, October 1997.

[25] J. Ian Munro, Venkatesh Raman and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[26] J. Ian Munro, Venkatesh Raman and Adam Storm. Representing dynamic binary trees succinctly. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 529–536, January 2001.

[27] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[28] Rajeev Raman, Venkatesh Raman and S. Srinivasa Rao. Succinct dynamic data structures. In F. Dehne, J.-R. Sack, R. Tamassia, editors, *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437, Providence, RI, USA, August 2001.

[29] Rajeev Raman, Venkatesh Raman and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pp. 233–242, 2002.

[30] Venkatesh Raman and S. Srinivasa Rao. Static dictionaries supporting rank. In A. Aggarwal and C. Pandu Rangan, editors, *Proceedings of the 10th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pages 18–26, Chennai, India, December 1999.

[31] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.