Representing Trees of Higher Degree

David Benoit[1,2], Erik D. Demaine[2], J. Ian Munro[2], and Venkatesh Raman[3]

[1] InfoInteractive Inc., Suite 604, 1550 Bedford Hwy., Bedford, N.S. B4A 1E6, Canada
[2] Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {dabenoit, eddemaine, imunro}@uwaterloo.ca
[3] Institute of Mathematical Sciences, Chennai, India 600 113, vraman@imsc.ernet.in

Abstract. This paper focuses on space efficient representations of 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 (rooted trees where each node has k positions each of which may have a reference to a child) and ordinal trees (the children of each node are simply ordered). Our representations use a number of bits within a lower order term of the information theoretic lower bound. For cardinal trees the structure supports finding the parent, child i or subtree size of a given node. For ordinal trees we support the operations of finding the degree, parent, ith child and subtree size. These operations provide a mapping from the n nodes of the tree onto the integers $[1, n]$ and all are performed in constant time, except finding child i in cardinal trees. For k-ary cardinal trees, this operation takes $O(\lg \lg k)$ time for the worst relationship between k and n, and constant time if k is much less than n.

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 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 (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 [10, 13] of simply storing an array of references to positions in the text rather than the valuable structure of the tree.

This and many others applications deal with large static trees. The representation of a tree is required to provide a mapping between the n nodes and the integers $[1, n]$. Whatever information need be stored for the application (e.g., the location of a word in the database) is found through this mapping. This application used binary trees, though trees of higher degree, for example degree 256 for text or 4 for genetic sequences, might be better. Trees of higher degree, i.e. greater than 2, are the focus of this paper.

Starting with Jacobson [11, 12] 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 base 2 (denoted lg) of the number of objects in the class. The number of binary trees on n nodes is $C_n \equiv \binom{2n+1}{n}/(2n+1)$; $\lg C_n \approx 2n$. 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 [9] 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 [3, 4] 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 tries 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 [15] 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.

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 mapping between these trees and binary trees is a well known undergraduate example, and so about $2n$ bits are necessary for representation of such a tree. Jacobson [11, 12] 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 [4] 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 [15] implement node degree, parent and subtree size in constant time, but take $\Theta(i)$ time to find the ith child. 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. A binary tree is a cardinal tree of degree 2. There are $C_n^k \equiv \binom{kn+1}{n}/(kn+1)$ such trees, so $(\lg(k-1) + k\lg(k/(k-1)))n$ bits is a good estimate of the lower bound on a representation, assuming n is large with respect to k. This bound is roughly $(\lg k + \lg e)n$ bits as k grows. Our work is the first, of which we know, to seriously explore succinct representations of cardinal trees of degree greater than 2. Our techniques answer queries asking for parent and subtree size in constant time, and find the ith child in $O(\lg\lg k)$ time, and better in most reasonable cases. The structure requires $(\lceil\lg k\rceil + 2)n + o(n)$ bits. This can be written to more closely resemble the lower bound as $(\lceil\lg k\rceil + \lceil\lg e\rceil)n + o(n)$ bits.

The rest of this paper is organized as follows. Section 2 describes previous encodings of ordinal trees. These two 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.

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 [11] 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 2.2), *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 spread throughout the encoding.

2.2 Rank and Select

The operations performed on Jacobson's tree representation require the use of two auxiliary structures: the rank and select structures [3, 4, 12, 14]. These structures support the following operations, which are used extensively, either directly or implicitly, in all subsequent work, including this paper:

Rank: $rank(j)$ returns the number of **1**s up to and including position j in an n bit string. An auxiliary $o(n)$ bit structure [12] supports this operation in constant time. $rank_0(j)$ is the analogous function counting the **0**s.

Select: $select(j)$ returns the position of the jth **1**. It also requires an auxiliary $o(n)$ bit structure [12]. Jacobson's method takes more than constant time, but inspects only $O(\lg n)$ bits. The modification of Clark and Munro [3, 4] reduces this to $\Theta(1)$ time. $select_0(j)$ is the analogous function locating a **0**.

The auxiliary structures for rank [3, 12] 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 subblocks of length $\lceil\lg\lg n\rceil$. Keep a table containing the number of **1**s within the block up to the last position in each subblock.
- Keep a table giving the number of **1**s up to every possible position in every possible distinct subblock.

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.

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, $select_1(rank_0(p)+1)$ turns out to find the **1** in the description of the parent that corresponds to this child. Thus, searching backwards to the previous zero (using $rank_1$ then $select_1$) 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 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.

2.3 The Balanced Parentheses of Munro and Raman

The binary tree encoding of Munro and Raman [15] 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, so equate: $rank_{open}(j) \equiv rank(j)$, $rank_{close}(j) \equiv rank_0(j)$, $select_{open}(j) \equiv select(j)$ and $select_{close}(j) \equiv select_0(j)$. The following operations, defined on strings of balanced parentheses, can be performed in constant time [15]:

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

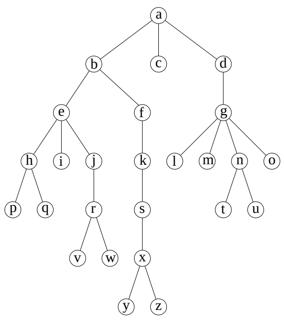
$findopen(i)$: find the position of the open parenthesis that matches the closing parenthesis in position i.

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

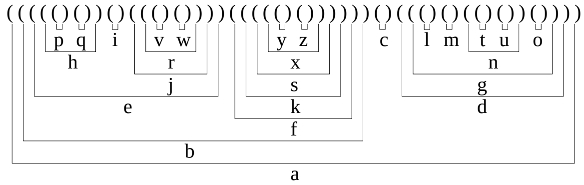
$enclose(i)$: given a parenthesis pair whose open parenthesis is in position i, find its closest enclosing matching parenthesis pair.

**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**

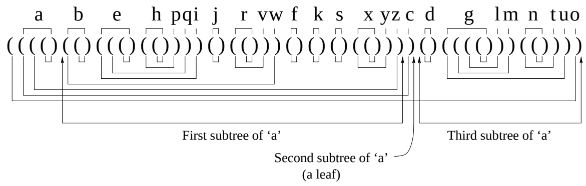
(b) Jacobson's *LOUDS* Representation (without the superroot). The commas have been added to aid the reader.



(a) The Ordinal Tree



(c) Munro and Raman's Balanced Parentheses Representation



(d) Our *DFUDS* Representation

Fig. 1. Three Ordinal Encodings of an Ordinal Tree

double_enclose(x, y): given two nonoverlapping parenthesis pairs whose open parentheses appear in positions x and y respectively, find a parenthesis pair, if one exists, that most tightly encloses both parenthesis pairs. (This is equivalent to finding the least common ancestor in the analogous tree.)

open_of(p) & *close_of*(p): select either the open or close parenthesis of the parenthesis pair, p, returned by *enclose*(i) or *double_enclose*(x, y).

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. This 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. Using the $findopen(i)$ and $findclose(i)$ operations, one can give 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 (described in [15]). 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 level-order 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 2.3. An example of our encoding is given in Fig. 1(d).

Theorem 1. *The DFUDS representation produces a string of balanced parentheses on all non-null trees.*

Proof. The construction's validity follows by induction and the observations:

- If the root has no children, then the representation is '**()**'.
- Assume that the method produces p strings, $R_1, R_2, \ldots, R_p$, of balanced parentheses for p different trees. We must prove that the method will produce a string of balanced parentheses 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. 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. □

3.1 Operations

The following subsections detail how the navigation operations are performed on this representation. They lead to our main result for ordinal trees.

Theorem 2. *There is a $2n + o(n)$ bit representation of an n node ordinal tree, that provides a mapping between the nodes of the tree and the integers in $[1, n]$ and permits finding the degree, parent, ith child and subtree size in constant time.*

Finding the ith Child of a Node. This includes the operation of finding the degree of a node. From the beginning of the description of the current node:

- Find the degree d of the current node, by finding the number of opening parentheses that are listed before the next closing parenthesis (using $rank_{close}$ and $select_{close}$). If $i > d$, 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.

Alternatively, assuming that the node has at least i children, the description of the ith child is a fully parenthesized expression beginning at position

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

Finding the Parent of a Node. 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 (if there are none, the parent is the root of the tree), and the description of the parent node is after this closing parenthesis (or the opening parenthesis in the case of the root node).

Alternatively, the description of the parent of a node is a fully parenthesized expression beginning at position

$$select_{close}(rank_{close}(findopen(position - 1))) + 1. \tag{2}$$

Note 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 equivalent to the number of open parentheses in the string of balanced parentheses that describes the subtree. 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.

Alternatively, the size of the subtree at the current position is

$$rank_{open}(close\_of(enclose(position))) - rank_{open}(position). \tag{3}$$

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

$$\lfloor (close\_of(enclose(position)) - position)\,/2 \rfloor . \tag{4}$$

Next we use this data structure to implement 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 [11, 12]. The modified encoding for k-ary trees simply encodes a node by k bits, where the ith 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.

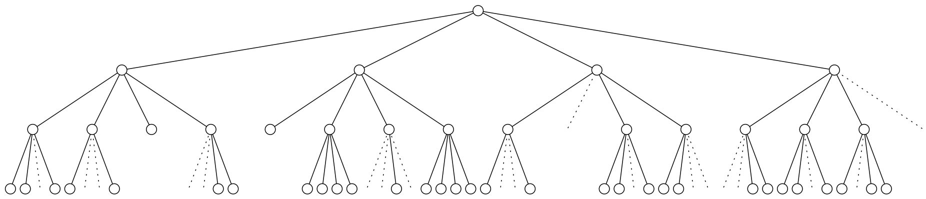


Fig. 2. Generalized Jacobson Encoding of a 4-ary Tree: 1111 1111 1111 1011 1110 1101 1001 0000 0011 0000 1111 0010 1111 1001 1101 1100 0011 1101 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000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

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., $(\lceil \lg k \rceil + \lceil \lg e \rceil)n + o(n)$ 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, $d\lceil \lg k \rceil$ bits to encode *which* children are present, where d is the number of children at that node.

If k is very small with respect to n ($k \leq (1-\varepsilon)\lg n$), we use the $o(n)$ term in the storage bound to write down the bitmap encoding for every type of node, because there are only 2^k types. The space allocated to each node ($d\lceil \lg k \rceil$ bits) can be used to store an index to the appropriate type of node. As the ordinal tree representation gives us the node degree, call it d, we only need $d\lceil \lg k \rceil$ bits to distinguish among almost k^d types with d children present.

On the other hand, even when k is a bit larger, we can reduce the decoding time (from a more efficient description) to a constant using the "powerful" $\lg n$ bit words. This idea is the topic of the remainder of the section. Our result is related to, but more precise in terms of model, than that of [5].

4.1 High Level Structure Description

The ordinal tree representation in Section 3 gives, in constant time, three of the four major operations we wish to perform on cardinal trees: subtree size, ith child, and parent. 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 question can be answered 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 label. In this way, the cardinal tree operations reduce to ordinal tree operations. It remains to show how to encode the child information at each node. For simplicity we will assume that k is at most n. If k is greater than n, the same results follow immediately from our proofs provided we use the (natural) model of $\lceil \lg k \rceil$ bit words.

4.2 Different Structures for Different Ranges

For each node, we know from the ordinal tree encoding the number of present children, d. If d is small, say one or two, we would list the children. If it is large, however, a structure such as a bit-vector is more appropriate. In this section we handle the details of this process.

In the case of a very small number of children (a *sparse* set), we are willing to simply list the children that are present. The upper bound on the number of items we choose to explicitly list is $\lg k$, meaning if a binary search is performed on this list, it will take at most $\lg\lg k$ time to answer membership queries. For reasonable values of k, say smaller than the number of particles in the universe, $\lg\lg k$ is a single digit number. Nevertheless, in the interests of stating a more general result, we consider packing several $\lceil \lg k \rceil$ bit indicators into a single $\lceil \lg n \rceil$ bit word. An auxiliary table of $o(n)$ bits can be used to detect whether, and if so where, a suitably aligned given $\lceil \lg k \rceil$ bits occur. This can be used either to scan the representation of the children that are present, or to replace a binary search with a search that branches in one of $\lfloor 1 + \lg n / \lg k \rfloor$ ways. As a consequence, the search cost is $O(\lg\lg k/(\lg\lg n - \lg\lg k))$. This search cost is constant, provided k is reasonably smaller than n, namely $k \leq 2^{(\lg n)^{1-\varepsilon}}$ for some fixed $\varepsilon > 0$.

Moderate sets, those with anywhere between $\lg k$ and k children, are trickier and the topic of the next section.

4.3 Representing Moderate Sets

This section deals primarily with storing a set of elements to facilitate fast searches while requiring near minimal space. This is a heavily studied area, but we have the additional requirement that if an element is found, its rank must also be returned.

To solve this problem we will use the method of Brodnik and Munro [2]. The method, for sets with the sparseness we are interested in, is as follows:

- Split the universe, k, into p buckets of equal size (the parameter p will be chosen later to optimize space usage).
- The values that fall into each bucket are stored using a perfect hash function [6–8].

To access the values stored in the buckets, we must index directly into the bucket for a given value. We do this by storing an array of pointers to buckets, so that given any value x, we search for it in bucket $\lfloor x/(k/p) \rfloor$. Each of these pointers occupies $\lceil \lg d \rceil$ bits, making the index to the buckets of size $p \lceil \lg d \rceil$ bits.

Because each of the subranges is of the same size, the space required to store each of the elements is $\lg(k/p)$ bits, giving a total of $d \lg(k/p)$ bits for the entire set. All that is left to describe is the hash function for each bucket. Brodnik and Munro refer to the method described by Fiat *et al.* [7] bounding the space needed to store a perfect hash function for bucket i by $6 \lg d_i + 3 \lg\lg(k/p) + O(1)$ where d_i is the size of the ith bucket. Thus the total space needed to store the hash functions for all the p buckets is $3p \lg\lg(k/p) + \sum_{i=1}^{p} 6 \lg d_i + O(p)$ which is at most $3p \lg\lg(k/p) + 6p \lg(d/p) + O(p)$. The structure of Fiat *et al.* does not support ranks. To support ranks, we store in a separate array the rank of the first element of each bucket. In addition, we create an array that for each bucket stores the rank of each element within its bucket. The extra space for these arrays is $p\lceil\lg d\rceil + \sum_{i=1}^{p} d_i \lg d_i$ which is at most $p\lceil\lg d\rceil + d\lg(d/p)$. We also need to know the d_i's, in order to look at the appropriate $\lg d_i$ bits to find the inner rank. But d_i is simply the difference between the rank of the first element of the ith bucket and the first element of the $(i+1)$st bucket.

Finally, in the rank structure we also need to know the beginning of the information for each bucket. The rank structure takes at most $d\lg(d/p)$ bits. So keeping a pointer to the first location describing the rank information of every bucket takes another $p\lg(d\lg(d/p))$ bits. Thus the overall space requirement is at most $3p\lceil\lg d\rceil + d\lg k - d\lg p + 3p\lg\lg(k/p) + 6p\lg(d/p) + O(p) + d\lg(d/p) + p\lg\lg(d/p)$. By choosing $p = d/c$ for a large constant c and using the fact that $d > \lg k$, we have that the space requirement for this range is at most $d\lg k$.

4.4 Representation Construction & Details

Construction of the representation of a k-ary cardinal tree is a two stage process. First we store the representation of the ordinal tree (the cardinal tree without the labels), and any auxiliary structures required for the 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 representation for the set of children at each node encountered. If we encounter a moderately sparse node, we also store the hash functions in an auxiliary table (separate from everything else). Each structure is written using $d\lceil\lg k\rceil$ bits, and if less than that is required, the structure is padded to fill the entire $d\lceil\lg k\rceil$ bits.

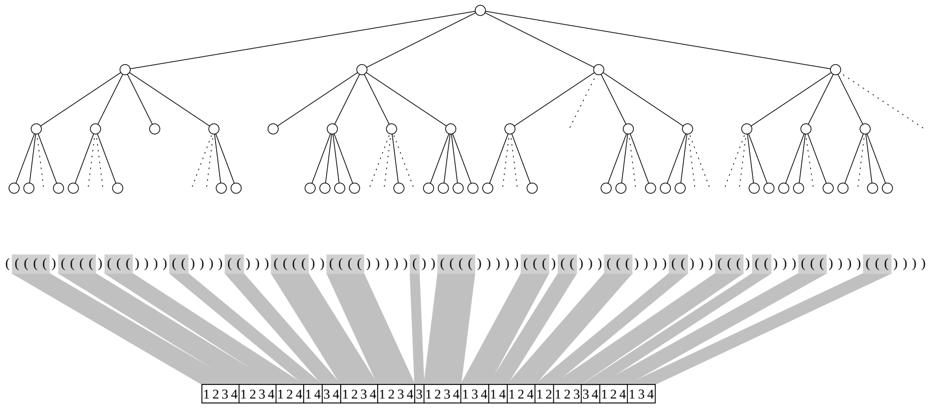


Fig. 3. Our Cardinal Tree Encoding of the Tree in Fig. 2

Navigation. Once we have stored the tree (an example of which is given in Fig. 3), we need to be able to navigate efficiently through the tree. Finding the parent of a node and the subtree size can both be performed using only the ordinal tree structure, but finding the child with label j of a node q requires the cardinal information. Now we outline the search procedure and its correctness:

- Let p be the beginning position of the representation of node q.
- Set bp to the number of open parentheses strictly before position p, i.e., $rank_{open}(p-1)$.
- Go to position $bp\lceil \lg k \rceil$ of the array storing the cardinal information, to get to the representation of the children of q.
- From k and d (the degree of node q which can be determined using an ordinal operation), we know which type of representation is stored in the the next $d\lceil \lg k \rceil$ bits and we can use the appropriate search for child j. If j is present, this search will return its rank within the set of q's children.
- This rank, r, can then be used to navigate down the tree using the ordinal tree operation "rth child."

Note that the time to execute this procedure is dominated by the cost of searching through the cardinal information, as described in Section 4.2.

4.5 Results

This representation takes $\lceil \lg k \rceil + 2 + o(1)$ bits per node, where the $o(1)$ term is for the structures needed by the ordinal tree (Section 3) as well as the space used to store the hash functions (Section 4.3). Compared with the lower bound of approximately $\lg k + \lg e$ bits per node, there is a difference of $0.5573 + o(1)$ bits per node plus the effects of the ceiling on the $\lg k$ term. This ceiling could be virtually eliminated by using the Chinese Remainder Theorem, although the modification would greatly increase the complexity of decoding the information.

Theorem 3. *There exists a $(\lceil \lg k \rceil + 2)\, n + o(n)$ bit representation of a k-ary cardinal tree on n nodes that provides a mapping between the nodes of the tree and the integers in $[1, n]$ and supports the operations of finding the parent of a node or the size of the subtree rooted at any node in constant time and supports finding the child with label j in $\lg \lg k$ time. More generally finding the child with label j can be supported in time $O(\lg \lg k / (\lg \lg n - \lg \lg k))$, which is constant if $k \leq 2^{(\lg n)^{1-\varepsilon}}$ for some fixed $\varepsilon > 0$.*

Proof. The ordinal tree component of our representation uses $2n + o(n)$ bits and supports, in constant time, finding the parent of any node and finding the size of the subtree rooted at any node (Section 3). To find the child with label j, we require the cardinal information, stored in $n\lceil \lg k \rceil$ bits, a table indexing the perfect hash functions, which uses $o(n)$ bits (Section 4.3) and the algorithm described in Section 4.4. The total space is therefore $(2 + \lceil \lg k \rceil)n + o(n)$. □

This structure is intended for a situation 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 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. Simplifying and tuning our prior discussions [1] we can obtain a more succinct encoding of cardinal trees of degree 4:

Corollary 1. *There exists a* $\left(3+\frac{5}{12}\right)n+o(n)$ *bit representation of* 4*-ary trees on* n *nodes, that supports, in constant time, finding the parent of a node, the size of the subtree rooted at any node and the child with label* j*.*

We note that this is substantially better that the naive approach of representing each node in the tree of degree 4 by a node and two children in a binary tree. The latter would require $6n+o(n)$ bits.

Acknowledgment: The last author would like to acknowledge with thanks the useful discussion he had with S. Srinivasa Rao.

References

1. David Benoit. *Compact Tree Representations.* MMath thesis, U. Waterloo, 1998.
2. Andrej Brodnik and J. Ian Munro. Membership in constant time and almost minimum space. *SIAM J. Computing*, to appear.
3. David Clark. *Compact Pat Trees.* PhD thesis, U. Waterloo, 1996.
4. David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 383–391, 1996.
5. John J. Darragh, John G. Cleary, and Ian H. Whitten. Bonsai: a compact representation of trees. *Software—Practice and Experience*, 23(3):277–291, March 1993.
6. A. Fiat and M. Naor. Implicit $O(1)$ probe search. *SIAM J. Computing*, 22(1):1–10, January 1993.
7. A. Fiat, M. Naor, J.P. Schmidt, and A. Siegel. Nonoblivious hashing. *J. ACM*, 39(4):764–782, April 1992.
8. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.
9. Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Computer and System Sciences*, 47(3):424–436, 1993.
10. G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indicies for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures & Algorithms*, 66–82, Prentice Hall, 1992.
11. Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science*, 549–554, 1989.
12. Guy Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU, 1989.
13. Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.
14. J. Ian Munro. Tables. In *Proc. 16th Conf. on the Foundations of Software Technology and Theoretical Computer Science*, *LNCS* vol. 1180, 37–42, Springer, 1996.
15. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, 118–126, 1997.