Engineering the LOUDS Succinct Tree Representation*

O'Neil Delpratt, Naila Rahman, and Rajeev Raman

Department of Computer Science, University of Leicester,
Leicester LE1 7RH, UK
{ond1, naila, r.raman}@mcs.le.ac.uk

Abstract. Ordinal trees are arbitrary rooted trees where the children of each node are ordered. We consider *succinct*, or highly space-efficient, representations of (static) ordinal trees with n nodes that use $2n + o(n)$ bits of space to represent ordinal trees. There are a number of such representations: each supports a different set of tree operations in $O(1)$ time on the RAM model.

In this paper we focus on the practical performance the fundamental Level-Order Unary Degree Sequence (LOUDS) representation [Jacobson, *Proc. 30th FOCS*, 549–554, 1989]. Due to its conceptual simplicity, LOUDS would appear to be a representation with good practical performance. A tree can also be represented succinctly as a balanced parenthesis sequence [Munro and Raman, *SIAM J. Comput.* **31** (2001), 762–776; Jacobson, *op. cit.*; Geary et al. *Proc. 15th CPM Symp.*, LNCS 3109, pp. 159–172, 2004]. In essence, the two representations are complementary, and have only the basic navigational operations in common (parent, first-child, last-child, prev-sibling, next-sibling).

Unfortunately, a naive implementation of LOUDS is not competitive with the parenthesis implementation of Geary et al. on the common set of operations. We propose variants of LOUDS, of which one, called *LOUDS++*, is competitive with the parenthesis representation. A motivation is the succinct representation of large static XML documents, and our tests involve traversing XML documents in various canonical orders.

1 Introduction

Ordinal trees are arbitrary rooted trees where the children of each node are ordered. We consider *succinct*, or highly space-efficient, representations of (static) ordinal trees with n nodes. An information-theoretically optimal representation of such trees would require $2n - O(\log n)$ bits. There are a number of representations that use $2n + o(n)$ bits of space, and support various navigational and other operations in $O(1)$ time on the RAM model of computation [6, 1, 5, 9].

This paper compares the practical performance of the fundamental *level-order unary degree sequence* succinct representation (hereafter LOUDS) [6] with non-succinct ordinal tree representations, as well as the *parenthesis* succinct representation (hereafter PAREN) [9], which supports a complementary set of operations to LOUDS. In practice, one must consider the lower-order terms in the space

* Delpratt is supported by PPARC e-Science Studentship PPA/S/E/2003/03749.

C. Álvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 134–145, 2006.

bound, which come from augmenting a bit-string of $2n + O(1)$ bits representing the tree with a number of *directories*, or auxiliary data structures, that are used to support operations in $O(1)$ time. The space used by each directory is, of course, asymptotically $o(n)$ bits, but is usually a function like $\Theta(n \log\log n / \log n)$ (and sometimes worse). For this and other reasons, the directories can use much more space than the representation of the tree, for practical values of n.

With this in mind, we consider the operations supported by the two succinct representations above, assuming a 'minimal' set of directories. Both support the basic navigational operations of parent, first-child, last-child, prev-sibling and next-sibling. However, LOUDS supports additional $O(1)$-time operations such as degree(x) (reporting the number of children of x), childrank(x) (the position of x among the children of its parent), child(x, i) (reporting the i-th child of x — recall that ordinal trees can have unbounded degree) and can enumerate all nodes at the same depth as a given node x, in time proportional to the number of such nodes. PAREN, on the other hand, readily supports operations such as desc(x) (report the number of children descended from x). LOUDS essentially numbers the nodes of the tree with integers in a level-order (breadth-first) numbering, while PAREN uses a depth-first (pre- or post- order) numbering.

The functionality of these 'minimal' representations can be expanded at negligible *asymptotic* cost. For example, PAREN can support degree(x) in $O(1)$ time by augmenting it with additional $o(n)$-bit directories [2], or level-ancestor queries in $O(1)$ time using yet another directory [10]. The *depth-first unary degree sequence* representation [1] comes close to being a 'union' of LOUDS and PAREN. However, its directories are also an (almost disjoint) union of the directories of both LOUDS and PAREN. Quite apart from the fact that none of these augmented data structures subsumes each other, adding additional directories may lead to poor practical performance. As noted above, directories consume significant space in practice. Different directories may have different memory access patterns, and adding additional ones can make it difficult to organise data to minimise cache misses. This motivates the study of alternative 'minimal' tree representations, such as LOUDS and PAREN, so that the one that best suits an application may be chosen, rather than a single 'universal' representation.

Although we defer a complete description of LOUDS to Section 3, we give a brief overview, in order to summarise the main issues and contributions. We first explain the task which we use to evaluate the data structures (the motivation is in the sub-section on XML below). We store, along with the tree, an array of size n, which stores a *satellite* symbol associated with each node. We traverse the nodes of the tree in pre-order, reverse pre-order and breadth-first order. As the traversal visits a node, we find the associated symbol in the array and gather some simple statistics (e.g. the number of nodes with a particular symbol). This set of tasks tests the first-child, last-child, prev-sibling and next-sibling operations[1], all of which are supported in $O(1)$ time by LOUDS and PAREN.

LOUDS stores an n-node ordinal tree as a bit-string of $2n + 1$ bits. Navigation on the tree is performed by rank and select operations on the bit-string:

[1] We currently use a recursive pre-order traversal, so parent is not tested.

$\mathsf{rank}_1(x)$ Returns the number of **1** bits to the left of, and including, position x in the bit-string.

$\mathsf{select}_1(i)$ Given an index i, returns the position of the i-th **1** bit in the bit-string, and -1 if $i \leq 0$ or i is greater than the number of **1**s in the bit-string.

The operations rank_0 and select_0 are defined analogously for the **0** bits in the bit-string; the operations are collectively referred to as rank and select. We refer to a data structure that supports (a nonempty subset of) rank and select operations on a bit-string as a *bit-vector*.

A bit-vector is a fundamental data structure and is used in many succinct and compressed data structures. A bit-vector that supports rank and select in $O(1)$ time can be obtained by augmenting a bit-string of length k with directories occupying $o(k)$ space [6, 3]. Unfortunately, rank and select, though $O(1)$-time asymptotically, are certainly not free in practice. Using the approach of [3] in practice is very slow [7]. In fact, Kim et al. [7] argue that their $3k + o(k)$-bit data structure is more practical than approaches based on [3]. Even in this well-engineered data structure, a select is over three times as slow as a rank, and a rank is somewhat slower than a memory access. Given this, it was perhaps not surprising that a direct implementation of LOUDS (using either of the bit-vectors of [4, 7]) was sometimes over twice as slow as the implementation of PAREN by [4], when parameters were chosen to make the space usages of the data structures somewhat similar. This rather negative result prompted our attempt at engineering LOUDS.

For LOUDS, Jacobson [6] suggested a numbering of nodes from 1 to $2n$. Using his numbering, parent, first-child and last-child all require just one call each to rank and select, and next-sibling and prev-sibling only require the inspection of a bit in the representation of the tree. As nodes are numbered from 1 to $2n$, rather than 1 to n, to access an array of size n that contains information associated with a given node, one has to perform a rank operation on its node number. We first observe that, due to the way rank and select calls are made in LOUDS, one may eliminate calls to rank altogether. The idea, called *double-numbering*, not only speeds up the navigational operations, it also numbers the nodes from 1 to n in level-order, making it easy to access information associated with a node. The resulting data structure, LOUDS1, is indeed much faster than LOUDS, but remains slower and more space-expensive than PAREN.

We then note that, in practice, ordinal trees have a high proportion of leaves: e.g., our XML trees have often about 67% leaves, and a random tree has about 50% leaves. Thus, a rapid test that determines whether a node is a leaf could speed up the first-child operation of LOUDS considerably in practice. We propose a numbering of nodes from 1 to $2n$, which is different from that of [6]. Using this numbering, testing whether a node is a leaf is quick. Applying double-numbering, we again require no rank operations to support navigation in this representation. In this scheme, parent and first-child (for non-leaf nodes), next-sibling and prev-sibling all require up to two select operations, compared with at most one in LOUDS1, and last-child requires one select. This data structure is called LOUDS0. Unfortunately, the performance advantages, such as speeding up first-child for

non-leaf nodes, does not gain enough to make up for the slow next-sibling and prev-sibling operations. The overall speed is poor for pre-order and BFS traversals, but is better (but still poor) for traversals in reverse pre-order.

Finally, we present a new variant of LOUDS, called LOUDS++, which partitions the bit-string representing the tree into two separate bit-strings, and stores them as bit-vectors. The advantages of LOUDS++ are:

- Testing if a node is a leaf, as well as next-sibling and prev-sibling, require only the inspection of bits in one of the bit-strings.
- The other navigational operations (first-child, last-child and parent) are slightly slower than LOUDS1, requiring a rank and a select_1 operation.
- The other operations, degree, childrank and $\mathsf{child}(x, i)$ are as easy as LOUDS1.
- Each of the bit-vectors only needs to support select_1 and rank, while the LOUDS1 bit-vector needs to support both select_1 and select_0 (but not rank, because of double-numbering). This gives significant space savings in practice, since the select directories are usually much larger; in addition, bit-vectors such as that of [7] need the rank directory to implement select.

The rest of this paper is as follows. Immediately following is a short background on XML files. Section 2 discusses bit-vectors, Section 3 introduces LOUDS, including all our variants. Section 4 contains our experimental results.

Representing XML Documents. Our motivation is in the use of this data structure for the representation of (large, static) XML documents. The correspondence between XML documents and ordinal trees is well-known (see e.g. Fig. 1). In this paper we focus on storing the tree structure. The XML Document Object Model (DOM) is a standard interface (see `www.w3.org`) through which applications can access XML documents. DOM implementations store an entire XML document in memory, with its tree structure preserved, but this can take many times more memory than the raw XML file. This 'XML bloat' seriously impedes the scalability and performance of XML query processors [12].

DOM allows tree navigation through the `Node` interface, which represents a single node in the tree. The node interface contains attributes to store information about the node, as well as navigational methods `parentNode`, `firstChild`,

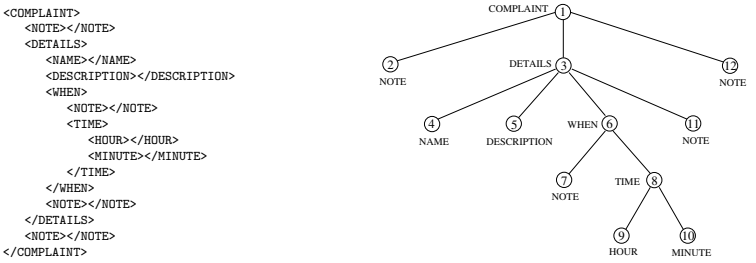


Fig. 1. Left: Small XML fragment (only tags shown). Right: Corresponding tree representation, nodes numbered in depth-first order.

`lastChild`, `previousSibling` and `nextSibling`. The usual way of storing the tree in DOM implementations is to store with each node a pointer to a (subset of) the parent, the first/last child, and the previous/next sibling. We store the tree succinctly, and simulate access to node information by accessing an array. Traversals are important primitives in DOM. There are two main orders of traversals: *document order*, which corresponds to pre-order, and *reverse document order*, which corresponds to reverse pre-order. There is no recognised equivalent of BFS traversal in DOM.

2 Bit-Vector Implementations

We now discuss the space usage of the two bit-vector implementations that we use. The formulae in Table 1 are implicit in [4, 7]. We break the space usage down into constituent parts, as this is important to understand the space-efficiency of LOUDS++. In what follows, k is the size of the bit-string to be represented. The implementations assume a machine with word-size 32 bits (and hence $k \leq 2^{32}$). The space usage figures given below do not include the space for pre-computed tables which are used to perform computations on short bit-strings of 8 or 16 bits (the 'four Russians' trick).[2]

Table 1 gives the space usage of the implementation of Clark-Jacobson bit-vector in [4]. The implementation has three parameters, L, B and s; we show the space usage when $B = 64$, $L = 256$ and $s = 32$. In Table 1, k_0 and k_1 are the numbers of **0**s and **1**s in the bit-string, and l_0 and l_1 are values that depend upon the precise bit-string (the number of so-called 'long gaps'). It is easy to show that $\max\{l_0, l_1\} \leq k/L$, but in practice, the number of long gaps is rather small in most cases (see Section 4). Note that since rank_0 trivially reduces to rank_1 and vice-versa, a single directory suffices to support both rank queries.

We now state the space usage of the 'byte-based' bit-vector Kim et al. [7], again broken down into its various components. Referring to their paper, the space usage of the the directory for select_1 comprises the space usage of its constituent components, including the *delimiter* bit-string and its rank directory (we use a block size of 64 rather than 32). The final terms (c_0 and c_1) are the space usages of the *clump delimiter* and the *clump array*. Their values depend upon the precise distribution of **0**s and **1**s in the bit-string. In the worst case, $c_1 \leq k_0 + 0.043k$, but (as the authors suggest and we confirm) it is much smaller than this upper bound. In order to support select_0 in addition to select_1, we augment the bit-string with a symmetric directory for indexing **0**s, whose space is shown in the last line. The similarity between the space bounds in Table 1 is highlighted in the following remark.

Remark 1. If $\mathsf{select}i$ is to be supported, for only one $i \in \{0, 1\}$, the space bound for the bit-vector implementations is of the form $k + fr(k) + fs(k) + gs(k_i) + hs_i(A)$,

[2] This may seem like cheating, but it is standard practice, since the size of tables is determined by factors such as the size of the cache, independently of k.

Table 1. The space usage of the two bit-vector implementations used

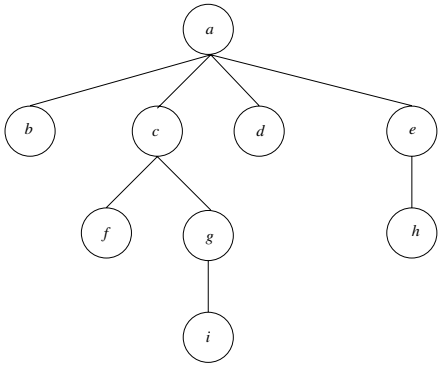
	Clark-Jacobson	Kim et al.
Input bit-string	k	k
$\mathsf{rank}_0/\mathsf{rank}_1$ directory	$0.5k$	$0.25k$
select_0 directory	$0.023k + k_0 + 1024l_0$	$1.25k_1 + c_1$
select_1 directory	$0.023k + k_1 + 1024l_1$	$1.25k_0 + c_0$

where fr, fs and gs are linear functions, indicating the space required for rank and two kinds of select directories respectively, and A is the input bit-string.

For example, if only rank and select_0 are to be supported, the Clark-Jacobson implementation of [4], as described in Table 1 has $fr(k) = 0.5k$, $fs(k) = 0.023k$, $gs(k_0) = k_0$, and $hs_0(A) = 1024l_0$.

3 The LOUDS Representation

The LOUDS bit-string (LBS) is defined as follows. We begin with an empty string. We visit every node in level-order, starting from the root. As we visit a node v with $d \geq 0$ children, we append $\mathbf{1}^d\mathbf{0}$ to the bit-string. Finally, we prefix the bit-string with a **10**, which is the degree of an imaginary 'super-root,' which is the parent of the root of the tree (see Figure 2).



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**

(b)

Vertex	a	b	c	d	e	f	g	h	i
BFS	1	2	3	4	5	6	7	8	9
1-based	1	3	4	5	6	9	10	13	16
0-based	7	8	11	12	14	15	17	18	19

(c)

Vertex	a	b	c	d	e	f	g	h	i
$R0$	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**
$R1$		**1**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**

(d)

Fig. 2. An example ordinal tree (a) and its representations ((b)–(d)).(b) the LOUDS bit-string (LBS); the vertical bars in the LBS have been inserted for readability. The numbers above are the positions of the bits. The initial **10** is for the 'super-root'. (c) Zeros- and ones-based numberings. (d) Partitioned bitvector.

Proposition 1. *The LBS of a tree T with n nodes has n **1**s and $n + 1$ **0**s. The i-th node of T in level-order is represented twice: as the i-th* **1***, which lies within the encoding of the degree of its parent, and is associated with the edge that attaches it to its parent, and also as the $i + 1$-st* **0***, which marks the end of its own degree sequence.*

Ones-based numbering. Jacobson [6] suggests numbering the i-th node in level-order by the position of the i-th **1** bit. This gives a node a number from $\{1, \ldots, 2n+1\}$. To access data associated with a node numbered x, calculating $\mathsf{rank}_1(x)$ numbers the nodes from $\{1, \ldots, n\}$ in level-order.

Zeros-based numbering. Proposition 1 suggests that a node may also be represented by a **0** bit, namely the bit that ends the unary sequence of that node's degree. Again, this is a number from $\{1, \ldots, 2n+1\}$, and a rank_0 operation may be needed to map the nodes to numbers from $\{1, \ldots, n\}$. Figure 3 indicates how the navigational operations might work on the zero-based numbering. Although the operations seem more complex, with the notable exception of isleaf, there is hope that the practical performance may not be too poor, as many of the operations apply select_0 to consecutive zeros in the LBS. Also, next-sibling (prev-sibling) requires only one select for the last (first) child; there are as many last (first) children as non-leaf nodes.

Ones-based numbering

```
isleaf(x)
  See first-child

parent(x)
  select1(rank0(x))

first-child(x)
  y := select0(rank1(x))+1
  if A[y] = 0 then -1 else y

last-child(x)
  y := select0(rank1(x)+1)-1
  if A[y] = 0 then -1 else y

next-sibling(x)
  if A[x+1] = 0 then -1
```

Zero-based numbering

```
isleaf(x)
  (A[x-1] = 0) and (A[x] = 0)

parent(x)
  select0(rank0(select1(rank0(x)-1)+1))

first-child(x)
  if (isleaf(x)) then -1
  else select0(rank1(select0(rank0(x)-1))+2)

last-child(x)
  if (isleaf(x)) then -1
  else select0(rank1(x)+1)

next-sibling(x)
  y := select1(rank0(x)-1)+1
  if A[y] = 0 then -1 else select0(rank0(x))
```

Fig. 3. Navigation operations for zeros-based and ones-based numberings (A is the LBS). prev-sibling is analogous to next-sibling.

3.1 Double-Numbering

Both the ones-based and the zeros-based numberings benefit from the following proposition:

Proposition 2. *Computing* $y = \mathsf{select}_i(x)$, *for* $i = 0$ *or* 1, *also computes* $\mathsf{rank}_0(y)$ *and* $\mathsf{rank}_1(y)$.

Proof. If $y = \mathsf{select}_0(x)$ then $\mathsf{rank}_0(y) = x$ and $\mathsf{rank}_1(y) = y - x$. select_1 is similar.

This allows us to maintain the following invariant. We represent a node as a pair $\langle x, y \rangle$, where y is the position of the node in level-order, and x is the position of the representation of the node in the bit-string. Clearly, depending on whether the numbering is one-based or zero-based, $x = \mathsf{rank}_1(y)$ or $x = \mathsf{rank}_0(y) - 1$. It follows that all computations of the form $\mathsf{rank}(\mathsf{select}(\cdot))$ are really just select operations. Also, since the final step in any (nontrivial) navigation operation is always a select, it follows that the invariant can be maintained at the end of each navigational operation. For example, the call to rank_0 in the parent function in the ones-based numbering in Figure 3 can be implemented as follows:

```
parent(<x,y>)
  rzerox := y - x
  newy   := select1(rzerox)
  newx   := newy - rzerox
  return(<newx, newy>)
```

We refer to the ones-based and zeros-based representations with double-numbering as LOUDS1 and LOUDS0 respectively.

3.2 Partitioned Representation

We now describe a new representation that has the simplicity of LOUDS1 and also allows the isleaf test in $O(1)$ time. The idea is to encode the runs of zeros and ones in the LBS in two separate bit-strings, which we will call R0 and R1. Specifically, if there are runs of **0**s of length $l_1, l_2, \ldots, l_z$ in the LBS, then the bit-string R0 is simply $\mathbf{0}^{l_1-1}\mathbf{1}\mathbf{0}^{l_2-1}\mathbf{1}\ldots\mathbf{0}^{l_z-1}\mathbf{1}$. R1 is defined analogously. Noting that the LBS begins with a **1** and ends with a **0**, it is clearly possible to reconstruct it from R0 and R1. It is now trivial to access the i-th **1** or the $i+1$-st **0** that represents the node numbered i in level-order. This means, in particular, that operations such as isleaf are trivial: the node numbered x in level order is a leaf iff the x-th and $x+1$-st **0** belong to the same run of **0**s, which is easily tested by probing the appropriate bits of R0. Likewise next-sibling and prev-sibling are trivial to implement by looking at R1. LOUDS++ is simply R0 and R1, each augmented with directories to support select_1 and rank^- operations, where:

$\mathsf{rank}^-(x)$ returns the number of **1** bits strictly to the left of position x in the bit-vector. ($\mathsf{rank}^-(x) = \mathsf{rank}_1(x-1)$ except when $x = 1$.)

We now observe:

Proposition 3. select *operations on the LOUDS bit-vector can be simulated by a* select_1 *and a* rank^- *on R0 and R1.*

Proof. We claim that $\mathsf{select}_1(\mathrm{LBS}, i) = \mathsf{select}_1(\mathrm{R0}, \mathsf{rank}^-(\mathrm{R1}, i)) + i$. Note that $\mathsf{rank}^-(\mathrm{R1}, i)$ equals the number of completed runs of **1**s before the run that i is in. There must be an equal number of completed runs of **0**s before i. The select on R0 then gives the total length of these runs, which is then added to i to give the position of the i-th **1**. $\mathsf{select}_0(\mathrm{LBS}, i)$ is similar.

Corollary 1. LOUDS++ *supports the operations* parent, first-child *and* last-child.

Proof. We look at the implementation of these operations in LOUDS1. Due to double-numbering, these operations only have a single select call, which can be simulated as in Proposition 3.

Proposition 4. *The number of* **1***s in R0 and R1 is equal to the number of non-leaf nodes in the input tree plus one.*

Proof. A run of **1**s in the LBS is a node of degree > 0, i.e. a non-leaf node (with the exception of the super-root). The number of **1**s in R1 is the number of runs in the LOUDS bit-string. The number of runs of **0**s in the LBS equals the number of runs of **1**s.

This proposition is key to the good space usage of LOUDS++: not only do we need to support just select_1, but also, the number of non-leaf nodes is usually just a small fraction of the number of nodes. In particular, the (usually considerable) space usage represented by functions $gs()$ in Remark 1 is much reduced. The above representation also gives a nice bound on the number of non-leaf nodes in a random n-node ordinal tree:

Proposition 5. *For any constant $c > 0$, with probability greater than $1 - 1/n^c$, the number of non-leaf nodes in a random n-node ordinal tree is $n/2 \pm o(n)$.*

Proof. (outline) The bit-strings R0 and R1 can be represented using $\lg \binom{n}{t}$ bits, where t is the number of non-leaf nodes. If, for a random random ordinal tree, t deviates significantly from $n/2$, the bit-strings R0 and R1 can be represented using significantly less than n bits, thus giving a representation of the random tree's LBS that uses significantly less than $2n$ bits. However, a simple counting argument shows that no representation of an ordinal tree can represent a random ordinal tree using less than $2n - O(\log n)$ bits, with probability greater than $1 - 1/n^c$, for any constant $c > 0$.

4 Experimental Evaluation

To test our data structures we obtained ordinal trees from the following 6 real-world XML files: xcdna.xml and xpath.xml, which contain genomic data, and mondial-3.0.xml, orders.xml, nasa.xml and treebank_e.xml [14]. We also tested the data structures on randomly generated XML files. These were obtained by using the algorithm described in [11] to generate random parentheses strings. A random parentheses string was converted to an XML file by replacing the opening and closing parentheses of non-leaf nodes by opening and closing tags. The parentheses for leaf nodes were replaced with short text nodes. Our real-world and random files were selected to get some understanding of the behaviour of the data structures as the file size varied with respect to the size of the hardware cache and as the structure of the trees varied. In all cases, the type of each node (element, text node etc.) was stored as a 4-bit value in an accompanying array.

			LOUDS++				LOUDS0/LOUDS1				
			KNKP-BV		CJ-BV		KNKP-BV		CJ-BV		
File	nodes	%leaf	total	clump DS	total	long gap	total	clump DS	total	long gap	Paren
mondial-3.0	57372	78	3.12	0.07	3.82	0.34	5.11	0.11	5.65	0.55	3.73
orders	300003	50	3.78	0.03	5.64	1.60	5.07	0.07	5.10	NEG	3.73
nasa	1425535	67	3.37	0.05	4.27	0.57	5.09	0.09	5.42	0.33	3.73
xpath	2522571	67	3.37	0.04	3.99	0.27	5.08	0.08	5.63	0.53	3.73
treebank_e	7312612	67	3.37	0.04	3.77	0.06	5.08	0.08	5.10	0.01	3.73
xcdna	25221153	67	3.35	0.02	3.80	0.08	5.11	0.11	5.48	0.38	3.73
R62K	62501	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R250K	250001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R1M	1000001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R4M	4000001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	3.73
R16M	16000001	49	3.81	0.04	4.07	NEG	5.08	0.08	5.10	NEG	3.73

Fig. 4. Space Usage. Test file, number of nodes, %leaf node. For LOUDS++ and for LOUDS0/LOUDS1 together: total space per node and space per node for the clump data structure using KNKP-BV; total space per node and space per node to support long gaps using the CJ-BV. For PAREN: space per node, where the numbers are obtained using a generic formula, that does not take into account tree-specific parameters. In [4] this formula was shown to be quite accurate for a wide variety of XML files.

We used Centerpoint XML's DOM [13] implementation to parse the XML files. Our experiments were to traverse the trees and to count the total number of nodes of a particular XML type by accessing the nodetype array. We tested with three different types of traversal, breath-first order, *BFO*, recursive depth-first order, *DFO*, and recursive reverse depth-first order, *RDO*, where we first visit the last child at each nodes and then each of its previous siblings in turn. We compared the three LOUDS data structures with CenterPoint XML's DOM [13] and the PAREN implementation of [4].

We implemented the data structures in C++ and tested them on a dual processor Pentium 4 machine and a Sun UltraSparc-III machine. The Pentium 4 has 512MB RAM, 2.8GHz CPUs and a 512KB L2 cache, running Debian Linux. The compiler was g++ 3.3.5 with optimisation level 2. The UltraSparc-III has 8GB RAM, a 1.2GHz CPU and a 8MB cache, running SunOS 5.9. The compiler was g++ 3.3.2 with optimisation level 2.

For rank and select we used an optimised version of the Clark-Jacobson bit-vector [4], with $B = 64$ and $s = 32$. We refer to this as CJ-BV. We also implemented the bit-vector described in [7], which we refer to as the KNKP-BV. In this data structure we use 256-bit superblocks and 64-bit blocks.

Figure 4 summarises the space usage per node. We see that LOUDS++ generally uses less space than the other LOUDS data structures and with the KNKP-BV its space usage is competitive with the PAREN. Note that LOUDS++ using CJ-BV uses more space than LOUDS1 for the file orders.xml. The structure of the file is such that the number of long gaps in the partitioned bit-strings is relatively large, but there are no long gaps in the LBS.

The performance measure we report is the slowdown relative to DOM of the succinct data structures. We first determine which bit-vector to use. The table below gives the slowdown relative to DOM of LOUDS++ using the KNKP-BV and using the CJ-BV for a DFO traversal on a Pentium 4. The CJ-BV based LOUDS++ outperforms the KNKP-BV based data structure. We saw the same relative performance for LOUDS1 and LOUDS0 and for RDO and BFS traversals. This is not too surprising since the KNKP-BV was designed for sparse bit-vectors, the bit-vectors here are dense. In the remaining experimental results the LOUDS data structures use CJ-BV.

	mond	order	nasa	xpath	treeb	R62K	R250K	R1M	R4M	R16M
KNKP-BV	1.82	3.24	2.82	3.13	3.26	3.63	3.73	3.77	4.09	2.14
CJ-BV	1.46	2.15	2.18	2.23	2.53	2.78	2.84	2.93	3.12	1.73

We now consider RDO traversals. At each node DOM stores a pointer to the parent, first child and next sibling in the tree. So the operation `getLastChild()` requires a traversal across all the children and `getPrevSibling()` at the i-th child requires a traversal across $i-1$ children. At a node with d children DOM performs $O(d^2)$ operations. In the real-world files orders.xml, xpath.xml and treebank_e.xml there is at-least one node with over 2^{14} children and for these files the slowdown relative to DOM of the LOUDS data structure is 0 (to two decimal points), for the other real-world XML files it is between 0.14 and 0.45.

Figure 5 summarises the performance of the data structures for DFO and BFO traversals. We see that LOUDS++ is faster than LOUDS0 or LOUDS1. LOUDS++ is also almost always faster than the PAREN when comparing performance of the basic tree navigation operations.

	Pentium 4								Sun UltraSparc-III							
	DFO				BFO				DFO				BFO			
File	L1	L0	L++	Par	L1	L0	L++	Par	L1	L0	L++	Par	L1	L0	L++	Par
mond	1.99	2.96	**1.46**	1.67	1.08	1.08	**0.80**	0.94	2.47	3.80	**2.15**	2.27	1.91	2.77	1.73	**1.67**
order	2.48	4.04	**2.15**	2.20	1.83	1.83	**1.67**	1.69	1.34	2.35	1.51	**1.33**	0.80	1.25	0.85	**0.74**
nasa	2.80	4.30	**2.18**	2.24	1.38	1.38	**1.11**	1.29	1.20	1.94	**1.16**	1.17	0.66	1.00	0.67	**0.59**
xpath	2.83	4.37	**2.23**	2.29	2.15	2.15	**1.39**	1.60	1.20	1.98	**1.18**	**1.18**	0.71	1.04	0.69	**0.61**
treeb	3.02	4.92	**2.53**	2.62	1.28	1.28	**1.01**	1.44	1.22	1.92	**1.18**	1.29	0.65	0.97	**0.72**	**0.72**
xcdna									1.21	1.95	1.15	**1.13**	0.75	1.03	0.65	**0.61**
R62K	3.17	5.04	**2.78**	3.16	2.06	3.24	**1.75**	3.07	2.30	3.58	**2.40**	2.82	2.14	3.45	**2.32**	3.22
R250K	3.22	5.10	**2.84**	3.22	2.02	3.21	**1.71**	3.01	1.53	2.40	**1.60**	1.85	1.42	2.25	**1.54**	2.12
R1M	3.29	5.25	**2.93**	3.19	1.92	3.08	**1.69**	2.96	1.23	1.90	**1.31**	1.75	1.12	1.78	**1.21**	1.71
R4M	3.46	5.61	**3.12**	3.21	1.13	1.86	**0.97**	3.01	1.24	1.93	**1.34**	1.77	0.99	1.59	**1.08**	1.57
R16M	1.76	2.97	**1.73**	1.84	0.50	0.80	0.44	**0.30**	1.22	1.93	**1.32**	1.81	0.61	0.96	**0.65**	1.17

Fig. 5. Performance evaluation on Pentium 4 and Sun UltraSparc-III.: Test file, slowdown relative to DOM for depth-first order (DFO) and breath-first order (BFO) traversals for LOUDS1 (L1), LOUDS0 (L0), LOUDS++ (L++) all using CJ-BV and for PAREN. The fastest data structure for each result is set in bold font. DOM could not fit XCNDA.xml into the internal memory of the Pentium 4.

5 Conclusions and Future Work

We have presented a partitioned version of Jacobson's [6] LOUDS representation, called LOUDS++, that appears to outperform other succinct tree representations in practice. Although we have demonstrated experimentally that LOUDS++ uses less space than LOUDS, this could be understood on a firmer theoretical basis. It would be interesting to see whether the partitioning idea generalises to other applications.

Acknowledgement. We thank Richard Geary for useful discussions.

References

1. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman and S. S. Rao, Representing trees of higher degree. *Algorithmica* **43** (2005), pp. 275-292.
2. Chiang, Y.-T., Lin, C.-C. and Lu, H.-I. Orderly spanning trees with applications. *SIAM Journal on Computing*, **34** (2005), pp. 924-945.
3. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th ACM-SIAM SODA*, pp. 383–391, 1996.
4. R. F. Geary, N. Rahman, R. Raman and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Symposium on Combinatorial Pattern Matching*, Springer LNCS 3109, pp. 159–172, 2004.
5. R. F. Geary, R. Raman and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th ACM-SIAM SODA*, pp. 1–10, 2004.
6. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, 549-554, 1989.
7. D. K. Kim, J. C. Na, J. E. Kim and K. Park. Efficient implementation of Rank and Select functions for succinct representation. In *Proc. 4th Intl. Wkshp. Efficient and Experimental Algorithms*, LNCS 3505, pp. 315–327, 2005.
8. J. I. Munro. Tables. In *Proc. 16th FST&TCS conference*, LNCS 1180, 37-42, 1996.
9. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Computing*, **31** (2001), pp. 762–776.
10. J. I. Munro, and S. S. Rao, Succinct representations of functions. In *Proc. 31st ICALP*, LNCS 3142, pp. 1006-1015, 2004.
11. H. W. Martin and B. J. Orr. A random binary tree generator. *Proceedings of the 17th ACM Annual Computer Science Conference*, pp. 33–38, 1989.
12. http://xml.apache.org/xindice/FAQ.
13. Centerpoint XML, http://www.cpointc.com/XML.
14. UW XML Repository. `http://www.cs.washington.edu/research/xmldatasets/`.
15. http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/traversal.html