Succinct and I/O Efficient Data Structures for Traversal in Trees[⋆]

Craig Dillabaugh, Meng He, and Anil Maheshwari

Carleton University, School of Computer Science
5302 Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada
Corresponding author: Craig Dillabaugh, cdillaba@connect.carleton.ca
Phone: 1-613-520-2600x8756 Fax: 1-613-520-4334

Abstract. We present two results for path traversal in trees, where the traversal is performed in an asymptotically optimal number of I/Os and the tree structure is represented succinctly. Our first result is for bottom-up traversal that starts with a node in the tree T and traverses a path to the root. For a tree on N nodes, and for a path of length K, we design data structures that permit traversal of the bottom-up path in $O(K/B)$ I/Os using only $2N + \frac{\epsilon N}{\log_B N} + o(N)$ bits, for an arbitrarily selected constant, ϵ, where $0 < \epsilon < 1$. Our second result is for top-down traversal in binary trees. We store T using $(3 + q)N + o(N)$ bits, where q is the number of bits required to store a key, while top-down traversal can still be performed in an asymptotically optimal number of I/Os.

Keywords succinct data structures, I/O efficient data structures, data structures, trees, path traversal

1 Introduction

Many operations on graphs and trees can be viewed as the traversal of a path. Queries on trees, for example, typically involve traversing a path from the root to some node, or from some node to the root. Often the datasets represented in graphs and trees are too large to fit in internal memory and traversal must be performed efficiently in external memory (EM). Efficient EM traversal in trees is important for structures such as suffix trees, and as a building block to graph searching and shortest path algorithms. In both cases huge datasets are often dealt with. Suffix trees are frequently used to index very large texts or collections of texts, while large graphs are common in numerous applications such as Geographic Information Systems.

Succinct data structures were first proposed by Jacobson [1]. The idea is to represent data structures using space as near the information-theoretical lower bound as possible, while allowing efficient navigation. Succinct data structures, which have been studied largely outside the external memory model, also have natural application to large data sets.

In this paper, we present data structures for traversal in trees that are both efficient in the EM setting, and that encode the trees succinctly. We are aware of only the work by Chien *et al.* [2] on succinct full-text indices supporting efficient substring search in EM, that follows the same track. Our contribution is the first such work on general trees that bridges these two techniques.

[⋆] Research supported by NSERC.

1.1 Previous Work

The I/O-model [3] splits memory into two levels; the fast, but finite, internal memory; and slow but infinite EM. Data is transferred between these levels by an input-output operation (I/O). In this model, algorithms are analyzed in terms of the number of I/O operations required to complete a process. The unit of memory that may be transferred in a single I/O is referred to as a *disk block*. In the I/O-model the parameters B, M, and N are used, respectively, to represent the size (in terms of the number of data elements) of a block, internal memory, and the problem instance. *Blocking* of data structures in the I/O model has reference to the partitioning of the data into individual blocks that can subsequently be transferred with a single I/O.

Nodine *et al.* [4] studied the problem of blocking graphs and trees, for efficient traversal, in the I/O model. In particular they looked at trade-offs between I/O efficiency and space when redundancy of data was permitted. The authors arrived at matching upper and lower bounds for complete d-ary trees and classes of general graphs with close to uniform average vertex degree. Among their main results they presented a bound of $\Theta(\log_d B)$ for d-ary trees where on average each vertex may be represented twice. Blocking of bounded degree planar graphs, such as Triangular Irregular Networks (TINs), was examined in Aggarwal *et al.* [5]. The authors show how to store a planar graph of size N, and of bounded degree d, in $O(N/B)$ blocks so that any path of length K can be traversed using $O(K/\log_d B)$ I/Os.

Hutchinson *et al.* [6] examined the case of bottom-up traversal, where the path begins with some node in T and proceeds to the root. They gave a blocking which supports bottom-up traversal in $O(K/B)$ I/Os when the tree is stored in $O(N/B)$ blocks. The case of top down traversal has been more extensively studied. Clark and Munro [7] describe a blocking layout that yields a logarithmic bound for root-to-leaf traversal in suffix trees. Given a fixed independent probability on the leaves, Gil and Itai [8], presented a blocking layout that yields the minimum expected number of I/Os on a root to leaf path. In the cache oblivious model, Alstrup *et al.* [9] gave a layout that yields a minimum worst case, or expected number of I/Os, along a root-to-leaf path, up to constant factors. Demaine *et al.* [10] presented an optimal blocking strategy that yields differing I/O complexity depending on the length of the path (see Lemma 3). Finally, Brodal and Fagerberg [11] describe the giraffe-tree, which likewise permits a $O(K/B)$ root-to-leaf tree traversal with $O(N)$ space in the cache-oblivious model.

1.2 Our Results

Throughout this paper we assume that $B = \Omega(\lg N)$ (i.e. the disk block is of reasonable size)[1]. Our paper presents two main results:

1. In Section 3, we show how a tree T can be blocked in a succinct fashion such that a bottom-up traversal requires $O(K/B)$ I/Os using only $2N + \frac{\epsilon N}{\log_B N} + o(N)$ bits to store T, where K is the path length and $0 < \epsilon < 1$. This technique is based on [6], and achieves an improvement on the space bound by a factor of $\lg N$.

[1] In this paper $\lg N = \lg_2 N$, where another base is used we state this explicitly (i.e. $\log_B N$).

2. In Section 4, we show that a binary tree, with keys of size $q = O(\lg N)$ bits, can be stored using $(3 + q)N + o(N)$ bits so that a root-to-node path of length K can be reported with: (a) $O\left(\frac{K}{\lg(1+(B\lg N)/q)}\right)$ I/Os, when $K = O(\lg N)$; (b) $O\left(\frac{\lg N}{\lg(1+\frac{B\lg^2 N}{qK})}\right)$ I/Os, when $K = \Omega(\lg N)$ and $K = O\left(\frac{B\lg^2 N}{q}\right)$; and (c) $O\left(\frac{qK}{B\lg N}\right)$ I/Os, when $K = \Omega\left(\frac{B\lg^2 N}{q}\right)$. This result achieves a $\lg N$ factor improvement on the previous space cost in [10] for the tree structure. We further show that when the size q of a key is constant that we improve the I/O efficiency for the case where $K = \Omega(B\lg N)$ and $K = O(B\lg^2 N)$ from $\Omega(\lg N)$ to $O(\lg N)$ I/Os.

2 Preliminaries

2.1 Bit Vectors

A key data structure used in our research is a bit vector $B[1..N]$ that supports the operations *rank* and *select*. The operations $\texttt{rank}_1(B, i)$ and $\texttt{rank}_0(B, i)$ return the number of 1s and 0s in $B[1..i]$, respectively. The operations $\texttt{select}_1(B, r)$ and $\texttt{select}_0(B, r)$ return the position of the r^{th} occurrences of 1 and 0, respectively. Several researchers [1, 7, 12] considered the problem of representing a bit vector succinctly to support `rank` and `select` in constant time under the word RAM model with word size $\Theta(\lg n)$ bits, and their results can be directly applied to the external memory model. The following lemma summarizes some of these results, in which part (a) is from Jacobson [1] and Clark and Munro [7], while part (b) is from Raman *et al.* [12]:

Lemma 1. *A bit vector B of length N can be represented using either: (a) $N + o(N)$ bits, or (b) $\lceil\lg\binom{N}{R}\rceil + O(N\lg\lg N/\lg N) = o(N)$ bits, where R is the number of 1s in B, to support the access to each bit,* `rank` *and* `select` *in $O(1)$ time (or $O(1)$ I/Os in external memory).*

2.2 Succinct Representations of Trees

As there are $\binom{2n}{n}/(n + 1)$ different binary trees (or ordinal trees) on N nodes, various approaches [1, 13, 14] have been proposed to represent a binary tree (or ordinal tree) in $2N + o(N)$ bits, while supporting efficient navigation. Jacobson [1] first presented the *level-order binary marked* (LOBM) structure for binary trees, which can be used to encode a binary tree as a bit vector of $2N$ bits. He further showed that operations such as retrieving the left child, the right child and the parent of a node in the tree can be performed using `rank` and `select` operations on bit vectors.

We make use of his approach to encode tree structures in Section 4. Another approach we use in this paper is based on the isomorphism between *balanced parenthesis sequences* and ordinal trees, proposed by Munro and Raman [13]. The balanced parenthesis sequence of a given tree can be obtained by performing a depth-first traversal, and outputting an opening parenthesis the first time a node is visited, and a closing parenthesis at the final visit. The complete subtree of a node is output between its opening and closing parentheses.

Munro and Raman [13] designed a succinct representation of an ordinal tree of N nodes in $2N+o(N)$ bits based on the balanced parenthesis sequence, which supports the computation of the parent, the depth and the number of descendants of a node in constant time, and the i^{th} child of a node in $O(i)$ time.

2.3 I/O Efficient Tree Traversal

Hutchinson *et al.* [6] presented a blocking technique for rooted trees in the I/O model that supports bottom-up traversal, their result is summarized in the following lemma:

Lemma 2. *A rooted tree T can be stored in $O(N/B)$ blocks on disk such that a bottom-up path of length K in T can be traversed in $O(K/\tau B)$ I/Os, where $0 < \tau < 1$ is a constant.*

Their data structure involves cutting T into layers of height τB, where τ is a constant $(0 < \tau < 1)$. A forest of subtrees is created within each layer, and the subtrees are stored in blocks. If a subtree needs to be split over multiple blocks, then the path to the top of the layer is stored for that block. This ensures that the entire path within a layer can be read by performing a single I/O.

Demaine *et al.* [10] described an optimal blocking technique for binary trees that bounds the number of I/Os in terms of the depth of the node within T. The blocking has two phases. The first blocks the first $c \lg N$ levels of the tree, where c is a constant, as if it were a complete tree. In the second phase, nodes are assigned recursively to blocks in a top-down manner. The proportion of nodes in either child's subtree assigned to the current block is determined based on the sizes of the subtrees. The following lemma summarizes their results:

Lemma 3. *For a binary tree T, a traversal from the root to a node of depth K requires the following number of I/Os:*

1. $\Theta(K/\lg(1+B))$, *when* $K = O(\lg N)$,
2. $\Theta(\lg N/(\lg(1+B\lg N/K)))$, *when* $K = \Omega(\lg N)$ *and* $K = O(B \lg N)$, *and*
3. $\Theta(K/B)$, *when* $K = \Omega(B \lg N)$.

3 Bottom Up Traversal

In this section, we present a set of data structures that encode a tree T succinctly so that the I/Os performed in traversing a path from a given node to the root is asymptotically optimal. Let A denote the maximum number of nodes that can be stored in a single block, and let K denote the length of the path. Given the bottom up nature of the queries, there is no need to encode a node's key value, since the path always proceeds to the current node's parent.

3.1 Blocking Strategy

Our blocking strategy is based on [6], we have modified the technique and introduced new notation. We partition T into layers of height τB where $0 < \tau < 1$. We permit the top layer and the bottom layer to contain fewer than τB levels as doing so provides the freedom to partition the tree into layers with a favourable distribution of nodes. We then group the nodes of each layer into blocks, and store with each block a *duplicate path* which is defined later in this section. In order to bound the space required by block duplicate paths, we further group blocks into *superblocks*. The duplicate path of a superblock's first block is the *superblock duplicate path* for that superblock. By loading at most the block containing a node, along with its associated duplicate path, and the superblock duplicate path we demonstrate that a layer can be traversed with at most $O(1)$ I/Os. A set of bit vectors, to be described later, that map the nodes at the top of one layer to their parents in the layer above are used to navigate between layers.

Layers are numbered starting at 1 for the topmost layer. Let L_i be the i^{th} layer in T. The layer is composed of a forest of subtrees whose roots are all at the top level of L_i. We now describe how the blocks and superblocks are created within L_i. We number L_i's nodes in preorder starting from 1 for the leftmost subtree and numbering the remaining subtrees from left to right. Once the nodes of L_i are numbered they are grouped into blocks of consequtive preorder number. Each block stores a portion of T along with the representation of its duplicate path, or the superblock duplicate path, if it is the first block in a superblock. We refer to the space used to store the duplicate path as *redundancy* which we denote W. In our succinct tree representation we require two bits to represent each node in the subtrees of L_i, so for blocks of $B \lg N$ bits we have:

$$A = \left\lfloor \frac{B \lg N - W}{2} \right\rfloor \tag{1}$$

We term the first block in a layer the *leading block*. Layers are blocked in such a manner that the leading block is the only block permitted to be non-full (may contain less than A nodes). The leading block requires no duplicate structure and thus $W = 0$ for leading blocks. All superblocks except possibly the first, which we term the *leading superblock*, contain exactly $\lfloor \lg B \rfloor$ blocks (see Fig. 1).

We select as a block's duplicate path the path from the node with minimum preorder number in the block (or superblock) to the layer's top level. This path has length at most τB and satisfies the following property:

Property 1. Given a block (or superblock) Y, for any node x in Y there exists a path from x to either the top of its layer, or to the duplicate path of Y, which consists entirely of nodes in Y.

Proof. Let T_v be the subtree induced by the layering which contains v. Now consider node $x \in Y$. For the case in which $x \in T_v$, the proof follows directly from the proof of Lemma 2, Property 3 in [6]. The second case is $x \notin T_v$. Let T_j be the subtree induced by the layering that contains x. Assume, contrary to what we wish to prove, that a node z exists, which is

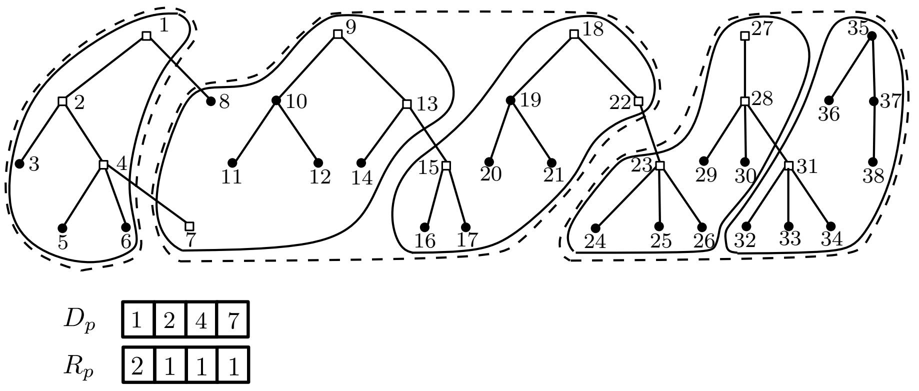


Fig. 1. Blocking within a layer is shown, along with block (solid lines) and superblock (dashed lines) boundaries. Numbers shown are the layer preorder values. Nodes on the duplicate paths are indicated by a hallow square. Below the graph the contents of the duplicate path array D_p and the root-to-path array R_p are shown for the second block (first block of the second superblock).

not in Y, on the path from x to the root of T_x. By the properties of preorder numbering, all nodes in T_v are numbered prior to the root of T_x. Furthermore, the preorder number of x is greater than the preorder number of the root of T_x as are all nodes on the from x path to T_x in the pre-order numbering scheme. Thus if $x \in Y$, then the path from x to the root of T_x is also in Y, and we have a contradiction. $\square$

3.2 Data Structures

Each block is encoded by three data structures:

1. An encoding of the tree structure, denoted B_e. The subtree(s) contained, or partially contained, within the block are encoded as a sequence of balanced parentheses (see Section 2). Note that in this representation, the i^{th} opening parenthesis corresponds to the i^{th} node in preorder in this block. More specifically, a preorder traversal of the subtree(s) is performed (again from left to right for blocks with multiple subtrees). At the first visit to a node an open parenthesis is output. When a node is visited for the last time (going up) a closing parenthesis is output. Each matching parenthesis pair represents a node while the parentheses in between represent the subtree rooted at that node. Consider the node v, for which the encoding opening 1 (open parenthesis) is the i^{th} such 1 bit in B_e, the preorder number of v within the block is i.
2. The duplicate path array, $D_p[j]$, for $1 < j \leq \tau B$. Let v be the node with the smallest preorder number in the block. Entry $D_p[j]$ stores the preorder number of the node at the j^{th} level on the path from v to the top level of the layer. The number recorded is the preorder number with respect to the block's superblock. It may be the case that v is not at the τB^{th} level of the layer. In this case the entries below v are set to 0. Recall that preorder numbers begin at 1, so the 0 value effectively flags an entry as invalid.
3. The root-to-path array, $R_p[j]$, for $1 < j \leq \tau B$. A block may include many subtrees rooted at nodes on the duplicate path. Consider the set of roots of these subtrees. The entry at

$R_p[j]$ stores the number of subtrees rooted at nodes on D_p from level τB up to level j. The number of subtrees rooted at node $D_p[j]$ can be calculated by evaluating $R_p[j] - R_p[j+1]$ when $j < \tau B$, or $R_p[j]$ when $j = \tau B$.

Now consider the first block in a superblock. The duplicate path of this block is the superblock's duplicate path. Unlike the duplicate path of a regular block, which stores the preorder numbers with respect to the superblock, the superblock's duplicate path stores the preorder numbers with respect to the preorder numbering in the layer. Excluding a superblock's first block, consider the duplicate paths of the remaining blocks within a superblock. These paths may share a common subpath with the superblock duplicate path. Each entry on a block duplicate path that is shared with the superblock duplicate path is set to -1.

For an arbitrary node $v \in T$, let v's layer number be ℓ_v and its preorder number within the layer be p_v. Each node in T is uniquely represented by the pair (ℓ_v, p_v). Let π define the lexicographic order on these pairs. Given a node's ℓ_v and p_v values, we can locate the node and navigate within the corresponding layer. The challenge is how to map between the roots of one layer and their parents in the layer above. Consider the set of N nodes in T. We define the following data structures, which will facilitate mapping between layers:

1. Bit vector $\mathcal{V}_{first}[1..N]$, where $\mathcal{V}_{first}[i] = 1$ iff the i^{th} node in π is the first node within its layer.
2. Bit vector $\mathcal{V}_{parent}[1..N]$, where $\mathcal{V}_{parent}[i] = 1$ iff the the i^{th} node in π is the parent of some node at the top level of the layer below.
3. Bit vector $\mathcal{V}_{first\_child}[1..N]$, where $\mathcal{V}_{first\_child}[i] = 1$ iff the i^{th} node in π is a root in its layer and its parent in the preceeding layer differs from that of the previous root in this layer.

Figure 2 demonstrates how the four bit vectors respresent the mapping between nodes in different layers.

All leading blocks are packed together on disk, separate from the full blocks. Note that leading blocks do not require a duplicate path or root-to-path array, so only the tree structure need be stored for these blocks. Due to the packing, the leading block may overrun the boundary of a block on disk. We use the first $\lg B$ bits of each disk block to store an offset which indicates the position of the starting bit of the first leading block starting in the disk block. This allows us to skip any overrun bits from a leading block stored in the previous disk block.

We store two bit arrays to aid in locating blocks. The first indexes the partially full leading blocks. The second indexes the full blocks. Let x be the number of layers on T, and let z be the total number of full blocks over all layers. The bit vectors are:

1. Bit vector $\mathcal{B}_l[1..x]$, where $\mathcal{B}_l[i] = 1$ iff the i^{th} leading block resides in a different disk block than the $(i-1)^{\text{th}}$ leading block.
2. Bit vector $\mathcal{B}_f[1..(x+z)]$ that encodes the number of full blocks in each layer in unary. More precisely, $\mathcal{B}_f[1..(x+z)] = 0^{l_1}10^{l_2}10^{l_3}1\ldots$, where l_i is the number of full blocks in layer i.

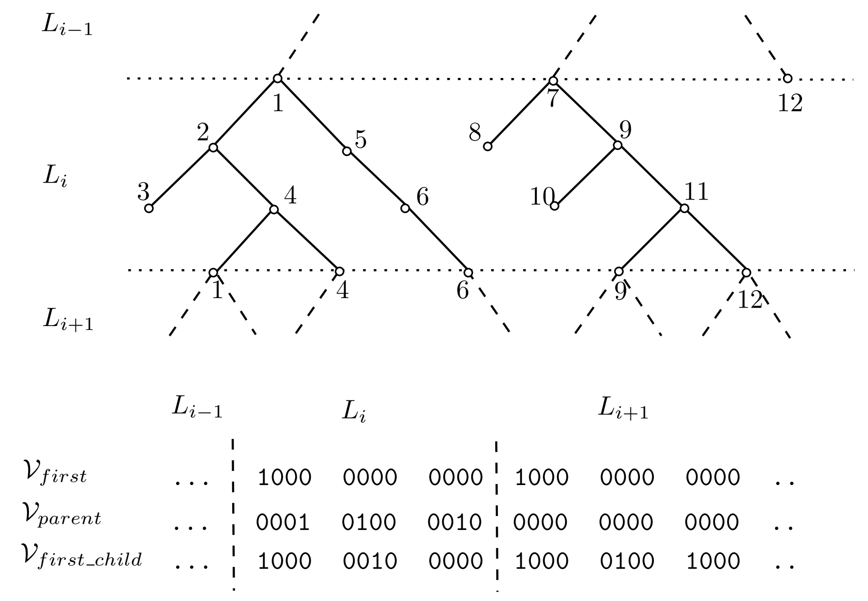

L_{i-1}
L_i
L_{i+1}
1
2
3
4
5
6
7
8
9
10
11
12
L_{i-1} L_i L_{i+1}
$\mathcal{V}_{first}$ ... 1000 0000 0000 1000 0000 0000 ..
$\mathcal{V}_{parent}$ ... 0001 0100 0010 0000 0000 0000 ..
$\mathcal{V}_{first\_child}$ ... 1000 0010 0000 1000 0100 1000 ..

Fig. 2. Scheme for mapping between layers. Shown at the top is the portion of the tree stored in layer L_i (hallow nodes) and the root nodes of layer L_{i+1} (solid nodes). The dashed horizontal lines indicate the top level of each layer. The bottom part of the figure shows the corresponding portions of the bit vectors used to maintain the mapping between layer L_i and its neighbouring layers. Vector $\mathcal{V}_{first}$ delimits the layers, $\mathcal{V}_{parent}$ identifies parent nodes, and $\mathcal{V}_{first\_child}$ maps root nodes to their parent at the level above.

We also define a *leading* superblock, which may hold fewer than $\lceil \lg B \rceil$ blocks. For all layers the first superblock is the leading superblock, all other superblocks hold $\lg B$ full blocks. As the first block in the leading superblock is the leading block this superblock contains no superblock dupcliate path.
To analyze the space costs of our data structures we have the following lemma.

Lemma 4. *The data structures described above occupy* $2N + \frac{12\tau N}{\log_B N} + o(N)$ *bits.*

Proof. First consider the number of bits used to store the actual tree structure of T. The balanced parentheses encoding requires $2N$ bits and our subdivision of T into blocks does not duplicate any nodes, as such we store the structure of T with the optimal $2N$ bits. We must also account for the space required to store the duplicate paths and their associated root-to-path arrays. The space required for block and superblock duplicate paths differs:

1. For each node in a block duplicate path, we store its preorder number in the superblock. As there are at most $(B\lceil \lg B \rceil \lceil \lg N \rceil)/2$ nodes in a superblock so $\lceil \lg ((B\lceil \lg B \rceil \lceil \lg N \rceil)/2) \rceil$ bits are sufficient to store each node on the path. As the duplicate path has τB entries the array requires the following number of bits:

$$\tau B \left\lceil \lg \left(\frac{B\lceil \lg B \rceil \lceil \lg N \rceil}{2}) \right) \right\rceil \leq \tau B \left(\lg \left(\frac{B\lceil \lg B \rceil \lceil \lg N \rceil}{2} \right) + 1 \right)$$
$$= \tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg B \rceil \rceil + \lceil \lg \lceil \lg N \rceil \rceil) \qquad (2)$$

2. The superblocks duplicate path stores preorder values with respect to the layer preorder numbering. There can be N nodes in a layer, so each entry can require $\lceil \lg N \rceil$ bits. The total space for the duplicate path is thus $\tau B \lceil \lg N \rceil$.

We store the array R_p for each block. As a block may have as many as $(B\lceil \lg N \rceil)/2$ nodes, each entry in R_p requires $\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil$ bits. Thus the space per block for this array is $\tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil)$ bits. This value holds whether the corresponding duplicate path is associated with a block or superblock.

For a regular block, the number of bits used to store both D_p and R_p is:

$$\begin{aligned} &\tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg B \rceil \rceil + \lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil) \\ &= \tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \end{aligned} \tag{3}$$

Now consider the total space required for all duplicate paths and root-to-path arrays within a superblock. The superblock duplicate path requires $\tau B \lceil \lg N \rceil$ bits. The $(\lceil \lg B \rceil - 1)$ remaining blocks each require the number of bits given by Eq. 3, thus the total redundancy per superblock, W, becomes:

$$\begin{aligned} W = &\tau B \lceil \lg N \rceil + \tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil) \\ &+ (\lceil \lg B \rceil - 1)\tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \end{aligned} \tag{4}$$

The average redundancy per block is then:

$$\begin{aligned} W_b = &\frac{\tau B \lceil \lg N \rceil}{\lceil \lg B \rceil} + \frac{\tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil)}{\lceil \lg B \rceil} \\ &+ \frac{(\lceil \lg B \rceil - 1)\tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil)}{\lceil \lg B \rceil} \\ &\leq \tau B \lceil \log_B N \rceil + \tau B(3\lceil \lg B \rceil + 3\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \end{aligned} \tag{5}$$

The value for the average redundancy, W_b, represents the worst case per block redundancy. The redundancy for leading blocks is $\lceil \lg B \rceil / (B \lceil \lg N \rceil) < W_b$ bits, since a $\lceil \lg B \rceil$-bit offset is stored for each of the blocks into which the leading blocks are packed. Therefore, including the leading blocks and superblocks in the analysis would not increase W_b.

The total number of blocks required to store T is $\frac{2N}{B\lceil \lg N \rceil - W_b}$. The the total size of the redundancy for T, denoted W_t is:

$$W_t = \frac{2N}{B\lceil \lg N \rceil - W_b} \cdot W_b < \frac{2N \cdot 2W_b}{B\lceil \lg N \rceil} \tag{6}$$

when $W_b < \frac{1}{2} B \lceil \lg N \rceil$.

We can ensure this condition is true by selecting a suitable value for τ as follows:

$$\begin{aligned}
B\lceil \lg N\rceil &> 2\cdot W_b\\
B\lceil \lg N\rceil &> 2\tau B\lceil \log_B N\rceil + \tau B(6\lceil \lg B\rceil + 6\lceil \lg\lceil \lg N\rceil\rceil + 2\lceil \lg\lceil \lg B\rceil\rceil)\\
\lceil \lg N\rceil &> 2\tau\lceil \log_B N\rceil + 6\tau\lceil \lg B\rceil + 6\tau\lceil \lg\lceil \lg N\rceil\rceil + 2\tau\lceil \lg\lceil \lg B\rceil\rceil \qquad (7)
\end{aligned}$$

Noting that each term on the right hand size of Eq. 7 consists of a constant, τ, and an expression less than $\lceil \lg N\rceil$ we have:

$$\begin{aligned}
\lceil \lg N\rceil &\geq 16\tau\lceil \lg N\rceil\\
\tau &\leq \frac{1}{16} \qquad (8)
\end{aligned}$$

Finally, we substitute for W_b in Eq. 6, to obtain the following:

$$\begin{aligned}
W_t &= \frac{2N\cdot 2\cdot(\tau B\lceil \log_B N\rceil + \tau B(3\lceil \lg B\rceil + 3\lceil \lg\lceil \lg N\rceil\rceil + \lceil \lg\lceil \lg B\rceil\rceil))}{B\lceil \lg N\rceil}\\
&= \frac{4N\tau B\lceil \log_B N\rceil}{B\lceil \lg N\rceil} + \frac{12N\tau B\lceil \lg B\rceil}{B\lceil \lg N\rceil} + \frac{12N\tau B\lceil \lg\lceil \lg N\rceil\rceil}{B\lceil \lg N\rceil} + \frac{4N\tau B\lceil \lg\lceil \lg B\rceil\rceil}{B\lceil \lg N\rceil}\\
&= \frac{4N\tau\lceil \log_B N\rceil}{\lceil \lg N\rceil} + \frac{12N\tau\lceil \lg B\rceil}{\lceil \lg N\rceil} + \frac{12N\tau\lceil \lg\lceil \lg N\rceil\rceil}{\lceil \lg N\rceil} + \frac{4N\tau\lceil \lg\lceil \lg B\rceil\rceil}{\lceil \lg N\rceil}\\
&= \frac{12\tau N}{\lceil \log_B N\rceil} + o(N) \qquad (9)
\end{aligned}$$

We arrive at out final bound because the first, third, and fourth terms are each asymptotically $o(N)$ (recall that we assume $B = \Omega(\lg N)$).

Now we consider the space required to store $\mathcal{V}_{first}$, $\mathcal{V}_{parent}$, and $\mathcal{V}_{first\_child}$. Each vector must index N bits. However, using Lemma 1b, we can do better than $3N + o(N)$ bits of storage, if we consider that the number of 1's in each bit vector is small.

For $\mathcal{V}_{first}$, the total number of 1's is $N/(\tau B)$ in the worst case, when T forms a single path. The number of 1's appearing in $\mathcal{V}_{parent}$ is bounded by the number of roots appearing on the top level of the layer below. As stated in [6], we have the flexibility to pick an arbitrary level within the first τB levels as the top of the second layer. Given this flexibility, we can pick a division of T into layers such that the total number of nodes on the top level of layers is bounded by $N/(\tau B)$. The bit vectors $\mathcal{V}_{first\_child}$ has at most as many 1 bits as $\mathcal{V}_{parent}$ and as such the number of 1 bits in each of the three vectors is bounded by $N/(\tau B)$.

Finally we have the bit vectors $\mathcal{B}_l$, and $\mathcal{B}_f$. $\mathcal{B}_l$ stores a single bit for each leading block. There are as many leading blocks as there are layers so this bit vector has at most $N/(\tau B)$ bits, when T forms a single path. $\mathcal{B}_f$ has a 1 bit for every layer proceeded as many 0 bits as there are full blocks on that layer. Once again the number of 1 bits is bounded by the maximum number of layers $N/(\tau B)$. That the total number of bits in $\mathcal{B}_f$ is likewise bounded by $N/(\tau B)$, can be easily demonstrated as follows. Starting with T as a path there will be no full blocks so $\mathcal{B}_f$ has $N/(\tau B)$ 1 bits and no zero bits. Now consider transforming T so that some layers contain full blocks. For every full block we create we reduce the maximum height of T by at least τB. Thus for every 0 bit added to $\mathcal{B}_f$ at least one 1 bit is removed so the size of $\mathcal{B}_f$ is bounded by $N/(\tau B)$.

By Lemma 1b, the bit vectors $\mathcal{B}_l$, and $\mathcal{B}_f$ require $\left\lceil \lg \binom{N}{N/(\tau B)} \right\rceil + O(N \lg\lg N / \lg N) = o(N)$ bits. □

3.3 Navigation

The algorithm for reporting a node-to-root path is given by algorithms *ReportPath*(T, v) (see Fig. 3) and *ReportLayerPath*(ℓ_v, p_v) (see Fig. 4). Algorithm *ReportPath*(T, v) is called with v being the number of a node in T given by π. *ReportPath* handles navigation between layers and calls *ReportLayerPath* to perform the traversal within each layer. The parameters ℓ_v and p_v are the layer number and the preorder value of node v within the layer, as previously described. *ReportLayerPath* returns the preorder number, within layer ℓ_v of the root of path reported from that layer. In *ReportLayerPath* we find the block b_v containing node v using the algorithm *FindBlock*(ℓ_v, p_v) described in Fig. 5. We now have the following lemma.

Lemma 5. *The algorithm ReportPath traverses a path of length K in T in $O(K/\tau B)$ I/Os.*

Proof. At each layer we progress τB steps toward the root of T. We must load the block containing the current node and possibly the block storing the superblock duplicate path. When we step between layers, we must then account for the I/Os involved in mapping the layer level roots to their parents in the predecessor layer. This involves a constant number of *rank* and *select* operations which may be done in $O(1)$ I/Os.

The *FindBlock* algorithm involves a scan on the disk blocks storing leading blocks, but this may generate at most 2 I/Os. The remaining operations in *FindBlock* use a constant number of *rank* and *select* calls, and therefore require $O(1)$ I/Os. □

```
Algorithm ReportPath(T, v)

1. Find $\ell_v$, the layer containing $v$. $\ell_v = \texttt{rank}_1(\mathcal{V}_{first}, v)$.
2. Find $\alpha_{\ell_v}$, the position in $\pi$ of $\ell_v$'s first node. $\alpha_{\ell_v} = \texttt{select}_1(\mathcal{V}_{first}, \ell_v)$.
3. Find $p_v$, $v$'s preorder number within $\ell_v$. $p_v = v - \alpha_{\ell_v}$.
4. Repeat the following steps until the top layer has been reported.
   (a) Let $r = ReportLayerPath(\ell_v, p_v)$ be the preorder number of the root of the path in layer $\ell_v$
       (This step also reports the path within the layer).
   (b) Find $\alpha_{(\ell_v-1)}$, the position in $\pi$ of the first node at the next higher layer. $\alpha_{(\ell_v-1)} =$
       $\texttt{select}_1(\mathcal{V}_{first}, \ell_v - 1)$.
   (c) Find $\lambda$, the rank of $r$'s parent among all the nodes in the layer above that have children in $\ell_v$.
       $\lambda = (\texttt{rank}_1(\mathcal{V}_{first\_child}, \alpha_{\ell_v} + r)) - (\texttt{rank}_1(\mathcal{V}_{first\_child}, \alpha_{\ell_v} - 1)$.
   (d) Find which leaf $\delta$, at the next higher layer corresponds to $\lambda$. $\delta =$
       $\texttt{select}_1(\mathcal{V}_{parent}, \texttt{rank}_1(\mathcal{V}_{parent}, \alpha_{(\ell_v-1)}) - 1 + \lambda)$.
   (e) Update $\alpha_{\ell_v} = \alpha_{(\ell_v-1)}$; $p_v = \delta - \alpha_{(\ell_v-1)}$, and; $\ell_v = \ell_v - 1$.
```

Fig. 3. Algorithm for reporting the path from node v to the root of T.

```
Algorithm $ReportLayerPath(\ell_v, p_v)$

1. Load block $b_v$ containing $p_v$. Scan $B_e$ (the tree's representation) to locate $p_v$. If $b_v$ is stored in a
   superblock, $SB_v$, then load $SB_v$'s first block if $b_v$ is not the first block in $SB_v$. Let $\texttt{min}(D_p)$ be the
   minimum valid preorder number of $b_v$'s duplicate path (let $\texttt{min}(D_p) = 1$ if $b_v$ is a leading block), and
   let $\texttt{min}(SB_{D_p})$ be the minimum valid preorder number of the superblock duplicate path (if $b_v$ is the
   first block in $SB_v$ then let $\texttt{min}(SB_{D_p}) = 0$).
2. Traverse the path from $p_v$ to a root in $B_e$. If $r$ is the preorder number (within $B_e$) of a node on this
   path report $(r-1) + \texttt{min}(D_p) + \texttt{min}(SB_{D_p})$. This step terminates at a root in $B_e$. Let $r_k$ be the rank
   of this root in the set of roots of $B_e$.
3. Scan the root-to-path array, $R_p$ from $\tau B...1$ to find the smallest $i$ such that $R_p[i] \geq r_k$. If $r_k \geq R_p[1]$
   then $r$ is on the top level in the layer so return $(r-1) + \texttt{min}(D_p) + \texttt{min}(SB_{D_p})$ and terminate.
4. Set $j = i - 1$.
5. $\texttt{while}(j \geq 1$ and $D_p[j] \neq 1)$ report $D_p[j] + \texttt{min}(SB_{D_p})$ and set $j = j - 1$.
6. If $j \geq 1$ then report $SB_{D_p}[j]$ and set $j = j - 1$ $\texttt{until}(j < 1)$.
```

Fig. 4. Steps to execute traversal within a layer, ℓ_v, starting at the node with preorder number p_v. This algorithm reports the nodes visited and returns the layer preorder number of the root at which it terminates.

```
Algorithm $FindBlock(\ell_v, p_v)$

1. Find $\sigma$, the disk block containing $\ell_v$'s leading block. $\sigma = \texttt{rank}_1(\mathcal{B}_l, \ell_v)$.
2. Find $\alpha$, the rank of $\ell_v$'s leading block within $\sigma$, by performing $\texttt{rank}/\texttt{select}$ operations on $\mathcal{B}_l$ to find
   $j \leq \ell_v$ such that $\mathcal{B}_l[j] = 1$. $\alpha = p_v - j$.
3. Scan $\sigma$ to find, and load, the data for $\ell_v$'s leading block (may required loading the next disk block).
   Note the size $\delta$ of the leading block.
4. If $p_v \leq \delta$ then $p_v$ is in the already loaded leading block, terminate.
5. Calculate $\omega$, the rank of the block containing $p_v$ within the $\texttt{select}_1(\mathcal{B}_f, \ell_v + 1) - \texttt{select}_1(\mathcal{B}_f, \ell_v)$ full
   blocks for this level.
6. Load full block $\texttt{rank}_0(\mathcal{B}_f, \ell_v) + \omega$ and terminate.
```

Fig. 5. *FindBlock* algorithm.

Lemmas 4 and 5 lead to the following theorem. To simplify our space result we define one additional term $\epsilon = 12\tau$.

Theorem 1. *A tree T on N nodes can be represented in $2N + \frac{\epsilon N}{\log_B N} + o(N)$ bits such that given a node-to-root path of length K, the path can be reported in $O(K/B)$ I/Os, when $0 < \epsilon < 1$.*

For the case in which we wish to maintain a key with each node we have the following corollary, when a key can be encoded with $q = O(\lg N)$ bits.

Corollary 1. *A tree T on N nodes with q-bit keys can be represented in $(2+q)N + q \cdot \left[\frac{6\tau N}{\lceil \log_B N \rceil} + \frac{2\tau q N}{\lceil \lg N \rceil} + o(N)\right]$ bits such that given a node-to-root path of length K, that path can be reported in $O(\tau K/B)$ I/Os, when $0 < \tau < 1$.*

Proof. To store the tree structure and keys we now requires $2N + qN = (2+q)N$ bits. The number of nodes per block and superblock also changes such that each block now stores no more than:

$$\frac{B \lg N - W_b}{2+q} < \frac{B \lg N}{q} \tag{10}$$

nodes.

For the duplicate path array, each entry must store a q-bit key but the number of entries per superblock decreases due to the smaller number of nodes per block. The space requirement for the duplicate path becomes:

$$\begin{aligned} &= \tau B \left(\left\lceil \lg \left(\frac{B \lceil \lg B \rceil \lceil \lg N \rceil}{q} \right) \right\rceil + q \right) \\ &= \tau B (\lceil \lg B \rceil + \lceil \lg \lceil \lg B \rceil \rceil + \lceil \lg \lceil \lg N \rceil \rceil - \lg q + q) \end{aligned} \tag{11}$$

or for the superblock duplicate path

$$\tau B (\lceil \lg N \rceil + q) \tag{12}$$

bits.

The size of the root-to-path array becomes:

$$\tau B \lg \left(\frac{B \lg N}{q} \right) = \tau B (\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil - \lg q) \tag{13}$$

bits.

Replacing the sizes of these data structures from the non-key case in Eq. 5 yields the following per block redundancy in the q-bit key case.

$$\begin{aligned} W_b = \tau B (\lceil \log_B N \rceil + q) + 3\tau B \lceil \lg B \rceil + 3\tau B \lceil \lg \lceil \lg N \rceil \rceil \\ + \tau B \lceil \lg \lceil \lg B \rceil \rceil + \tau B q - 2 \lg q \end{aligned} \tag{14}$$

From Eq. 6 we can calculate the total redundancy as follows:

$$\frac{(2+q)N \cdot 2W_b}{B \lceil \lg N \rceil} = \frac{(2qN + 4N) \cdot W_b}{B \lceil \lg N \rceil} \tag{15}$$

Finally substituting the value for W_b from Eq. 14 we obtain the following result:

$$\begin{aligned} &= \frac{(2qN+4N)\tau B (\lceil \log_B N \rceil + q)}{B \lceil \lg N \rceil} + \frac{(6qN + 12N)\tau B \lceil \lg B \rceil}{B \lceil \lg N \rceil} \\ &+ \frac{(6qN+12N)\tau B \lceil \lg \lceil \lg N \rceil \rceil}{B \lceil \lg N \rceil} + \frac{(6qN+12N)\tau B \lceil \lg \lceil \lg B \rceil \rceil}{B \lceil \lg N \rceil} \\ &+ \frac{(2qN+4N)\tau B q}{B \lceil \lg N \rceil} - \frac{(4qN+8N)\tau B \lg q}{B \lceil \lg N \rceil} \end{aligned}$$

$$= O\left(\frac{(2qN+4N)\tau}{\lceil \lg B \rceil}\right) + \frac{(6qN+12N)\tau}{\lceil \log_B N \rceil}$$
$$+ \frac{(6qN+12N)\tau \lceil \lg \lceil \lg N \rceil \rceil}{\lceil \lg N \rceil} + \frac{(6qN+12N)\tau \lceil \lg \lceil \lg B \rceil \rceil}{\lceil \lg N \rceil}$$
$$+ \frac{(2qN+4N)\tau q}{\lceil \lg N \rceil} - \frac{(4qN+8N)\tau \lg q}{\lceil \lg N \rceil} \tag{16}$$

The constant c is added to the first term since $q = O(\lg N)$. All terms except the second and fifth are $q \cdot o(N)$ so we can summarize the space complexity of the redundancy as:

$$q \cdot \left[\frac{6\tau N}{\lceil \log_B N \rceil} + \frac{2\tau q N}{\lceil \lg N \rceil} + o(N) \right] \tag{17}$$

Adding this space to the $(2+q)N$ bits required to store the tree structure and keys gives the total space complexity. $\square$

In Corollary 1 it is obvious that the first and third terms are small so we will consider the size of the the second term inside the brackets $((2\tau q N)/\lceil \lg N \rceil)$. When $q = o(\lg N)$ this term becomes $o(N)$. When $q = \Theta(\lg N)$ we can select τ such that this term becomes (ηN) for $0 < \eta < 1$.

4 Top Down Traversal

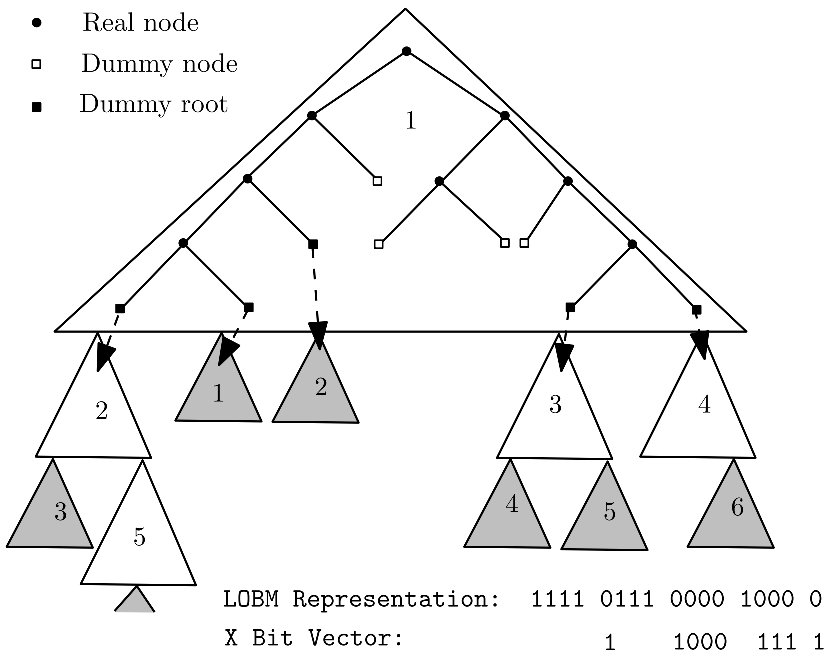
Given a binary tree T, in which every node is associated with a key, we wish to traverse a top-down path of length K starting at the root of T and terminating at some node $v \in T$. Let A be the maximum number of nodes that can be stored in a single block, and let $q = O(\lg N)$ be the number of bits required to encode a single key. Keys are included in the top-down case because it is assumed that the path followed during the traversal is selected based on the key values in T.

4.1 Data Structures

We begin with a brief sketch of our data structures. A tree T is partitioned into subtrees, where each subtree T_i is laid out into a *tree block*. Each block contains a succinct representation of T_i and the set of keys associated with the nodes in T_i. The edges in T that span a block boundary are not explicitly stored within the tree blocks. Instead, they are encoded through a set of bit vectors (detailed later in this section) that enable navigation between blocks.

To introduce our data structures, we give some definitions. If the root node of a block is the child of a node in another block, then the first block is a *child* of the second. There are two types of blocks: *internal* blocks that have one or more *child* blocks, and terminal blocks that have no *child* blocks. The *block level* of a block is the number of blocks along a path from the root of this block to the root of T.

We number the internal blocks in the following manner. First number the block containing the root of T as 1, and number its child blocks consecutively from left to right. We then consecutively number the internal blocks at each successive block level (see Fig. 6). The internal blocks are stored on the disk in an array I, such that the block numbered j is stored in entry $I[j]$.



● Real node
□ Dummy node
■ Dummy root
1
2
1
2
3
4
3
5
4
5
6
LOBM Representation: 1111 0111 0000 1000 0
X Bit Vector: 1 1000 111 1

Fig. 6. Numbering of internal (hallow triangles) and external (shaded triangles) blocks for T. The structure of T within internal block 1 is also shown. The dashed arrows indicate the parent-child relationship between dummy roots in internal block 1 and their child blocks. Finally, the LOBM representation for internal block 1 and the corresponding bits in bit vector X are shown at the bottom. Bits in the X bit vector have been spaced such that they align with the their corresponding 0 bits (the dummy nodes/roots) in the LOBM representation.

Terminal blocks are numbered and stored separately. Starting again at 1, they are numbered from left to right. Terminal blocks are stored in the array Z. As terminal blocks may vary in size, there is no one-to-one correspondence between disk and tree blocks in Z; rather, the tree blocks are packed into Z to minimize wasted space. At the start of each disk block j, a $\lg B$-bit *block offset* is stored which indicates the position of the starting bit of the first terminal block stored in $Z[j]$. Subsequent terminal blocks are stored immediately following the last bits of the previous terminal blocks. If there is insufficient space to record a terminal block within disk block $Z[j]$, the remaining bits are stored in $Z[j+1]$.

We now describe how an individual internal tree block is encoded. Consider the block of subtree T_j; it is encoded using the following structures:

1. The block keys, B_k, is an A-element array which encodes the keys of T_j.
2. The tree structure, B_s, is an encoding of T_j using the LOBM sequence of Jacobson [1]. More specifically, we define each node of T_j as a *real* node. T_j is then augmented by

adding *dummy* nodes as the left and/or right child of any real node that does not have a corresponding real child node in T_j. The dummy node may, or may not, correspond to a node in T, but the corresponding node is not part of T_j. We then perform a level order traversal of T_j and output a 1 each time we visit a real node, and a 0 each time we visit a dummy node. If T_j has A nodes the resulting bit vector has A 1s for real nodes and $A+1$ 0s for dummy nodes. Observe that the first bit is always 1, and the last two bits are always 0s, so it is unnecessary to store them explicitly. Therefore, B_s can be represented with $2A-2$ bits.

3. The *dummy offset*, B_d. Let Γ be a total order over the set of all dummy nodes in internal blocks. In Γ the order of dummy node d is determined first by its block number, and second by its position within B_s. The dummy offset records the position in Γ of the first dummy node in B_s.

The encoding for terminal blocks is identical to internal blocks except: the dummy offset is omitted, and the last two 0s of B_s are encoded explicitly.

We now define a *dummy root*. Let T_j and T_k be two tree blocks where T_k is a child block of T_j. Let r be the root of T_k, and v be r's parent in T. When T_j is encoded, a dummy node is added as a child of v which corresponds to r. Such a dummy node is termed a dummy root.

Let ℓ be the number of dummy nodes over all internal blocks. We create three bit arrays:

1. $X[1..\ell]$ stores a bit for each dummy node in internal blocks. Set $X[i]=1$ iff the i^{th} dummy node in Γ is the dummy root of an internal block.
2. $S[1..\ell]$ stores a bit for each dummy node in internal blocks. Set $S[i]=1$ iff the i^{th} dummy node in Γ is the dummy root of a terminal block.
3. $S_B[1..\ell']$, where ℓ' is the number of 1s in S. Each bit in this array corresponds to a terminal block. Set $S_B[j]=1$ iff the corresponding terminal block is stored starting in a disk block of Z that differs from that in which terminal block $j-1$ starts.

4.2 Block Layout

We have yet to describe how T is split up into *tree* blocks. This is achieved using the two-phase blocking strategy of Demaine *et al.* [10]. Phase one blocks the first $c\lg N$ levels of T, where $0<c<1$. Starting at the root of T the first $\lfloor \lg(A+1)\rfloor$ levels are placed in a block. Conceptually, if this first block is removed, we are left with a forest of $O(A)$ subtrees. The process is repeated recursively until $c\lg N$ levels of T have thus been blocked.

In the second phase we block the rest of the subtrees by the following recursive procedure. The root, r, of a subtree is stored in an empty block. The remaining $A-1$ capacity of this block is then subdivided, proportional to the size of the subtrees, between the subtrees rooted at r's children. During this process, if at a node the capacity of the current block is less than 1, a new block is created. To analyze the space costs of our structures, we have the following lemma.

Lemma 6. *The data structures described above occupy $(3+q)N+o(N)$ bits.*

Proof. We first determine the maximum block size A. In our model a block stores at most $B \lg N$ bits. The encoding of the subtree T_j requires $2A$ bits, we need Aq bits to store the keys, and $\lceil \lg N \rceil$ bits to store the dummy offset. We therefore have the following equation: $2A + Aq + \lceil \lg N \rceil = \lfloor B \lg N \rfloor$. Thus, number of nodes stored in a single block satisfies:

$$\begin{aligned}
B \lg N - 1 < 2A + A \cdot q + \lceil \lg N \rceil &\leq B \lg N \\
B \lg N - \lceil \lg N \rceil - 1 < A(q+2) &\leq B \lg N - \lceil \lg N \rceil \\
\frac{B \lg N - \lceil \lg N \rceil - 1}{q+2} < A &\leq \frac{B \lg N - \lceil \lg N \rceil}{q+2} \\
A &= \Theta\left(\frac{B \lg N}{q}\right)
\end{aligned} \tag{18}$$

During the first phase of the layout, non-full internal blocks may be created. However, the height of the phase 1 tree is bounded by $c \lg N$ levels, so the total number of wasted bits in such blocks is bounded by $o(N)$.

The arrays of blocks I and Z store the structure of T using the LOBM succinct representation which requires $2N$ bits. The dummy roots are duplicated as the roots of child blocks, but as the first bit in each block need not be explicitly stored, the entire tree structure still requires only $2N$ bits. We also store N keys which require $N \cdot q$ bits. Each of the $O(N/A)$ blocks in I stores a block offset of size $\lg(N/A)$ bits. The total space required for the offsets is $N/A \cdot \lg(N/A)$, which is $o(N)$ bits since $q = O(\lg N)$. The bit vectors X and S_B have size N, but in both cases the number of 1 bits is bounded by N/A. By Lemma 1b, we can store these vectors in $o(N)$ bits. The number of 1 bits in X is bounded by $N/2$. By lemma 1a it can be encoded by $N + o(N)$ bits. The total space is thus $(3+q)N + o(N)$ bits. □

4.3 Navigation

Navigation in T is summarized in Figures 7 and 8 which show the algorithms $Traverse(key, i)$ and $TraverseTerminalBlock(key, i)$ respectively. During the traversal the function `compare`(key) compares the value *key* to the key of a node to determine which branch of the tree to traverse. The parameter i is the number of a *disk* block. Traversal is initiated by calling $Traverse(key, 1)$.

Lemma 7. *For a tree T laid out in blocks and represented by the data structures described above, a call to TraverseTerminalBlock can be performed in $O(1)$ I/Os, while Traverse can be executed in $O(1)$ I/Os per recursive call.*

Proof. Internal blocks are traversed by the algorithm $Traverse(key, i)$ in Fig. 7. Loading the block in step 1 requires a single I/O, while steps 2 through 4 are all performed in main memory. Steps 5 and 6 perform lookups and call `rank` on X and S, respectively. This requires a constant number of I/Os. Step 7 requires no additional I/Os.

The algorithm $TraverseTerminalBlock(key, i)$ is executed at most once per traversal. The look-up and `rank` require only a single I/O. The only step that might cause problems

```
Algorithm $Traverse(key, i)$

1. Load block $I[i]$ to main memory. Let $T_i$ denote the subtree stored in $I[i]$.
2. Scan $B_s$ to navigate within $T_i$. At each node $x$ use compare$(key, B_k[x])$ to determine which branch
   to follow until a dummy node $d$ with parent $p$ is reached.
3. Scan $B_s$ to determine $j = \texttt{rank}_0(B_s, d)$.
4. Determine the position of $j$ with respect to $\Gamma$ by adding the dummy offset to calculate $\lambda = B_d + j$.
5. If $X[\lambda] = 1$, then set $i = \texttt{rank}_1(X, \lambda)$ and call $Traverse(key, i)$.
6. If $X[\lambda] = 0$ and $S[\lambda] = 1$, then set $i = \texttt{rank}_1(S, \lambda)$ and call $TraverseTerminalBlock(key, i)$.
7. If $X[\lambda] = 0$ and $S[\lambda] = 0$, then $p$ is the final node on the traversal, so the algorithm terminates.
```

Fig. 7. Top down searching algorithm for a blocked tree.

```
Algorithm $TraverseTerminalBlock(key, i)$

1. Load disk block $Z[\lambda]$ containing terminal block $i$, where $\lambda = \texttt{rank}_1(S_B, i)$.
2. Let $B_d$ be the offset of disk block $Z[\lambda]$.
3. Find $\alpha$, the rank of terminal block $i$ within $Z[\lambda]$ by scanning from $S_B[i]$ backwards to find $j \leq i$ such
   that $S_B[j] = 1$. Then $\alpha = i - j$.
4. Starting at $B_d$ scan $Z[\lambda]$ to find the start of the $\alpha^{\text{th}}$ terminal block. Recall that each block stores
   a bit vector $B_s$ in the LOBM encoding, so we can determine when we have reached the end of one
   terminal block as follows:
   (a) Set two counters $\mu = \beta = 1$. $\mu$ records the number of 1 bits (real nodes) encountered in $B_s$ during
       the scan, while $\beta$ records the excess of 1 to 0 bits (dummy nodes) encountered. Both are initially
       set to 1 as the root node of the block is implied.
   (b) Scan $B_s$. When a 1 bit is encountered increment $\mu$ and $\beta$. When a 0 bit is encountered decrement
       $\beta$. Terminate the scan when $\beta < 0$ as the end of $B_s$ has been reached.
   (c) Now $\mu$ records the number of nodes in the terminal block so calculate the length of the array $B_k$
       needed to store the keys and jump ahead this many bits. This will place the scan at the start of
       the next terminal block.
5. Once the $\alpha^{\text{th}}$ block has been reached, the terminal block can be read in (process is the same as
   scanning the previous blocks). It may be the case the this terminal block overruns the disk block $Z[\lambda]$
   into $Z[\lambda + 1]$. In this case skip the first $\lceil \lg B \rceil$ bits of $Z[\lambda + 1]$ and continue reading in the terminal
   block.
6. With the terminal block in memory, the search can be concluded in a manner analogous to that for
   internal blocks except that once a dummy node is reached, the search terminates.
```

Fig. 8. Performing search for a terminal block.

is step 3 in which the bit array S_B is scanned. Note that each bit in S_B corresponds to a terminal block stored in Z. The terminal block corresponding to i is contained in $Z[\lambda]$, and the terminal block corresponding to j also starts in $Z[\lambda]$. A terminal block is represented by at least $2 + q$ bits. As blocks in S_B are of the same size as in Z, we cross at most one block boundary in S_B during the scan. $\square$

The I/O bounds are then obtained directly by substituting our succinct block size A for the standard block size B in Lemma 3. Combined with Lemmas 6 and 7, this gives us the following result:

Theorem 2. *A rooted binary tree, T, of size N, with keys of size $q = O(\lg N)$ bits, can be stored using $(3 + q)N + o(n)$ bits so that a root to node path of length K can be reported with:*

1. $O\left(\frac{K}{\lg(1+(B \lg N)/q)}\right)$ I/Os, when $K = O(\lg N)$

2. $O\left(\frac{\lg N}{\lg(1+\frac{B\lg^2 N}{qK})}\right)$ *I/Os, when* $K = \Omega(\lg N)$ *and* $K = O\left(\frac{B\lg^2 N}{q}\right)$*, and*

3. $O\left(\frac{qK}{B\lg N}\right)$ *I/Os, when* $K = \Omega\left(\frac{B\lg^2 N}{q}\right)$.

When key size is constant the above result leads to the following corollary.

Corollary 2. *Given a rooted binary tree,* T*, of size* N*, with keys of size* $q = O(1)$ *bits,* T *can be stored using* $3N + o(n)$ *bits in such a manner that a root to node path of length* K *can be reported with:*

1. $O\left(\frac{K}{\lg(1+(B\lg N))}\right)$ *I/Os when* $K = O(\lg N)$

2. $O\left(\frac{\lg N}{\lg(1+\frac{B\lg^2 N}{K})}\right)$ *I/Os when* $K = \Omega(\lg N)$ *and* $K = O\left(B\lg^2 N\right)$*, and*

3. $O\left(\frac{K}{B\lg N}\right)$ *I/Os when* $K = \Omega(B\lg^2 N)$.

Corollary 2 shows that, in the case where the number of bits required to store each search key is constant, our approach not only reduces storage space, but also improves the I/O efficiency. For the case where $K = \Omega(B\lg N)$ and $K = O(B\lg^2 N)$ the number of I/Os required in Lemma 3 is $\Theta(K/B) = \Omega(\lg N)$ while that required in the corollary is $O\left(\frac{\lg N}{\lg(1+\frac{B\lg^2 N}{K})}\right) = O(\lg N)$.

5 Conclusions

We have presented two new data structures that are both I/O efficient and succinct for bottom-up and top-down traversal in trees. Our bottom-up result applies to trees of arbitrary degree while out top-down result applies to binary trees. In both cases the number of I/Os is asymptotically optimal.

Our results lead to several open problems. Our top-down technique is valid for only binary trees. Whether this approach can be extended to trees of bounded degree is an open problem. For the bottom-up case it would be interesting to see if the asymptotic bound on I/Os can be improved from $O(K/B)$ to something closer to $O(K/A)$ I/Os, where A is the number of nodes that can be represented succinctly in a single block. In both the top-down and bottom-up cases several `rank` and `select` operations are required to navigate between blocks. These operations use only a constant number of I/Os, and it would be useful to reduce this constant factor. This might be achieved either by reducing the number `rank` and `select` operations used in the algorithms, or by demonstrating how the bit arrays could be interleaved to guarantee a low number of I/Os per block.

References

1. Guy Jacobson. Space-efficient static trees and graphs. *FOCS*, 42:549–554, 1989.
2. Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *DCC*, pages 252–261, 2008.

3. Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
4. Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
5. Pankaj K. Agarwal, Lars Arge, T. M. Murali, Kasturi R. Varadarajan, and Jeffrey Scott Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking (extended abstract). In *SODA*, pages 117–126, 1998.
6. David A Hutchinson, Anil Maheshwari, and Norbert Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126:55–82, 2003.
7. David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *SODA*, pages 383–391, 1996.
8. Joseph Gil and Alon Itai. How to pack trees. *J. Algorithms*, 32(2):108–132, 1999.
9. Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010 [cs:DS], 2004.
10. Erik D. Demaine, John Iacono, and Stefan Langerman. Worst-case optimal tree layout in a memory hierarchy. arXiv:cs/0410048v1 [cs:DS], 2004.
11. Gerth StBrodal and Rolf Fagerberg. Cache-oblivious string dictionaries. In *SODA*, pages 581–590, 2006.
12. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.
13. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
14. David Benoit, Erik D. Demaine, J.Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.