Succinct and I/O Efficient Data Structures for Traversal in Trees*

Craig Dillabaugh[1], Meng He[2], and Anil Maheshwari[1]

[1] School of Computer Science, Carleton University, Canada
[2] Cheriton School of Computer Science, University of Waterloo, Canada
Corresponding author: Meng He, Cheriton School of Computer Science, University of Waterloo,
200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1
Phone: 1-519-888-4567 x37824 Fax: 1-519-885-1208

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 a tree on N nodes and traverses a path to the root. For a bottom-up path of length K, we design data structures that permit traversal of this path in $O(K/B)$ I/Os (B denotes the size of a disk block) 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 the tree in $(3 + q)N + o(N)$ 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 is a building block in 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 [2]. 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.* [3] 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.

* The preliminary version of this paper appeared in Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008) [1]. This work was supported by NSERC of Canada. The work was done when the second author was in School of Computer Science, Carleton University, Canada.

1.1 Previous Work

The I/O model [4] splits memory into two levels: the fast but finite internal memory, and the slow but infinite EM. Data are 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.* [5] 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.* [6]. 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.* [7] examined the case of bottom-up traversal, where the path begins with some node in the tree and proceeds to the root. They gave a blocking that supports bottom-up traversal in $O(K/B)$ I/Os when the tree is stored in $O(N/B)$ blocks (see Lemma 2). The case of top down traversal has been studied more extensively. Clark and Munro [8] described 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 [9] presented a blocking layout that yields the minimum expected number of I/Os on a root to leaf path. In the cache oblivious model where the sizes of blocks and internal memory are unknown, Alstrup *et al.* [10] 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.* [11] 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 [12] 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)[3]. 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,

[3] In this paper, $\lg N$ denotes $\lg_2 N$. When another base is used, we state it explicitly (i.e. $\log_B N$).

where K is the path length and $0 < \epsilon < 1$. This technique is based on [7], and achieves an improvement on the space bound by a factor of $\lg N$.

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 [11]. We further show that, when q is constant, 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 [2, 8, 13] 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 [2] and Clark and Munro [8], while part (b) is from Raman *et al.* [13]:

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 [2, 14, 15] have been proposed to represent a binary tree (or ordinal tree) in $2N + o(N)$ bits, while supporting efficient navigation. Jacobson [2] 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. 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 after we visit all its descendants.

Based on this, Munro and Raman [14] designed a succinct representation of an ordinal tree of N nodes in $2N + o(N)$ bits, 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.* [7] 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 ([7]). *A rooted tree T on N nodes 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.

To support top-down traversal, Demaine *et al.* [11] 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 top $c \lg N$ 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 ([11]). *A binary tree T on N nodes can be stored in $O(N/B)$ blocks on disk such that 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. Given the bottom up nature of the queries, there is no need to check a node's key value while traversing, since the path always proceeds to the current node's parent. Thus, we first consider trees whose nodes do not store key values, and then show how to apply our techniques to encode trees on nodes with keys.

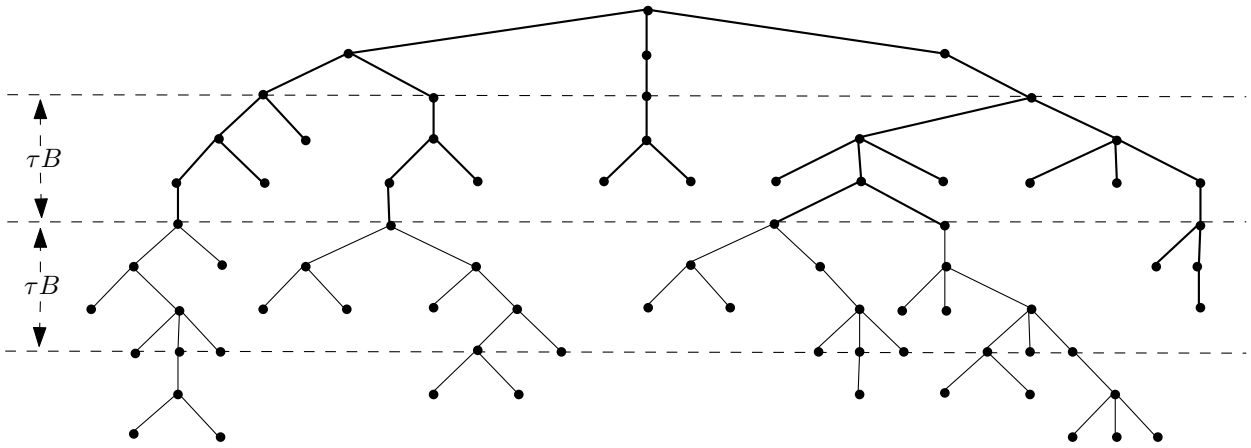

Fig. 1. An example of partitioning a tree into layers.

3.1 Blocking Strategy

Our blocking strategy is inspired by [7]; we have modified their technique and introduced new notation. We first give an overview of our approach. 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 desired distribution of nodes. See Figure 1 for an example. We then group the nodes of each layer into *tree blocks*[4], 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 disk 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. As stated in the previous paragraph, we have the flexibility to choose an arbitrary level within the first τB levels as the top of the second layer. Given this flexibility, we can prove the following lemma:

Lemma 4. *There exists a division of T into layers such that the total number of nodes on the top level of layers is bounded by $\lceil N/(\tau B) \rceil$.*

Proof. There are τB different ways to divide T into layers. Let s_i denote the total number of nodes on the top level of layers under the i^{th} way of dividing T, and let S_i denote the set of such nodes. We observe that given a node of T that is not the root, there is only one value of i such that this node is in S_i, while the root of T appears once in each S_i. Therefore, we have

[4] A tree block is a portion of the tree, which is different from the notion of disk block. A tree block is not necessarily stored in a disk block, either. In the rest of the paper, when the context is clear, we may use the term block to refer to either a tree block or a disk block.

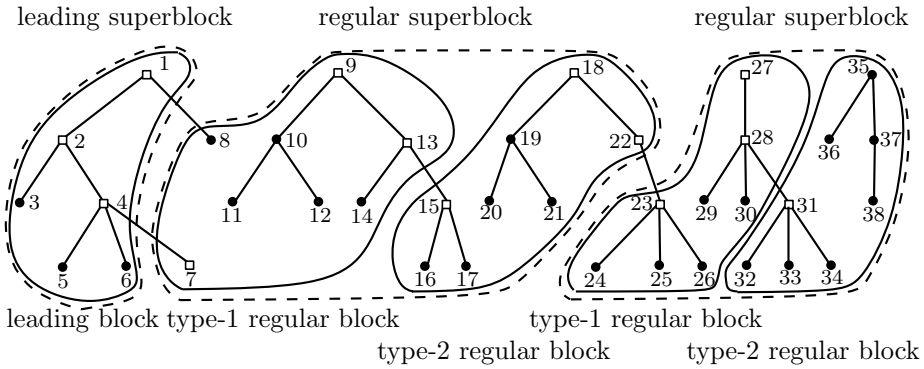

Fig. 2. Blocking within a layer (Layer L_3 in Figure 1) is shown, along with block (solid lines) and superblock (dashed lines) boundaries. Numbers shown are preorder values in the layer. Nodes on the duplicate paths are indicated by a hollow square.

$\sum_{i=1}^{\tau B} s_i = N - 1 + \tau B$. Then $\min s_i \leq \lfloor (N - 1 + \tau B)/(\tau B) \rfloor \leq \lfloor N/(\tau B) + 1 - 1/(\tau B) \rfloor \leq \lceil N/(\tau B) \rceil$. □

Thus, we pick the division of T into layers that ensures the total number of nodes on the top level of layers to be bounded by $\lceil N/(\tau B) \rceil$. 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 number the nodes of the remaining subtrees from left to right. Once the nodes of L_i are numbered, they are grouped into tree blocks of consecutive preorder number. We term the first tree block in a layer the *leading block*, and the remaining tree blocks in a layer *regular blocks*. Each superblock except possibly the first one in a layer, which we term the *leading superblock*, contains exactly $\lfloor \lg B \rfloor$ tree blocks (see Figure 2). We term each of the remaining superblock a *regular superblock*.

The sizes of tree blocks are dependent on the approach we use to store them in the disk. When storing tree blocks in external memory, we treat leading blocks and regular blocks differently. We use a disk block to store a regular block along with the representation of its duplicate path, or the superblock duplicate path if it is the first tree block in a superblock. In such a disk block, we refer to the space used to store the duplicate path as *redundancy* of the disk block. For simplicity, such space is also referred to as the redundancy of the regular tree block stored in this disk block. In our succinct tree representation, we require two bits to represent each node in the subtrees of L_i. Therefore, if a disk block of $B \lg N$ bits has redundancy W, the maximum number of nodes that can be stored in it is:

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

Layers are blocked in such a manner that when a regular block is stored in a disk block, it has maximum number of nodes as computed above, and the leading block is the only block permitted to have fewer nodes than any regular block. As the sizes of leading blocks can be arbitrarily small, we pack them into a sequence of disk blocks. There are two types of regular

blocks: a *type-1 regular block* is the first block in a regular superblock, while a *type-2 regular block* is a regular block that is not the first block in its superblock. In our representation, the redundancy in each disk block that stores a type-1 or type-2 regular block is fixed. Therefore, the number, A_1, of nodes in a type-1 regular block and the number, A_2, of nodes in a type-2 regular block are both fixed. To divide a layer into tree blocks, it suffices to know the values of A_1 and A_2 (we give these values in Lemma 5). More precisely, we first compute the number, s, of nodes in a regular superblock using $s = A_1 + (\lfloor \lg B \rfloor - 1)A_2$. Let l_i be the number of nodes in layer L_i. We then put the last $s\lfloor l_i/s \rfloor$ nodes into $\lfloor l_i/s \rfloor$ superblocks, each of which can be easily divided into tree blocks. Finally, the first $l'_i = l_i \bmod s$ nodes are in the leading superblock, in which the first $l'_i \bmod A_2$ nodes form the leading block.

We select as a tree block's (or a superblock's) duplicate path the path from the parent of the node with minimum preorder number in the block (or superblock) to the layer's top level. In the example in Figure 2, the duplicate path of the second block consists of nodes $7, 4, 2, 1$. A duplicate path has at most τB nodes and satisfies the following property (this is analogous to Property 3 in Hutchinson *et al.* [7]):

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

Proof. Let v be the node with the minimum preorder number in Y, and in the forest stored in Y, let T_v be the subtree that contains v. For example, in Figure 2, if Y is the fourth block, then T_v is the subtree consisting of nodes 23, 24, 25 and 26. As the preorder number of v is smaller than that of any other node in T_v, we conclude that node v is the root of T_v. For the case in which $x \in T_v$, because the path from x to v is entirely within T_v, and v is on the duplicate path of Y, our claim is true.

The second case is $x \notin T_v$. In the forest stored in Y, let T_x be the subtree that contains x, and let node y be its root. If y is at the top level of its layer, as the path from x to y contains entirely of nodes in Y, the lemma follows directly. Thus we need only consider the case where y is not at the top level of this layer, and it suffices to prove that the parent, z, of y is on the duplicate path of Y. Assume to the contrary that z is not (i.e. $z \neq v$ and z is not v's ancestor). As the preorder number of z is smaller than that of y, z is in a block (or superblock), Z, that is to the left of Y. Therefore, the preorder number of z is smaller than that of v. As v is not a descendant of z, by the definition of preorder traversal, the preorder number of v is larger than any node in the subtree rooted at z including y, which is a contradiction. □

3.2 Data Structures

Each tree block is encoded by three data structures:

1. An encoding of the tree structure, denoted B_e. The subtree(s) contained within the block are encoded as a sequence of balanced parentheses (see Section 2.2). 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 opening 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. For example, the fourth block in Figure 2 is encoded as (()())(()())).

2. The duplicate path array, $D_p[1..j]$, for $1 < j \leq \tau B$. Let v be the node with the smallest preorder number in the block. Entry $D_p[j]$ stores node at the j^{th} level on the path from v to the top level of the layer. 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). To identify each node in D_p, there are three cases:
   (a) The block is a leading block. Then v is the only node in the duplicate path. Thus, for a leading block, we do not store D_p.
   (b) The block is a type-1 regular block. For such a block, D_p stores the preorder numbers of the nodes with respect to the preorder numbering in the layer. For example, in Figure 2, the duplicate path array for the second block stores $1, 2, 4, 7$.
   (c) The block is not a leading block. In this case, each node in D_p is identified by its preorder number with respect to the block's superblock. Then the duplicate path array for the third block in Figure 2 stores $3, 7, 9, 0$.
3. The root-to-path array, $R_p[1..j]$, where $1 < j \leq \tau B$, for each regular block. A regular block may include subtrees whose roots are not at the top level of the layer. Among them, the root of the leftmost subtree is on the duplicate path, and by Property 1, the parents of the roots of the rest of all such subtrees are on the duplicate path of the block. R_p is constructed to store such information, in which $R_p[j]$ stores the number of subtrees whose roots are either on the duplicate path or have parents on the duplicate path from level τB up to level j. The number of subtrees whose roots are children of the node stored in $D_p[j]$ can be calculated by evaluating $R_p[j] - R_p[j+1]$, if $D_p[j]$ is not the node with the smallest preorder number in the regular block. For example, in Figure 2, the content of the root-to-path array for the second block is $2, 1, 1, 1$.

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 by navigating 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 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 no root in this layer with a smaller preorder number has the same parent.

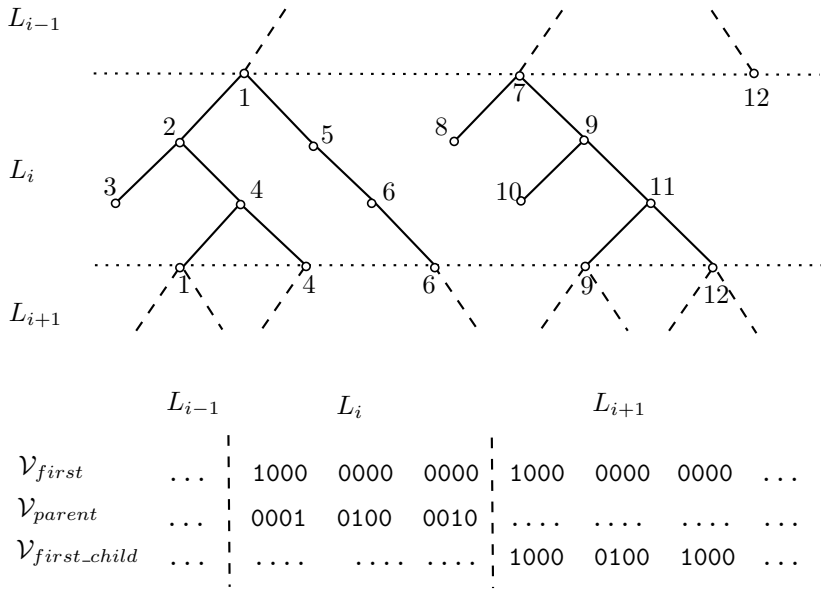

Fig. 3. Scheme for mapping between layers. The dashed horizontal lines indicate the top level of each layer. The bottom part of the figure shows the corresponding portions of the bit vectors used to maintain the mapping between layer L_i and its neighbouring layers.

Figure 3 demonstrates how the three bit vectors represent the mapping between nodes in different layers.

All leading blocks are packed together on disk. 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, a leading block may overrun the boundary of a block on disk. We use the first $\lceil \lg (B\lceil \lg N \rceil) \rceil$ bits of each disk block to store an offset that indicates the position of the starting bit of the first leading block inside this 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 leading blocks, and the second indexes regular blocks. Let x be the number of layers on T, and let z be the total number of regular 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 $\mathcal{B}_r[1..(x+z)]$ that encodes the number of regular blocks in each layer in unary. More precisely, $\mathcal{B}_r[1..(x+z)] = 0^{l_1}10^{l_2}10^{l_3}1\ldots$, where l_i is the number of regular blocks in layer i.

With the details of data structures given, we can now determine the number of nodes in a type-1 or type-2 regular block.

Lemma 5. *To use the approach in Section 3.1 to divide a layer into blocks, it is sufficient to choose:*

1. $A_1 = \frac{(1-\tau)B\lceil \lg N \rceil - \tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil)}{2}$, *and*

2. $A_2 = \frac{B\lceil \lg N\rceil - \tau B(2\lceil \lg B\rceil + 2\lceil \lg \lceil \lg N\rceil\rceil + \lceil \lg \lceil \lg B\rceil\rceil)}{2}$.

Proof. By Equation 1, to choose appropriate values for A_1 and A_2, we need only compute the redundancy of a type-1 regular block and a type-2 regular block, respectively. Thus, we consider the space required to store D_p and R_p. To compute the space required for D_p, there are two cases:

1. For a type-1 regular block, D_p stores preorder values with respect to the layer preorder numbering. There can be as many as N nodes in a layer, so each entry requires $\lceil \lg N\rceil$ bits. The total space for D_p is thus $\tau B\lceil \lg N\rceil$.
2. For each node in a duplicate path of a type-2 regular block, we store its preorder number in the superblock. 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 D_p requires the following number of bits:

$$\begin{aligned}\tau B\left\lceil \lg\left(\frac{B\lceil \lg B\rceil\lceil \lg N\rceil}{2}\right)\right\rceil &\le \tau B\left(\lg\left(\frac{B\lceil \lg B\rceil\lceil \lg N\rceil}{2}\right)+1\right)\\ &= \tau B(\lceil \lg B\rceil + \lceil \lg\lceil \lg B\rceil\rceil + \lceil \lg\lceil \lg N\rceil\rceil) \qquad (2)\end{aligned}$$

As a block may have as many as $B\lceil \lg N\rceil/2$ nodes, each entry in R_p can be encoded in $\lceil \lg B\rceil + \lceil \lg\lceil \lg N\rceil\rceil$ 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 type-1 or type-2 regular block.

The redundancy of a type-1 regular block includes space required to encode both D_p and R_p, which is $\tau B(\lceil \lg N\rceil + \lceil \lg B\rceil + \lceil \lg\lceil \lg N\rceil\rceil)$ bits.

For a type-2 regular block, the number of bits used to store both D_p and R_p is:

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

The results in the lemma can then be proved by substituting the redundancy W in Equation 1 by the redundancy of a type-1 or type-2 regular block as computed above. □

To analyze the space costs of our data structures, we have the following lemma:

Lemma 6. *The data structures described in this section occupy* $2N + \frac{8\tau N}{\log_B N} + o(N)$ *bits.*

Proof. First consider the number of bits used to store the actual tree structure of T (i.e. the total space used by all the B_e's). The balanced parentheses encoding requires $2N$ bits, and each node of T is contained in one and only one block. Hence the structure of T is encoded using $2N$ bits.

Next consider the total space required for all the duplicate paths and root-to-path arrays. By the proof of Lemma 5, the sum of the redundancy of all the blocks in a regular superblock is:

$$\tau B(\lceil \lg N \rceil + \lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil) \\ +(\lceil \lg B \rceil - 1)\tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \quad (4)$$

The average redundancy per block in a regular superblock is then:

$$\begin{aligned} \overline{W} &= \frac{\tau B\lceil \lg N \rceil}{\lceil \lg B \rceil} + \frac{\tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil)}{\lceil \lg B \rceil} \\ &\quad + \frac{(\lceil \lg B \rceil - 1)\tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil)}{\lceil \lg B \rceil} \\ &< \frac{\tau B(\lg N + 1)}{\lg B} + \frac{\tau B(\lceil \lg B \rceil + \lceil \lg \lceil \lg N \rceil \rceil)}{\lceil \lg B \rceil} \\ &\quad + \tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \\ &\quad - \frac{\tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil)}{\lceil \lg B \rceil} \\ &< \tau B \log_B N + \tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil) \end{aligned} \quad (5)$$

A regular block in a leading superblock is a type-2 regular block, and its redundancy is given by Inequality 3, which is smaller than $\overline{W}$. Therefore, we use $\overline{W}$ to bound the average redundancy of a regular block. The total number of blocks required to store T is then at most $\frac{N}{\lfloor (B\lceil \lg N \rceil - \overline{W})/2 \rfloor} < \frac{N}{(B\lceil \lg N \rceil - \overline{W})/2 - 1}$. Then the total size, R, of the redundancy for T is:

$$R < 2N \cdot \frac{\overline{W}}{B\lceil \lg N \rceil - \overline{W} - 2} < 2N \cdot \frac{2\overline{W} + 2}{B\lceil \lg N \rceil} = \frac{4N\overline{W}}{B\lceil \lg N \rceil} + o(N) \quad (6)$$

when $\overline{W} < \frac{1}{2}B\lceil \lg N \rceil - 1$. By Inequality 5, this condition is true if:

$$\begin{aligned} B\lceil \lg N \rceil &> 2\tau B \log_B N + \tau B(4\lceil \lg B \rceil + 4\lceil \lg \lceil \lg N \rceil \rceil + 2\lceil \lg \lceil \lg B \rceil \rceil) + 2 \\ \lceil \lg N \rceil - 2/B &> 2\tau \log_B N + 4\tau\lceil \lg B \rceil + 4\tau\lceil \lg \lceil \lg N \rceil \rceil + 2\tau\lceil \lg \lceil \lg B \rceil \rceil) \end{aligned} \quad (7)$$

For any possible values of B and N in practice, the inequality $\lceil \lg N \rceil/4 > 2/B$ (i.e. $B\lceil \lg N \rceil > 8$) holds. Thus to guarantee that Inequality 7 is true, it suffices to have:

$$\frac{3}{4}\lceil \lg N \rceil > 2\tau \log_B N + 4\tau\lceil \lg B \rceil + 4\tau\lceil \lg \lceil \lg N \rceil \rceil + 2\tau\lceil \lg \lceil \lg B \rceil \rceil \quad (8)$$

Noting that each term on the right hand side of Inequality 8 consists of a constant, τ, and an expression less than $\lceil \lg N \rceil$, Inequality 8 is true if $\frac{3}{4}\lceil \lg N \rceil \geq 12\tau\lceil \lg N \rceil$. Therefore, by choosing an appropriate value for τ such that $\tau \leq \frac{1}{16}$, we can ensure that Inequality 6 holds. We then substitute for $\overline{W}$ in Inequality 6, to obtain the following:

$$
\begin{aligned}
R &< \frac{4N(\tau B \log_B N + \tau B(2\lceil \lg B \rceil + 2\lceil \lg \lceil \lg N \rceil \rceil + \lceil \lg \lceil \lg B \rceil \rceil))}{B\lceil \lg N \rceil} + o(N) \\
&= \frac{4N\tau \log_B N}{\lceil \lg N \rceil} + \frac{8N\tau \lceil \lg B \rceil}{\lceil \lg N \rceil} + \frac{8N\tau \lceil \lg \lceil \lg N \rceil \rceil}{\lceil \lg N \rceil} + \frac{4N\tau \lceil \lg \lceil \lg B \rceil \rceil}{\lceil \lg N \rceil} + o(N) \\
&< \frac{4N\tau \log_B N}{\lceil \lg N \rceil} + \frac{8N\tau (\lg B + 1)}{\lg N} + \frac{8N\tau \lceil \lg \lceil \lg N \rceil \rceil}{\lceil \lg N \rceil} + \frac{4N\tau \lceil \lg \lceil \lg B \rceil \rceil}{\lceil \lg N \rceil} + o(N) \\
&= \frac{8\tau N}{\log_B N} + o(N) \qquad (9)
\end{aligned}
$$

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

When packed on the disk, each leading block requires an offset of $\lceil \lg (B\lceil \lg N \rceil) \rceil$ bits. As the number of leading blocks is $\lceil N/\tau B \rceil$, such information requires $o(N)$ bits in total.

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 $\lceil N/(\tau B) \rceil$ 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, which is bounded by $\lceil N/(\tau B) \rceil$ as shown in Lemma 4. 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 $\lceil N/(\tau B) \rceil$. By Lemma 1b, each of three bit vectors requires at most $\left\lceil \lg \binom{N}{N/(\tau B)} \right\rceil + O(N \lg \lg N / \lg N) = o(N)$ bits.

Finally, we consider the bit vectors $\mathcal{B}_l$ and $\mathcal{B}_r$. $\mathcal{B}_l$ 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 $\lceil N/(\tau B) \rceil$ bits. Thus, it can be represented using $o(N)$ bits. The number of 1's in $\mathcal{B}_r$ is equal to the number of layers, which is $\lceil N/(\tau B) \rceil$. The number, a, of 0's in $\mathcal{B}_r$ is equal to the total number of regular blocks. As there are less than $2N$ nodes in all the regular blocks, and each regular block contains a non-constant number of nodes, we have $a = o(N)$. Therefore, the length of $\mathcal{B}_r$ is $o(N)$, which can be stored in $o(N)$ bits using Lemma 1a.

Summing up, our data structures require $2N + \frac{8\tau N}{\log_B N} + o(N)$ bits in total. □

3.3 Navigation

The algorithm for reporting a node-to-root path is given by algorithms $ReportPath(T, v)$ (see Figure 4) and $ReportLayerPath(\ell_v, p_v)$ (see Figure 5). 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 $FindBlock(\ell_v, p_v)$ described in Figure 6. We now have the following lemma.

Lemma 7. *The algorithm ReportPath traverses a path of length K in T in $O(K/\tau B)$ I/Os.*

Proof. In each layer we progress τB steps toward the root of T. To do so, we must load the disk 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.

As a path of length K has nodes in $\lceil K/\tau B \rceil$ layers, and to traverse the path, the number of I/Os required in each layer and between two consecutive layers is constant as show above, we conclude that it takes $O(K/\tau B)$ I/Os to traverse the path. □

Algorithm *ReportPath*(T, v)

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

Fig. 4. Algorithm for reporting the path from node v to the root of T.

Lemmas 6 and 7 lead to the following theorem. To simplify our space result, we define one additional term $\epsilon = 8\tau$.

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, for any constant number ϵ such that $0 < \epsilon < 1$.*

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.

Algorithm *ReportLayerPath*(ℓ_v, p_v)

1. Load block b_v containing p_v by calling $FindBlock(\ell_v, p_v)$. Scan B_e (the tree's representation) to locate p_v. Let SB_v be the superblock containing b_v, and 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 τB to 1 to find the largest 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. 5. 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.

Corollary 1. *A tree T on N nodes with q-bit keys can be represented in $(2+q)N + q \cdot \left\lceil \frac{4\tau N}{\log_B N} + \frac{2\tau q N}{\lg N} + o(N) \right\rceil$ 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. We use the same strategy as the one used for the case in which no keys are associated with nodes. We also construct similar data structures (the only difference is that the duplicate path array D_p for each block now also stores the keys of the nodes in the duplicate path) and use the same algorithms to traverse a path. The challenge is to analyze the space cost.

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

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

nodes, where W is the redundancy.

For the duplicate path array D_p, 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. For a type-2 regular block, the space requirement (in bits) of its duplicate path array D_p becomes:

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

Algorithm *FindBlock*(ℓ_v, p_v)

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

Fig. 6. *FindBlock* algorithm.

For a type-1 regular block, D_p (recall that D_p stores the superblock duplicate path in this case) uses:

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

bits.

The size of the root-to-path array R_p 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 Inequality 5 yields the following per block redundancy in the q-bit key case.

$$\overline{W} < \tau B(\log_B N + q) + 2\tau B\lceil \lg B \rceil + 2\tau B\lceil \lg \lceil \lg N \rceil \rceil + \tau B\lceil \lg \lceil \lg B \rceil \rceil \tag{14}$$

From Inequality 6 we can bound the total redundancy, R, as follows:

$$R < \frac{(2+q)N \cdot 2\overline{W}}{B\lceil \lg N \rceil} + o(N) = \frac{(2qN + 4N) \cdot \overline{W}}{B\lceil \lg N \rceil} + o(N) \tag{15}$$

Finally substituting the value for $\overline{W}$ from Inequality 14 we obtain the following result:

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

$$+\frac{(2qN+4N)\tau Bq}{B\lceil \lg N\rceil}+o(N) \tag{16}$$

$$=O\left(\frac{(2qN+4N)\tau}{\lceil \lg B\rceil}\right)+\frac{(4qN+8N)\tau(\lg B+1)}{\log N}$$
$$+\frac{(4qN+8N)\tau\lceil \lg\lceil \lg N\rceil\rceil}{\lceil \lg N\rceil}+\frac{(4qN+8N)\tau\lceil \lg\lceil \lg B\rceil\rceil}{\lceil \lg N\rceil}$$
$$+\frac{(2qN+4N)\tau q}{\lceil \lg N\rceil}+o(N) \tag{17}$$

The first term above is obtained using the equation $q = O(\lg N)$. All terms except the second and fifth are $q \cdot o(N)$, so we can summarize the space complexity of the redundancy as:

$$R < q \cdot \left[\frac{4\tau N}{\log_B N}+\frac{2\tau qN}{\lg N}+o(N)\right] \tag{18}$$

The space analysis in Lemma 6 on all the other data structures also applies here, so they occupy $o(N)$ bits. Adding this space and R to the $(2+q)N$ bits required to store the tree structure and keys gives the total space complexity. □

In Corollary 1 it is obvious that the first and third terms inside the brackets are small, so we need only consider the size of the the second term, i.e. $(2\tau qN)/\lg N$. 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 stored at nodes 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 tree 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 tree blocks.

To introduce our data structures, we give some definitions. If the root node of a tree block is the child of a node in another block, then the first block is a *child* of the second.

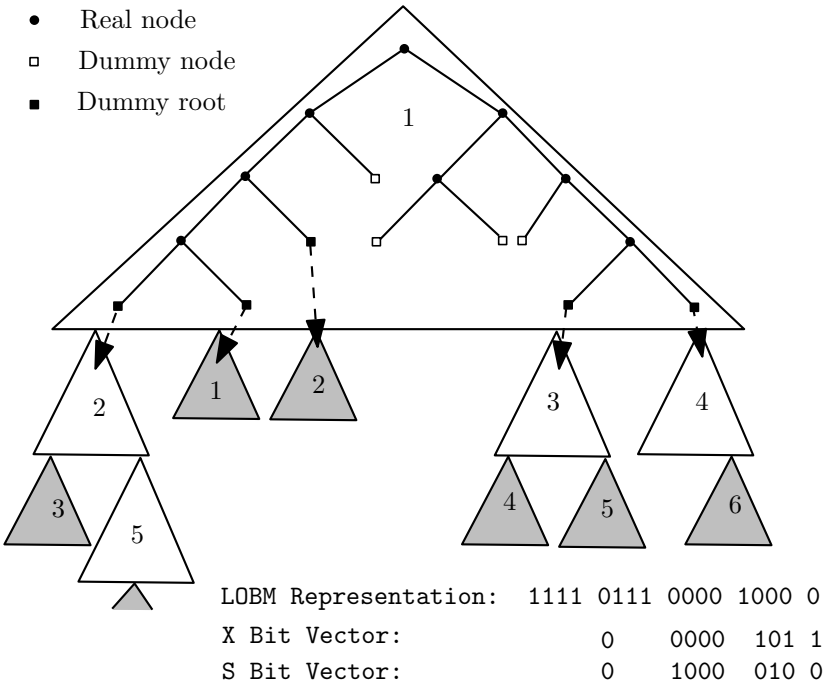

Fig. 7. Numbering of internal (hallow triangles) and terminal (shaded triangles) blocks for T. The structure of T within internal block 1 is also shown. The dashed arrows indicate the parent-child relationship between dummy roots in internal block 1 and their child blocks. Finally, the LOBM representation for internal block 1 and the corresponding bits in bit vectors X and S are shown at the bottom. Bits in bit vectors X and S have been spaced such that they align with the their corresponding 0 bits (the dummy nodes/roots) in the LOBM representation.

There are two types of tree 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 Figure 7). The internal blocks are stored on the disk in an array I of disk blocks, such that the tree block numbered j is stored in entry $I[j]$.

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 $\lceil \lg (B \lceil \lg N \rceil) \rceil$-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 [2]. 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 that 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 the one 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.* [11]. 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 8. *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 $B\lceil \lg N \rceil$ bits. The encoding of the subtree T_j requires $2A$ bits. We also need Aq bits to store the keys, and $\lceil \lg N \rceil$ bits to store the dummy offset. We therefore have the following equation: $2A + Aq + \lceil \lg N \rceil \leq B\lceil \lg N \rceil$. Thus, number of nodes stored in a single block satisfies:

$$A \leq \frac{(B-1)\lceil \lg N \rceil}{q+2} \tag{19}$$

Therefore, we choose $A = \lfloor \frac{(B-1)\lceil \lg N \rceil}{q+2} \rfloor$ to partition the tree. Thus:

$$A = \Theta\left(\frac{B \lg N}{q}\right) \tag{20}$$

During the first phase of the layout, a set of 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 these 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. The block offsets stored in Z and the dummy offsets stored for internal blocks require $o(N)$ bits in total. The bit vectors X and S_B have size at most $N+1$, 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. Bit vector X can be encoded in $N + o(N)$ bits using Lemma 1a. The total space is thus $(3+q)N + o(N)$ bits. □

4.3 Navigation

Navigation in T is summarized in Figures 8 and 9 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 9. *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 Figure 8. 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 $\texttt{compare}(key, B_k[x])$ 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 Γ 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. 8. Top down searching algorithm for a blocked tree.

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. □

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 8 and 9, 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* $K = \Omega(B\lg^2 N)$.

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 α, the rank of terminal block i within $Z[\lambda]$ by scanning from $S_B[i]$ backwards to find the largest $j \leq i$ such that $S_B[j] = 1$. Set $\alpha = i - j$.
4. Starting at B_d, scan $Z[\lambda]$ to find the start of the α^{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 $\mu = \beta = 1$; μ records the number of 1 bits (*real* nodes) encountered in B_s during the scan, while β 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 μ and β. When a 0 bit is encountered decrement β. Terminate the scan when $\beta < 0$ as the end of B_s has been reached.
   (c) Now μ 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 α^{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. 9. Performing search for a terminal block.

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 Corollary 2 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 larger degrees 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 disk 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 by reducing the number of `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. Dillabaugh, C., He, M., Maheshwari, A.: Succinct and I/O efficient data structures for traversal in trees. In: ISAAC. (2008) 112–123
2. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. (1989) 549–554
3. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: DCC. (2008) 252–261
4. Aggarwal, A., Jeffrey, S.V.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9) (1988) 1116–1127
5. Nodine, M.H., Goodrich, M.T., Vitter, J.S.: Blocking for external graph searching. Algorithmica **16**(2) (1996) 181–214
6. Agarwal, P.K., Arge, L., Murali, T.M., Varadarajan, K.R., Vitter, J.S.: I/O-efficient algorithms for contour-line extraction and planar graph blocking (extended abstract). In: SODA. (1998) 117–126
7. Hutchinson, D.A., Maheshwari, A., Zeh, N.: An external memory data structure for shortest path queries. Discrete Applied Mathematics **126** (2003) 55–82
8. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA. (1996) 383–391
9. Gil, J., Itai, A.: How to pack trees. J. Algorithms **32**(2) (1999) 108–132
10. Alstrup, S., Bender, M.A., Demaine, E.D., Farach-Colton, M., Rauhe, T., Thorup, M.: Efficient tree layout in a multilevel memory hierarchy. **arXiv:cs.DS/0211010 [cs:DS]** (2004)
11. Demaine, E.D., Iacono, J., Langerman, S.: Worst-case optimal tree layout in a memory hierarchy. **arXiv:cs/0410048v1 [cs:DS]** (2004)
12. Brodal, G.S., Fagerberg, R.: Cache-oblivious string dictionaries. In: SODA. (2006) 581–590
13. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4) (2007) 43:1–43:25
14. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3) (2001) 762–776
15. Benoit, D., Demaine, E.D., Munro, J., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica **43**(4) (2005) 275–292