Compact Dictionaries for Variable-Length Keys and Data, with Applications

DANIEL K. BLANDFORD
Google Inc.
and
GUY E. BLELLOCH
Carnegie Mellon University

We consider the problem of maintaining a dynamic dictionary T of keys and associated data for which both the keys and data are bit strings that can vary in length from zero up to the length w of a machine word. We present a data structure for this variable-bit-length dictionary problem that supports constant time lookup and expected amortized constant time insertion and deletion. It uses $O(m+3n-n\log_2 n)$ bits, where n is the number of elements in T, and m is the total number of bits across all strings in T (keys and data). Our dictionary uses an array $A[1\ldots n]$ in which locations store variable-bit-length strings. We present a data structure for this variable-bit-length array problem that supports worst-case constant-time lookups and updates and uses $O(m+n)$ bits, where m is the total number of bits across all strings stored in A.

The motivation for these structures is to support applications for which it is helpful to efficiently store short varying length bit strings. We present several applications, including representations for semi-dynamic graphs, order queries on integers sets, cardinal trees with varying cardinality, and simplicial meshes of d dimensions. These results either generalize or simplify previous results.

Categories and Subject Descriptors: E.2 [**Data Storage Representations**]: —*Hash-table representations*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Compression

1. INTRODUCTION

There has been significant recent interest in data structures that use near optimal space while supporting fast access [Jacobson 1989; Munro and Raman 2001; Chuang et al. 1998; Brodnik and Munro 1999; Pagh 2001; Raman et al. 2002; Blandford et al. 2003; Raman and Rao 2003; Fotakis et al. 2005; Grossi and Vitter 2005]. In addition to theoretical interest, such structures have significant practi-

This work was supported in part by the National Science Foundation as part of the Aladdin Center (`www.aladdin.cmu.edu`) under grants ACI-0086093, CCR-0085982, and CCR-0122581. This work was carried out while the first author was at Carnegie Mellon University.
Authors' addresses: Blandford, Google, 1600 Amphitheatre Parkway, Mountain View, CA 94043, email: `dan.blandford@gmail.com`; Blelloch, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15232, email: `guyb@cs.cmu.edu`.

cal implications. In experimental work [Blandford et al. 2004], for example, it has been demonstrated that a compact representation of graphs not only requires significantly less space than standard representations (*e.g.*, adjacency lists), but can also be faster. This is because the representation requires less data to be loaded into the cache.

Given a universe U of keys, the dictionary problem is to support lookup (membership) queries on a set $T \subset U$. Often we also want to associate *satellite data* with each key. If the data comes from a universe U' the dictionary can be described as a set of key-value pairs $T \subset U \times U'$, with the condition that keys are unique. We say that a dictionary is static if it only supports lookups and dynamic if it also supports insertions and deletions. In this paper we assume the unit-cost RAM with standard arithmetic and logical operations. We allow for any word length $w > \log_2 \mathcal{M}$ bits, where $\mathcal{M}$ is the size of the memory in bits.

In many situations it is useful to store variable-length strings in a dictionary. This problem is fundamental and has been studied for years. Previous solutions, however, are not space efficient when the average number of bits (or characters) in each string is small. The standard implementations of tries, for example, are neither time nor space efficient: on n entries they require $\Omega(n \log n)$ bits and use $\Omega(\log n)$ time to look up an element of length w.

In this paper we are interested in space-efficient dynamic dictionaries for storing variable-length bit strings. We use S_w to denote the set of bit strings up to length w (*i.e.*, $S_w = \epsilon \cup \{0,1\}^1 \cup \{0,1\}^2 \cup \ldots \{0,1\}^w$). The *variable-bit-length dictionary* (VLD) problem is the dictionary problem where $T \subset S_w \times S_w$. We present a data structure for this problem that supports lookups in $O(1)$ time, insertions and deletions in $O(1)$ expected amortized time, and uses $O(m+3n-n\log_2 n)$ bits, where $n = |T|$, and $m = \sum_{(s,t)\in T}(|s| + |t|)$. Since the keys are unique, $m > n \log_2 n - 2n$. The results can be extended to bit strings of arbitrary length by storing bit strings longer than w separately using known techniques—*e.g.*, using vector hashing [Carter and Wegman 1979] with dynamic perfect hash tables [Dietzfelbinger et al. 1994]. This would involve replacing $O(1)$ with $O((|t| + |s|)/w)$ in the time bounds.

Our dictionary structure makes use of a data structure for the simpler *variable-bit-length array* (VLA) problem. The problem is to maintain an array $A[1 \ldots n]$ of locations each storing an element from S_w while supporting lookups and updates on any of the n locations. We present a data structure for the problem that supports lookups and updates in $O(1)$ worst-case time, and uses $O(m + n)$ bits, where $m = \sum_{i=1}^{n} |A[i]|$.

Any representation for a collection type with l possible values requires at least $\log_2 l$ bits to distinguish the values. This is often referred to as the information-theoretic lower bound[1]. Consider a collection type C parameterized on some size parameters, including the number of elements n. Given an information-theoretic lower bound as a function of the size parameters, $I(n, \ldots)$, we say a data structure for storing the collection type is *compact* if it uses at most $O(I(n, \ldots) + n)$ bits and *succinct* if it uses at most $I(n, \ldots) + o(I(n, \ldots))$ bits. In both cases we require that the operations on the data structure are efficient. The data structures presented in this paper for variable-bit-length arrays and dictionaries are compact

[1]Technically this is the information-theoretic lower bound assuming all values are equally likely.

when parameterized on the number of elements and the total number of bits across all elements.

1.1 Related Work

For fixed-length keys, space-efficient solutions of the dictionary problem have received much attention. The information-theoretic lower bound for representing n elements from a universe U is $B = \log_2\lceil\binom{|U|}{n}\rceil = n\log_2(|U|/n) + O(n)$ bits. For bit strings of fixed length l, $|U| = 2^l$ so $B = nl - n\log_2 n + O(n)$. Cleary [1984] described a compact data structure for fixed-length keys that used $(1+\epsilon)B + O(n)$ bits with $O(1/\epsilon^2)$ expected time for lookup and insertion while allowing satellite data. His structure used the technique of *quotienting* [Knuth 1973], which involves storing only part of each key in the hash table; the part not stored can be inferred from the position that the key was initially hashed to.

Brodnik and Munro [1999] described a succinct data structure for static dictionaries supporting $O(1)$ time lookups. It did not support satellite data. Pagh [2001] extended this result to allow for satellite data and improve the space bound in the lower-order term. Raman and Rao [2003] described a succinct dynamic dictionary that supports lookup in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time. The structure allows attaching fixed-length satellite data. This previous research did not consider variable-bit-length keys or data.

Cohen and Matias [2003] described a dynamic "string-array index" that can keep track of variable-bit-length strings. This is similar to the VLA problem, but is more limited in that their application made updates to bit strings in random locations (determined by a set of hash functions), and only required increasing the length of a bit string by one bit at a time. Under these assumptions, they proved bounds similar to ours. They did not consider the general case.

1.2 Difference coding and Applications

Our main motivation is to use variable-bit-length dictionaries as building blocks for various other applications. We describe applications of variable-bit-length dictionaries to directed graphs, cardinal trees with nodes of varying cardinality, ordered sets, and simplicial meshes. These applications either generalize or simplify previously known results.

All the applications are based on simple "pointer based" data structures, but replace pointers with difference codes. A *difference code* encodes a value by specifying the difference between that value and another known value instead of encoding it directly. For example the value of a pixel in an image might be encoded using the difference from the previous pixel. If the difference is small it can be encoded in a small number of bits, but the difference will vary requiring a different number of bits for different pixels. In all applications we describe, the average difference is small, but the differences can vary significantly—hence the importance of efficiently storing variable-bit-length strings. Typically the differences we use are differences between integer labels given to elements, such as integer labels on the vertices of a graph.

For directed graphs we describe a data structure that supports adjacency queries, listing the neighbors of a vertex, and deleting and inserting edges. Queries take $O(1)$ time, and updates take $O(1)$ expected amortized time. For a graph with n vertices

with integer labels bounded by $O(n)$ the representation uses $O(n+\sum_{(u,v)\in E}\log|u-v|)$ bits. Any graph from a class satisfying an $n^{1-\epsilon}$ edge-separator theorem ($\epsilon > 0$) can be labeled with integers $[1,\ldots,n]$ so that $\sum_{(u,v)\in E}\log|u-v| < kn$ for some constant k [Blandford et al. 2003], and hence can be coded in $O(n)$ bits. It is well known, for example, that the class of bounded-degree planar graphs satisfies an $n^{1/2}$ edge-separator theorem. For graphs with bounded degree this extends previous results [Turán 1984; Keeler and Westbrook 1995; He et al. 2000; Blandford et al. 2003] by permitting insertion and deletion of edges. We say that the graph is partially dynamic since although it allows dynamic insertions and deletions, the space bound relies on $\sum_{(u,v)\in E}\log|u-v|$ remaining small.

For an ordered set $S \subset \{0,\ldots,m\}$ of size n we describe a compact data structure that supports insertion and deletion in $O(1)$ expected amortized time, and finger searching in $O(\log l)$ time, where l is the number of keys between the finger and the key that is found. It also supports generating a finger in $O(\log n)$ time. The representation uses $O(n\log(m/n))$ bits. These bounds match those reported in [Blandford and Blelloch 2004], except that the updates are expected amortized time here and are worst case time bounds in the previous work. The structure we describe here, however, is simpler and quite different from the previous structure and allows for attaching a satellite bit string to each key. We also show how the structure can be used to support the findNext operation (finding the next key in S larger than a given key k) in $O(\log\log m)$ time with the same memory bounds.

For cardinal trees (also known as tries) we describe a structure that supports trees in which each node can have a different cardinality. Queries can request the k^{th} child, or the parent of any node, and updates can add or delete the k^{th} child if it is a leaf. Queries take $O(1)$ worst-case time and updates take $O(1)$ expected amortized time. Again we can attach satellite bit strings to each node. Using an appropriate labeling of the vertices the structure uses $O(\sum_{v\in V}\log c(p(v)))$ bits, which is asymptotically optimal. This generalizes previous results on cardinal trees [Benoit et al. 2005; Raman et al. 2002] to varying cardinality. We do not match their optimal constant in the first order term.

For d-dimensional simplicial (triangulated) meshes we describe a data structure that supports insertion and deletion of simplices of dimension d, and returning the neighbors across all faces of dimension $d-1$. For example, in a 3-dimensional tetrahedral mesh one can add and delete tetrahedrons, and ask for the neighboring tetrahedron across any of the four faces. For a mesh with n vertices with integer labels bounded by $O(n)$ the representation uses $O(n+\sum_{f\in F} d\max_{a,b\in f}\log|a-b|)$ bits, where F is the set of $d-1$ dimensional simplices (faces) in the mesh. In three dimensions we show that this simplifies to $O(n+\sum_{(a,b)\in E}(\log|a-b|))$ where E is the set of edges in the mesh. Updates take $O(1)$ expected amortized time and queries take $O(1)$ worst case time. We used a similar data structure as described here to implement triangulated and tetrahedral meshes [Blandford et al. 2005].

The remainder of the paper is organized as follows. Section 2 describes some preliminary concepts that will be useful. Section 3 describes the array structure that is a building block for our dictionary, and Section 4 describes the dictionary itself. Sections 5–8 discuss the applications of our dictionary structure.

2. PRELIMINARIES

2.1 Processor model

We assume the unit-cost RAM with $w > \log \mathcal{M}$, where w is the length of a machine word and $\mathcal{M}$ is the size of the memory in bits. We assume the processor supports standard logical and arithmetic operations on words including integer multiplication and division (needed by the hash functions), and bit shifting. We also use two special operations on words, `bitSelect` and `bitRank`, defined as follows. Given a machine word interpreted as a bit string $s[1, \ldots, w]$, $\texttt{bitSelect}(s, i)$ returns the least bit position j such that $|\{s[k] = 1 | k \in [1, \ldots, j]\}| = i$, and $\texttt{bitRank}(s, j)$ returns $|\{s[i] = 1 | i \in [1, \ldots, j]\}|$. These operations mimic the function of the `rank` and `select` data structures of Jacobson [1989]. If the processor does not support these operations, they can be implemented in constant time and within the needed memory bounds using table lookup (see Section 2.5).

2.2 Quotienting

Quotienting based hash schemes take advantage of the fact that when using hashing, the bucket number to which a key hashes gives information about the key. In particular for n buckets, since only about $1/n$ elements from the universe U hash to that bucket, the bucket gives $\log n$ bits of information about the key—*i.e.*, it narrows down the set of possible keys by a factor of n. For a key with l bits (2^l possible values) we can therefore reconstruct it by storing only $l - \log n$ bits in the bucket and using these bits along with the bucket number to reconstruct the key. This idea was observed by Knuth [1973, Section 6.4, exercise 13] and has been used by several others [Cleary 1984; Pagh 2001; Raman and Rao 2003; Fotakis et al. 2005]. We believe the term quotienting was suggested by Pagh [2001].

A quotienting scheme must supply three functions:

(1) a hash function $h(x)$ that maps each key to a bucket,
(2) a quotient function $g(x)$ that maps each key to a quotient to be stored in the bucket, and
(3) an inverse function $i(h, g)$ that takes the bucket and quotient and returns the original key.

Various quotienting schemes have been suggested by the authors listed above. Since it is convenient for us to use a scheme that works on keys of different lengths, we use a variant in this paper.

2.3 Gamma codes

Throughout the paper we use gamma codes [Elias 1975] to encode integers. The gamma code is a variable-length prefix code that represents a positive integer v with $\lfloor \log v \rfloor$ zeroes, followed by the $(\lfloor \log v \rfloor + 1)$-bit binary representation of v, for a total of $2\lfloor \log v \rfloor + 1$ bits.

Given a string s containing a gamma code (of length $\leq w$) for an integer d followed possibly by other information, it is possible to decode the gamma code in constant time. First, the decoder uses $\texttt{bitSelect}(s, 1)$ to find the location j of the first 1 in s. The length of the gamma code is $2j + 1$, so the decoder uses shifts to extract the first $2j + 1$ bits of s. These bits are the binary code for d. To

encode negative (or zero) as well as positive integers a sign bit can be stored with the gamma code (before or after).

Gamma codes are only one of several variable-length codes which use $O(\log n)$ bits to represent a positive integer n.

2.4 Memory allocation

There are various models for analyzing memory usage. In the paper we assume the number of words used by a computation at any point in time is one more than the location of the last word of memory currently being used. The number of bits $\mathcal{B} < \mathcal{M}$ is w times the number of words. In our algorithm descriptions, however, we assume all memory is allocated in contiguous *regions* of memory of arbitrary sizes specified by the user (integer number of words), and that these regions can be freed by the user at a later time. We say a region is *live* if it has been allocated but not freed. It is convenient to analyze space in terms of the total number of bits across all live regions, which we denote as $\mathcal{B}_m$. Since the live regions might be scattered $\mathcal{B}$ might be much larger than $\mathcal{B}_m$. However, using standard techniques (*e.g.*, [Baker 1978]) it is possible to implement a memory management scheme so that $\mathcal{B} \in O(\mathcal{B}_m)$, all allocations take $O(1)$ time, and all frees take $O(b)$ time, where b is the size of the freed region in words. We outline how such a scheme can be implemented here.

The idea is to maintain two memories, and to allocate in one of these, the *active memory*, until it becomes too sparse, at which point it is copied and compacted into the other memory. The two memories can be implemented in one memory by interleaving the words. Regions are allocated one after the other in the active memory, and each contains a header with its length, a forwarding pointer (used during copying) and a flag indicating whether the region is allocated or freed. Pointers to regions are somehow marked as a pointer—*e.g.*, using one bit in the word. We define the *density* as the total size of live regions (in words), divided by the last position of the last allocated region. The goal is to bound the density below by a constant, which implies $\mathcal{B} \in O(\mathcal{B}_m)$ as long as the second memory is no bigger than the first.

Copying is executed incrementally. It starts when the density goes below some threshold $\alpha < 1$, and finishes when all live regions are copied to the new memory. The copying happens in two passes over the active memory. The first pass scans the active memory from region to region using the lengths to find the next region. It allocates a location in the new memory for each live region it encounters, and places this location in the forwarding pointer of the region. As with the active memory, regions are allocated one after the other in the new memory. The second pass copies the data from the live regions to their new locations. Note that whenever a pointer is copied, the value of the pointer that is written in new memory needs to refer to the correct location in the new memory instead of the old memory. This can be found through the forwarding pointer—hence the need for two passes.

While copying, the user continues to work in the active memory. Each read is taken from the active memory. Each allocation is performed in both memories returning a pointer to the active copy to the user, and placing a pointer to the new copy in the forwarding pointer of the active copy. Each write is executed in just the active memory during the first pass of copying, and in both memories during the

second. When writing pointers in the new memory, the forwarded pointer needs to be used. When the copying finishes, any pointers in the registers are updated to point to their forwarded pointers, and the new memory becomes the active memory.

Every free of a region of size b is responsible for executing kb steps of the copy procedure. This is required to make sure the memory does not get freed faster than it is copied. Setting k appropriately large will guarantee that by the time the copying is finished, the density is still bounded below by a constant.

In Section 3 we will need to free non-constant size regions in constant time. This can be done in the above scheme under the following bounded-release condition. We say a region of size b is *busy* after it is freed but before $\Omega(b)$ instructions are accounted towards that free. Every instruction can only be accounted towards one busy region and only after the region is freed. The *bounded-release* condition requires that at all times the total size of busy regions is no more than a constant factor larger than the total size of the live regions. This condition will allow freeing in constant time while maintaining the $\mathcal{B} \in O(\mathcal{B}_m)$ bound since (a) the instructions that are accounted towards the free can each be responsible for executing a constant number of steps of the copy procedure, and (b) the total number of frees which are not yet accounted for (are busy) affects the density by at most a constant factor.

2.5 Table lookup

Table lookup can be used to implement the `bitSelect` and `bitRank` operations if they are not supported by the machine. We also use table lookup in the perfect-hashing version of the variable-bit-length dictionary (Section 4.3). In both cases the table lookup can support the required operations on b bits in $O(1/\epsilon)$ time using a table containing at most $2^{\epsilon b}$ bits, $0 < \epsilon \leq 1$. The tables can be shared among any number of dictionaries.

One way to account for the memory needed for the table is to set $b = w$ and explicitly include the time and space cost for the table in our bounds. If $w \in \Theta(\log \mathcal{B})$ then a constant ϵ can be selected so that the table size is smaller than $\mathcal{B}$ and therefore only affects the space bound by at most a factor of two.

If, however, we want to allow for larger w and don't want to pay $O(2^{\epsilon w})$-bits for a table we can instead use a virtual word size w_e, where $w_e \in \Theta(\log \mathcal{B})$. By packing virtual words into machine words, operations on these virtual words can easily be simulated in constant time and with no loss in space. All the data structures described in the paper can use words of this size internally. However, we need a way to store user data and keys with length l where $w_e < l \leq w$. This can be implemented using a level of indirection where we allocate any string longer than w_e in the global memory pool and store a w_e-bit pointer to the word in the data structure. In the structures described in the next two sections the items that could need to be stored in this way are the data fields of the variable-bit-length array and hash table, and the quotient in the variable-bit-length hash table.

Using this technique, the table size can be kept to $2^{\epsilon O(\log \mathcal{B})}$ bits and by picking ϵ appropriately, the space of the table can be made smaller than $\mathcal{B}$, asymptotically. As the table grows and shrinks, w_e might have to change dynamically and all structures will have to be rebuilt. If only applied when the memory size changes by a constant factor the cost of resizing w_e can be amortized against the cost of other operations in the same way as resizing the hash tables can be.

3. VARIABLE-BIT-LENGTH ARRAYS

The *array* problem is to support lookup and updating the locations of an array $A[1 \ldots n]$, where each location stores an element from some universe U'. The size n is fixed when the array is created. Arrays are supported by just about every programming language and are trivial to implement on the RAM when the elements are of equal size.

The *variable-bit-length array* (VLA) problem is the array problem where the universe is S_w. We define the *total bits* of A as $m = \sum_{i=1}^{n} |A[i]|$. We describe a data structure for the problem that supports lookups and updates in $O(1)$ worst-case time and uses $O(m+n)$ bits. In the discussion below we assume each element has at least one bit. This can easily be achieved by padding a bit onto every string, which does not affect the bounds. We will use a_i to refer to the value stored at $A[i]$.

3.1 Overview

Our VLA data structure is based on maintaining a dynamic partition of the contents of A into *blocks* of contiguous elements so that the average number of bits per block is $\theta(w)$. The blocks are stored in a dictionary using the location of the first element in the block (the leader) as the key, and the block as the data. A bit array is used to find the index of an element's leader in constant time, and an auxiliary word is kept with each block that is used to locate an element within a block in constant time. Using these structures we can support `update` and `lookup` in constant time. We now present the structure in more detail.

3.2 Blocks

The elements of the array A are partitioned into a set of blocks $\{B_{i_1}, B_{i_2}, \ldots B_{i_l}\}$ where each *block* B_i is an encoding of a consecutive set of entries from A: a_i, a_{i+1}, …, a_{i+r}. The block stores the concatenation of the bit strings $b_i = a_i a_{i+1} \ldots a_{i+r}$, together with information from which the start location of each string can be found. It suffices to store a second bit string b'_i such that b'_i contains a 1 at position j if and only if some bit string a_k ends at position j in b_i. A block B_i consists of the pair (b_i, b'_i), and we define the size of a block by $|b_i| = \sum_{j=0}^{r} |a_{i+j}|$. The following invariants are maintained:

(1) the size of each block is at most w

(2) for any two adjacent blocks (*i.e.*, one containing a_{i-1} and the next containing a_i), the sum of their sizes is greater than w.

We refer to $L = \{i_1, i_2, \ldots i_l\}$ (the start position of each block) as the set of *leaders* and say that i is the leader of j if a_i is the first element in $a'_j s$ block. The blocks are stored in a dictionary H that maps each leader i to its block (b_i, b'_i). A bit array $I[1 \ldots n]$ is also maintained where $A[i] = 1$ if and only if $i \in L$.

3.3 Lookups and updates

We begin by observing that from any position $k, 1 \leq k \leq n$, the distance to the nearest leader in either direction is at most w. This is because a block can only hold w bits and each string has at least one bit. To find the nearest leader to the

left (with lesser index), we let $s = I[k-w]\dots I[k-1]$ and compute $\texttt{bitSelect}(s, \texttt{bitRank}(s, w-1))$. To find the nearest leader to the right (with larger index), we let $s = I[k+1]\dots I[k+w]$ and compute $\texttt{bitSelect}(s, 1)$. These operations take constant time.

To lookup a location $A[k]$, one locates its leader i using I (nearest leader to the left) and finds the block (b_i, b'_i) using H. Once the block is located, a_k can be extracted from b_i since the start and end locations of a_k can be found from b'_i using $\texttt{bitSelect}(b'_i, k-i)$ and $\texttt{bitSelect}(b'_i, k-i+1)$, respectively.

To update $A[k]$, its block B_i needs to be rewritten. Since the new element might be of a different size than the previous element, this could involve some shifting of bits in B_i to reduce or open space for the new element. This shifting might, in turn, make the block either too small, violating the second invariant on block sizes, or too large, violating the first.

If a block becomes too small, it is merged with the adjacent block with which it violates the invariant. This must be possible since the sum of their lengths is less than w. If it violates the invariant with both adjacent blocks it is merged with one, and if it still violates the invariant with the other, it is also merged with the other. Merging blocks B_i and $B_j, i < j$ involves concatenating the contents from the blocks with shifts and logical operations, re-inserting the result into H with key i, deleting the leader j from H, and setting $I[j] := 0$. This will restore the invariant since the block with a_k no longer violates the invariant with the original adjacent blocks, and any block on the far side of the merged adjacent block will now be adjacent to a larger block.

If B_i becomes too large, it is split into at most three blocks. The new blocks will be either a block beginning with a_k, a block beginning with a_{k+1}, or (if the new $|a_k|$ is large) both. To maintain the size invariant, it may then be necessary to merge B_i with the block on its left, or to merge the rightmost new block with the block on its right. Splitting blocks involves extracting the appropriate bits, inserting a new leader in H, and setting the appropriate bit in I. Splitting and merging cannot propagate since although a split can force up to two merges, a merge cannot force a split.

All of the operations on blocks and on I take $O(1)$ time. The time is therefore limited by the time for operations on the dictionary H.

3.4 A fast dictionary for blocks

Implementing the dictionary H with separate chaining and universal hashing [Carter and Wegman 1979] gives an implementation of the variable-bit-length-array data structure for which lookup takes expected $O(1)$ time and update takes $O(1)$ expected amortized time. Implementing H with Cuckoo hashing [Pagh and Rodler 2004] or the dynamic version of the FKS perfect hashing scheme [Dietzfelbinger et al. 1994] improves lookup to $O(1)$ worst case time. Here we present an implementation of H that takes advantage of the particular nature of the problem and supports insertion and deletion in $O(1)$ worst-case time.

We are looking to solve a dictionary problem where the keys come from $\{1, \dots, n\}$, the data associated with each key is of size $\Theta(w)$, and for every w consecutive integers in the domain $[1, \dots, n]$, at least one of them is in the dictionary. We call this the *bounded sparsity dictionary problem.*

THEOREM 3.1. *The bounded sparsity dictionary problem on a dictionary T can be solved with $O(1)$ time insertions, deletions, and lookups, using $O(|T|w)$ bits.*

PROOF. The idea for the dictionary is to partition the domain $\{1, \ldots, n\}$ into *groups* of size w of contiguous indices (*i.e.*, $\{1, \ldots, w\}, \{w+1, \ldots, 2w\}, \ldots$). Each group is responsible for the entries with keys in its range, and will therefore contain between 1 and w entries. For each group $g \in \{1, \ldots, \lceil n/w \rceil\}$ the data structure will include

(1) a count c of how many entries it contains,
(2) an array D containing the data for the entries stored one after the other, but not necessarily in the same order as their keys, and
(3) an array $E[1 \ldots w]$ storing for each key $[g * w + 1, \ldots, (g+1) * w]$ a pointer to its position in D. Entries with no data are marked with a null pointer.

All this information for each group is stored in a contiguous *region* of memory which is allocated from the global pool. An array $P[1 \ldots \lceil n/w \rceil]$ of pointers is maintained with pointers from each group index to its region of memory.

For a lookup with key $g * w + i$ in group g, we check $E[i]$ and if null, we return null, otherwise we return $D[E[i]]$. When an entry with key $g * w + i$ is inserted, we increment the counter c, store the data at location $D[c]$, and set the pointer $E[i] := c$. When an entry is deleted, if it is the topmost entry in D (*i.e.*, $E[i] = c$) we delete it, otherwise we swap it with the topmost entry, adjusting pointers in E, and delete it. In either case c is decremented and $E[i]$ is set to null. All operations take constant worst-case time assuming there is enough space in the array D.

The array D is allocated is sizes that are powers of two and resized when it fills or becomes too empty (*e.g.*, double the size when it fills, and halve the size when it goes below 1/4th full). The algorithm also updates the number of bits used to store each of the pointers in E when D is resized so the number of bits is just enough to point to all locations in D—this is necessary to achieve the required bounds. Since a constant fraction of the elements of D are occupied, the space used by D is $O(cw)$. For E, each pointer will require $O(\log c)$ bits, for a total of $\Theta(w \log c)$. This can be charged against the number of bits taken by the $\Theta(c)$ entries, which is $\Theta(cw)$. The counter only takes $O(\log c)$ bits. Hence the space required by a group is $O(cw)$.

Freeing of memory regions satisfies the bounded-release condition described in Section 2.4 since we can account the instructions used in the insertions or deletions executed on a group between resizing towards the free for the old region. Furthermore, the contents of a region can be copied to the new resized region incrementally—every insertion or deletion does a constant number of copy steps with a constant selected such that by the time another resizing is required, the data is fully copied. While incrementally copying, any insertion or deletion is applied to the new copy, and any lookup is done in the new copy if copied or updated already, and in the old if not.

All operations therefore take $O(1)$ worst case time and the total space for the dictionary is $O(|T|w)$ bits. □

THEOREM 3.2. *An array $A[1 \ldots n]$ storing elements from S_w with m total bits can be represented using $O(n + m)$ bits while supporting lookups and updates in*

$O(1)$ worst-case time.

PROOF. For an array A the variable-bit-length array structure described contains n bits in I plus $O(w)$ bits per block. There are $O((n+m)/w)$ blocks since the invariant guarantees that the average number of bits per block is at least $w/2$, not including the last block, and we padded each string with a bit, adding n bits. When the blocks are stored in a bounded sparsity dictionary they will therefore use $O(n+m)$ space. All operations as described take constant time and make a constant number of calls to the bounded sparsity dictionary. □

4. VARIABLE-BIT-LENGTH DICTIONARIES

The dynamic variable-bit-length dictionary (VLD) problem is to support lookup, insertion and deletion on a dictionary $T \subset S_w \times S_w$ of key-data pairs. We assume the keys in T are distinct, that lookup on a key returns the data associated with the key or some special value if the key is not in T, and that insertion using a key already in T overwrites the old data with the new data. We define the *total bits* of T as $m = \sum_{(s,t)\in T}(|s| + |t|)$.

We first discuss a straightforward implementation based on chained hashing that supports lookups in $O(1)$ expected time and insertions and deletions in $O(1)$ expected amortized time. We then present an implementation based on the dynamic version [Dietzfelbinger et al. 1994] of the FKS perfect hashing scheme [Fredman et al. 1984] that improves the lookup time to $O(1)$ worst-case time. Both structures use quotienting as described in Section 2.2 and make use of the VLA structure described in the previous section.

For all hash tables we use a number of buckets that is a power of 2—*i.e.*, 2^q for some integer q. As the number of entries in T grows or shrinks, we resize the structure using a standard doubling or halving scheme so that $2^q \in \Theta(|T|)$. For convenience we assume buckets are numbered from 0 to $2^q - 1$ instead of 1 to 2^q.

4.1 Hashing

For hashing it will be convenient to treat the bit strings s as integers. Accordingly, when necessary we interpret each bit string as the binary representation of an integer. To ensure that every string has a unique integer representation given that they have different lengths, we prepend a `1` to each string. The strings used for keys can thus have length up to $w+1$.

In the framework described in Section 2.2, our quotienting scheme uses a hash function $h(s) : S_{w+1} \to \{0, \ldots, 2^q - 1\}$, a quotient function $g(s) : S_{w+1} \to S_{w-q+1}$, and an inverse function $i(h, g) : \{0, \ldots, 2^q - 1\} \times S_{w-q+1} \to S_{w+1}$. For our space and time bounds we require that $|g(s)| = \max(|s| - q, 0)$, and that $h(s)$ comes from a family of $c-$universal hash functions.

A family $\mathbb{H}$ of hash functions $h : U \to R$ is $c-$*universal* if for any $x_1, x_2 \in U, x_1 \neq x_2$, and uniformly selected random $h \in \mathbb{H}$, $\Pr(h(x) = h(y)) \leq c/|R|$. A family $\mathbb{H}$ is $(c, 2)-$*universal* (or c pairwise universal) if for any $x_1, x_2 \in U, x_1 \neq x_2$, any $y_1, y_2 \in R$, and uniformly random $h \in \mathbb{H}$, $\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2) \leq c/|R|^2$.

To construct h we use any family of $(2,2)-$universal hash function $\mathbb{H}_0$ with domain $\{0, \ldots, 2^{w+1-q} - 1\}$ and range $\{0, \ldots, 2^q - 1\}$. For example, we can use:

$$h_0(x) = ((\alpha x + \beta) \bmod p) \bmod 2^q$$

where $p > 2^{w+1-q}$ is prime and α, β are randomly and independently chosen from $\{1, \ldots, p-1\}$ and $\{0, \ldots, p-1\}$ respectively [Carter and Wegman 1979].

Given $\mathbb{H}_0$, we construct a family of hash functions

$$\mathbb{H} = \{h(s) = (s \bmod 2^q) \oplus h_0(s \operatorname{div} 2^q) : h_0 \in \mathbb{H}_0\}$$

where $\oplus$ indicates logical exclusive or of the bits of two integers.

We choose our hash function $h(s)$ randomly from $\mathbb{H}$ and use the quotient function $g(s) = s \operatorname{div} 2^q$ (this is simply shifting s right by q bits). It is not hard to verify that the following works as an inverse function given the $h_0(s)$ that $h(s)$ is based on:

$$i(h, g) = g \cdot 2^q + h \oplus h_0(g)$$

We show that the family $\mathbb{H}$ is 2−universal as follows. Given $x_1, x_2 \in S_{w+1}, x_1 \neq x_2$, we have

$$\begin{aligned} \Pr(h(x_1) = h(x_2)) &= \Pr((x_1 \bmod 2^q) \oplus h_0(x_1 \operatorname{div} 2^q) = (x_2 \bmod 2^q) \oplus h_0(x_2 \operatorname{div} 2^q)) \\ &= \Pr(h_0(x_1 \operatorname{div} 2^q) \oplus h_0(x_2 \operatorname{div} 2^q) = (x_1 \bmod 2^q) \oplus (x_2 \bmod 2^q)) \end{aligned}$$

If $x_1 \operatorname{div} 2^q = x_2 \operatorname{div} 2^q$ the probability is zero since $h_0(x_1 \operatorname{div} 2^q) \oplus h_0(x_2 \operatorname{div} 2^q) = 0$, but $x_1 \bmod 2^q$ and $x_2 \bmod 2^q$ must be different given that $x_1 \neq x_2$. Otherwise the probability is $\leq 2/2^q$ by the $(2,2)$−universality of $\mathbb{H}_0$). Thus $\Pr(b_1 = b_2) \leq 2/2^q$.

Note also that selecting a function from $\mathbb{H}_0$ and hence $\mathbb{H}$ requires $O(w)$ random bits.

4.2 Dictionary with chained hashing

Our simpler data-structure for the VLD uses a variant of the standard hashing with separate chaining. The data structure consists of a hash table using a variable-bit-length array $A[0 \ldots 2^q]$ for the buckets. It uses a hash function $h(s) \in \mathbb{H}$, quotient function $g(s)$, and inverse function $i(h, g)$ as described above. To insert a key-value pair (s, t) into the structure, we store the pair $(g(s), t)$ in bucket $A[h(s)]$. To store the pair $(g(s), t)$ we prepend a gamma code to each of $g(s)$ and t indicating their length, and then concatenate these together into a single bit string. The gamma code increases the length by at most a constant factor.

It is also necessary to handle several strings hashing to the same bucket. If all the entries in the bucket along with a gamma coded count of the number of entries fit within w bits, we simply concatenate the bits using bit-shifting and store the concatenation in the appropriate array slot A. Otherwise we allocate a separate VLA for the bucket and store the entries one per location, possibly doubling the array when it overflows and halving it when it becomes too empty. The cost of halving or doubling can be amortized against the insertions or deletions between resizing. A pointer is maintained to the secondary VLA, and the cost of this pointer can be charged against the fact that the number of bits within the bucket is more than w.

To implement a lookup on a key s we look in bucket $h(s)$ and search for the quotient $g(s)$. If we find $g(s)$ we return the corresponding t—since the bucket and

quotient map to a unique key, matching on the quotient implies matching on the key. To search for the quotient we note that the elements in a bucket can be decoded one after the other, each taking constant time. The count on the number of elements indicates when to stop. Each bucket has expected $O(1)$ elements, since the hash functions used are 2-universal, so lookups for any element can be accomplished in expected $O(1)$ time. Deletion requires searching for the key as above, and then splicing out the quotient and value using bit shifts. Since we assume insertion removes any equal valued key, insertion also has to first search and then insert at the end if not found.

Inserting or deleting might require incrementing or decrementing q (doubling or halving the number of buckets), and rehashing the whole table. To rehash we need $i(h, g)$ to determine the key for each element in the hash table. The cost of the free and rehashing can be amortized against the previous 2^{q-2} operations. Therefore insertions and deletions can be accomplished in expected amortized $O(1)$ time.

4.3 Dictionary with perfect hashing

Our second data-structure for the VLD uses a variant of the dynamic version [Dietzfelbinger et al. 1994] of the FKS perfect hashing scheme [Fredman et al. 1984]. Recall that the FKS scheme uses two levels of hashing. The first level hashes all keys into about n buckets. Let l_i be the number of elements that hash to bucket i, then each bucket uses its own second level hash table of size about l_i^2. This size results in a constant probability that there are no collisions in the second level table. If there are collisions, another hash function is selected for the bucket and this repeats until one is found which has no collisions. Also if there is too much imbalance between buckets, a different first-level hash function is selected.

In the dynamic version of FKS, the buckets are resized and the hash functions selected as necessary to keep appropriate balance between buckets and avoid collisions at the second level. The scheme we use follows directly from Dietzfelbinger et al. [1994] dynamic FKS except in three ways:

(1) we store the quotient from the first-level hash function instead of the key in the buckets,

(2) if there are fewer than w bits that hash to a bucket, then we store the bits directly in the bucket and do not use a second-level hash table, and,

(3) all arrays for the first and second level hash tables use a VLA structure.

For the first level hashing we use the same functions $h(s), g(s)$ and $i(h, g)$ as above. We maintain a variable-bit-length array of 2^q buckets, and as before we store each pair $(g(s), t)$ in the bucket indicated by $h(s)$. The number of buckets 2^q is set to maintain the condition of the dynamic FKS scheme, and a new function $h(s)$ is selected from $\mathbb{H}$ when required by the scheme.

If multiple strings collide within a bucket, and their total length is w bits or less, then we store the concatenation of the strings in the bucket, as we did with chained hashing above. If the length is greater than w bits we allocate a separate VLA to store the elements using a second level of hashing. We maintain the size and hash function of the second-level hash table exactly as described by Dietzfelbinger et al. [1994]. We note that they suggest a lazy deletion scheme in which deleted locations

are just marked as deleted until the next resizing. This can work for us, but when marking a location as deleted, its contents has to be removed (overwritten with an empty string) to maintain the space bound.

In the primary array we store a w-bit pointer to the secondary array for that bucket. We charge the cost of this pointer, and the $O(w)$-bit overhead for the array and hash function, to the cost of the w bits that were stored in that bucket. The space bounds for our dictionary structure follow from the bounds proved in [Dietzfelbinger et al. 1994]: the structure allocates only $O(n)$ array slots, and using our VLA structure requires only $O(1)$ bits per unused slot. Thus the space requirement of our structure is dominated by the space required to store the quotients and data from the dictionary entries.

Access to entries stored in secondary arrays takes worst-case constant time. Access to entries stored in the primary array is more problematic, as the potentially w bits stored in a bucket might contain $O(w)$ entries, and to meet a worst-case bound it is necessary to find the correct entry (quotient) in constant time.

We can solve this problem using table lookup by scanning the string s in regions of size αw, $0 < \alpha \leq 1$. Each lookup would take two arguments: a region starting at the beginning of some entry (quotient and data) in s, and the quotient to be looked up. It would return whether the quotient appears in the region and where, and therefore potentially process multiple entries at once. If it does not appear in the region it would return the start of the last entry that starts in the region. The finger can then be moved to that point for the next lookup. If the quotient being looked up is longer than αw we can just try to match it directly. If it does not match, we can use a table to skip to the start of the last entry that starts in the region. The main table would have $2^{\alpha w}2^{\alpha w}$ entries mapping a region and a quotient to a pointer within a region, requiring $2^{2\alpha w}\log(\alpha w)$ bits. The process takes at most $O(1/\alpha)$ steps. By setting $\epsilon = 3\alpha$ this is within our bounds described in Section 2.5.

This scheme gives us the following theorem:

THEOREM 4.1. *A dictionary $T \subset S_w \times S_w$ with n elements and m total bits can be represented using $O(m + 3n - n\log_2 n)$ bits, while supporting lookups in $O(1)$ worst-case time, and insertions and deletions in $O(1)$ expected amortized time.*

PROOF. The quotient stored for each s uses $O(\max(|s| - q, 1))$ bits and the data t uses $O(|t|)$ bits (including the gamma code). We maintain $q \geq \log_2 n$ by resizing. The VLA structures increases the space by at most a constant factor plus a linear factor in the total number of entries. The total space used by our variable-bit-length dictionary structure is therefore $O(n + \sum_{(s,t)\in T}(\max(|s| - \log_2 n, 1) + |t|))$.

Because the keys need to be unique, we have the bound $3n > \sum_{(s,t)\in T}(\max(|s| - \log_2 n, 1) - (|s| - \log_2 n))$. Therefore the space bound is within $O(3n + \sum_{(s,t)\in T}(|s| + |t| - \log n))$ which is equivalent to $O(m + 3n - n\log_2 n)$.

The time bounds follow from the discussion above. □

We now show that the representation is compact. Recall that the information theoretic lower bound $I(n, ...)$ is a function representing the logarithm of the number of distinct values of the given size, and that a data structure is compact if it uses $O(n + I(n, ...))$ bits and supports its operations efficiently.

THEOREM 4.2. *The VLD data structure we described is compact with respect to*

the number of entries n and total bits m.

PROOF. We first consider the case when the data fields are empty. To derive the desired bounds it is sufficient to consider just the sets for which all keys are the same length within one, *i.e.*, between $\lfloor m/n \rfloor$ and $\lceil m/n \rceil$. Including other sets would just increase the lower bound. Since there are at least $2^{\lfloor m/n \rfloor}$ possible keys, the number of possible such sets is at least $\binom{2^{\lfloor m/n \rfloor}}{n}$. This gives:

$$\begin{aligned} I(n,m) &> \log_2 \binom{2^{\lfloor m/n \rfloor}}{n} \\ &> \log_2((2^{\lfloor m/n \rfloor}/n)^n) \\ &> n(\log_2(2^{m/n-1}) - \log_2 n) \\ &= m - n - n\log_2 n \end{aligned}$$

We also note that $I(n,m)$ must be positive, so $O(n + I(n,m))$ is the same as $O(\max(m-n\log_2 n, n))$. From Theorem 4.1 we have the space for the VLD bounded by $S(n,m) \in O(m+3n-n\log n)$ which is within the bound $O(\max(m-n\log_2 n, n))$. To include the data fields we note that each bit in a data field contributes at least one bit to $I(n,m)$ since it can have two values. Since it also contributes 1 bit to m and hence $O(1)$ bit to $S(n,m)$, we have our bounds. All the operations on sets are as efficient as non-compact dictionaries. □

We finish the section by noting that another possible solution to the VLD problem would be to use w separate dictionaries, one for each of the possible lengths of the key. In fact, $\log w$ dictionaries would suffice, one for each power of two up to w. Each length would use a succinct or compact dynamic dictionary for fixed length keys which supports satellite data (*e.g.*, [Raman and Rao 2003]). However, these structures would need to be extended to handle variable-length data fields. This should not be hard using the VLA structure described in Section 3. When doing a lookup or update, the data-structure would apply the operation to the appropriate dictionary. For small n one would need to be careful that the overhead for multiple dictionaries is not too large, *i.e.*, that it is $O(w)$ bits.

5. COMPACT GRAPH REPRESENTATION

We describe a data-structure for maintaining directed graphs. It is based on an integer labeling of the vertices and uses difference coding between neighboring vertices to reduce space. For an appropriate labeling, any graph with small edge separators—*e.g.*, bounded-degree planar graphs—can be represented using a linear number of bits.

We consider the following operations on a directed graph $G = (V, E)$:

ADJACENT(u, v): true iff $(u, v) \in E$

FIRSTEDGE(u): return the first neighbor of u in G

NEXTEDGE(u, v): for an edge $(u, v) \in E$, return the next neighbor of u

ADDEDGE(u, v): add the edge (u, v) to E as the first edge leaving v

DELETEEDGE(u, v): delete the edge (u, v) from E.

```
ADJACENT(u, v)
    return (LOOKUP((u, v)) ≠ null)

FIRSTEDGE(u)
    (v_p, v_n) ←LOOKUP((u, u))
    return v_n

NEXTEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, v))
    return v_n

ADDEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, u))
    (u, v_nn) ←LOOKUP((u, v_n))
    INSERT((u, u), (v_p, v))
    INSERT((u, v), (u, v_n))
    INSERT((u, v_n), (v, v_nn))

DELETEEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, v))
    (v_pp, v) ←LOOKUP((u, v_p))
    (v, v_nn) ←LOOKUP((u, v_n))
    INSERT((u, v_p), (v_pp, v_n))
    INSERT((u, v_n), (v_p, v_nn))
    DELETE((u, v))
```

Fig. 1. Pseudocode for the graph operations.

FIRSTEDGE and NEXTEDGE are used to list the neighbors of a vertex and we assume there is an arbitrary ordering on outgoing edges (neighbors) from each vertex. We begin by describing a general data structure for representing integer labeled n-vertex graphs. We then describe how this structure can be efficiently compressed by assigning labels appropriately.

We uses a representation similar to the standard adjacency-list representation but represent each edge (element in a "list") using an entry in a dictionary. Consider a vertex u and some ordering on its neighboring vertices $v_1, \ldots, v_{d(u)}$, where $d(u)$ is the degree of u. We define $v_0 = u$ and $v_{d(u)+1} = u$. We represent each edge (u, v_i) (for $1 \leq i \leq d$) using the dictionary entry $(u, v_i; v_{i-1}, v_{i+1})$—that is, (u, v_i) is the key, and (v_{i-1}, v_{i+1}) is the associated data. This effectively forms a doubly-linked "list" among the edges of a vertex. For each vertex we include an entry $(u, u; v_{d(u)}, v_1)$ which is used as the head and tail of this list.

Given this representation we can support all of the above operations using dictionary operations—the pseudocode is shown in Figure 1. The ADDEDGE inserts the edge at the front of the adjacency list of the source vertex, and DELETEEDGE splices the edge out of its list. For ADDEDGE we leave out the case where there is no existing edge on the vertex, which is straightforward to implement. By using an appropriate implementation of hashing [Dietzfelbinger et al. 1994; Pagh and Rodler 2004] the update operations, ADDEDGE and DELETEEDGE take $O(1)$ expected amortized time and the rest of the operations take $O(1)$ worst-case time. For n vertices and m edges, the structure uses $O((n + m)w)$ bits.

To compress this structure we make use of difference coding by storing each dictionary entry using differences with respect to u. That is to say, rather than storing an entry $(u, v_i; v_{i-1}, v_{i+1})$ in the dictionary, we instead store $(u, v_i - u; v_{i-1} - u, v_{i+1} - u)$. We then use a variable-bit-length dictionary to store the entries, where the binary code for u and gamma code for $(v_i - u)$ are concatenated to form the key, and the gamma codes for $(v_{i-1} - u)$ and $(v_{i+1} - u)$ are concatenated together

to form the data.[2] This gives the following result.

THEOREM 5.1. *Any graph with n vertices with integer labels bounded by $O(n)$ can be stored in $O(n + \sum_{(u,v)\in E} \log|u-v|)$ bits, while supporting queries in $O(1)$ time and updates in $O(1)$ expected amortized time.*

PROOF. The time bounds follow from the discussion above. For space we consider the total bits in the dictionary. For each entry $(u, v_i; v_{i-1}, v_{i+1})$, the binary code for u uses $\log n + O(1)$ bits since the label is bounded by $O(n)$. For the three differences that are encoded in each entry we note that every difference $u - v$ corresponds to an edge in the graph (or a 0 when encoding $u - u$ at the beginning or end of the lists). Furthermore every edge appears in at most three entries—the previous, the actual edge, and the next entry in the adjacency list. Since each edge (u, v) appears $O(1)$ times in the differences, and the gamma code for each difference takes $O(\log(|u-v|))$ bits, the total bits is bounded by $n(\log n + O(1)) + O(\sum_{(u,v)\in E} \log|u-v|)$ bits. When used with Theorem 4.1, this gives the desired space bounds. □

The graph structure we describe can be used to represent any graph but the space bound is best only for graphs which have some locality on their labels. We say that a labeling of an n-vertex graph is *k-compact* if $\sum_{e\in E} \log_2|u-v| < kn$. We say a graph is *$k$-compact* if it has a k-compact labeling. Blandford et al. showed that for any family of graphs satisfying an $O(n^{1-\epsilon})$-edge separator theorem, $\epsilon > 0$, all members are $O(1)$-compact [Blandford et al. 2003]. This includes bounded-degree planar graphs, which satisfy an $O(n^{\frac{1}{2}})$-edge separator theorem, and certain well-shaped meshes [Miller et al. 1997] of fixed dimension. The labeling can be found using separator trees. Additionally, many graphs in practice have been found to be k-compact for much smaller k than would be expected for random graphs [Blandford et al. 2004] (*e.g.*, web link graphs, VLSI circuits, and Internet connectivity graphs). The following follows directly from the theorem above.

COROLLARY 5.2. *All n-vertex graphs with a k-compact labeling can be stored in $O(kn)$ bits while allowing updates in $O(1)$ expected amortized time and queries in $O(1)$ worst-case time.*

6. ORDERED INTEGER SETS

The ordered integer set problem is to represent a dynamic set $S \subset \{0, \ldots, m-1\}$, while supporting operations that take advantage of the order. Here we consider finger searching: *fingerSearch* with a finger to a key $k_1 \in S$ and a search key $k_2 \in S$ finds the key $\min\{k_3 \in S | k_3 > k_2\}$, and returns k_3 and a finger to k_3. Finger searching will take $O(\log l)$ time, where $l = |\{k \in S | k_1 \le k \le k_2\}|$. Insertion and deletion, and generating a finger to the next key in S greater or equal to a given key k, will take time $O(\log|S|)$. We show that this also leads to a data structure which supports insertion, deletion and finding the next key greater than a given key (*findNext*) in time $O(\lg\lg m)$, but not finger searching. In all cases the data structures use $O(n\log(m/n))$ bits, which is asymptotically optimal.

[2]We assume the word length is sufficient to hold the concatenation—if not it is straightforward to simulate the longer words (at most a constant times the word length) with shorter words.

To represent the set we use a standard red-black tree on the elements, but use difference codes between keys to store "pointers". Other balanced trees could also be used. We will refer to nodes of the tree by the value of the element stored at the node. We assume $n \leq m/2$. (If $n > m/2$, then rather than storing S, our representation stores the complement of S.) For each element v we denote the parent, left-child, right child, and red-black flag as $p(v)$, $l(v)$, $r(v)$, and $q(v)$ respectively.

The tree is represented as a dictionary containing entries of the form $(v; l(v) - v, r(v) - v, q(v))$. (It is also possible to add parent pointers $p(v) - v$ without violating the space bound, but in this case they are unnecessary.) We store the integer at the root directly. It is straightforward to traverse the tree from top to bottom in the standard way. It is also straightforward to implement a rotation by inserting and deleting a constant number of dictionary elements. Assuming dictionary queries take $O(1)$ time, using a hand data structure [Blelloch et al. 2003], finger searching can be implemented in $O(\log l)$ time with an additional $O(\log^2 n)$ space. Insertion and deletion take $O(\log n)$ expected amortized time. Unlike balanced trees, this structure has the added advantage of supporting lookup (membership testing) in $O(1)$ time.

It remains to show the space bound for the structure.

LEMMA 6.1. *For a set of integers $S \subset \{0, \ldots, m-1\}$ of size n stored in a red-black tree T, $\sum_{v \in T}(\log |p(v) - v|) \in O(n \log(m/n))$.*

PROOF. Consider the elements of a set $S \subset \{0, \ldots, m-1\}$ organized in a set of levels $L(S) = \{L_1, \ldots, L_l\}$, $L_i \subset S$. If $|L_{i+1}| \leq \alpha |L_i|, 1 \leq i < l, \alpha < 1$, we say such an organization is a *proper level covering* of the set.

We first consider the sum of the log-differences of cross pointers within each level, and then count the pointers in the red-black trees against these pointers. For any set $S \subset \{0, \ldots, m-1\}$ we define $\mathit{next}(e, S) = \min\{e' \in S \cup \{m\} | e' > e\}$, and $M(S) = \sum_{j \in S} \log(\mathit{next}(j, S) - j)$. Since logarithms are concave the sum is maximized when the elements are evenly spaced and hence $M(S) \leq |S| \log(m/|S|)$. For any proper level covering L of a set S this gives:

$$\begin{aligned}
\sum_{L_i \in L(S)} M(L_i) &\leq \sum_{L_i \in L} |L_i| \log(m/|L_i|) \\
&\leq \sum_{i=0}^{i<l} \alpha^i |S| \log(m/(\alpha^i |S|))) \\
&\leq |S| \sum_{i=0}^{i<l} \alpha^i (i \log(1/\alpha) + \log(m/|S|)) \\
&\in O(|S| \log(m/|S|))
\end{aligned}$$

This represents the total log-difference when summed across all "next" pointers. The same analysis bounds similarly defined "previous" pointers. Together we call these *cross pointers*.

We now account for each pointer in the red-black tree against one of the cross pointers. First partition the red-black tree into levels based on one more than the number of black nodes in the path from any leaf to the node, not including the

node—a red-black tree maintains the invariant that this is the same for all paths from a leaf. This gives a proper level covering with $\alpha = 1/2$. Now for each node i, the difference of its value to that of each of its two children is at most the difference of its value to that of the previous and next elements in its level. Therefore we can account for the cost of the left child against the previous pointer and the right child against next pointer. The sum of the log-differences of the child pointers is therefore at most the sum of the log-differences of the next and previous cross pointers. This gives the desired bound. □

THEOREM 6.2. *A set of integers $S \subset \{0, \ldots, m-1\}$ of size n represented as a dictionary red-black tree and using a compressed dictionary uses $O(n \log((n+m)/n))$ bits and supports finger-search queries in $O(\log l)$ time, and insertion and deletion in $O(\log n)$ expected amortized time.*

PROOF. Recall that the space needed for a compressed dictionary is bounded by $O(s + 3n - n\log_2 n)$, where s is the total bits in the dictionary. The bits stored in the dictionary consist of n keys of $log_2 m$ bits each, and a total of $O(n \log(m/n))$ bits for the pointers (by Lemma 6.1). This gives $s = n \log_2 m + O(n \log(m/n))$. The total space is therefore bounded by $O(n \log_2 m + O(n \log(m/n)) + 3n - n \log_2 n)$, which simplifies to $O(n \log((n+m)/n))$ bits. □

COROLLARY 6.3. *A set of integers $S \subset \{0, \ldots, m-1\}$ of size n, can be represented using $O(n \log(m/n))$ bits while supporting findNext in $O(\log \log m)$ time, and insert and delete in $O(\log \log m)$ expected amortized time.*

PROOF. The van Emde Boas tree structure [van Emde Boas et al. 1976] supports findNext, insert and delete in $O(\log \log m)$ time. By using dynamic perfect hashing [Dietzfelbinger et al. 1994] to store the elements of the tree, the structure can be implemented with $O(n \log m)$ bits, but the time for updates becomes $O(\log \log m)$ expected amortized time [Mehlhorn and Naher 1990]. To reduce the space, the set can be partitioned into groups of $\Theta(\log n)$ contiguous elements each (except perhaps the first and last groups which can be smaller). The least key in each group is called the *leader*. Each group is stored in its own dictionary red-black tree, and therefore supports find, insert, and delete in $O(\log \log n)$ time (expected amortized for the updates). The leader of each group is stored in the van Emde Boas structure. When a group becomes too large during an insertion it is split, and when it becomes too small during a deletion it is merged with a neighbor, and possibly re-split. The costs for splits and merges can be amortized against the number of insertions or deletions between splits or merges. The total time is therefore $O(\log \log m)$ for findNext, and $O(\log \log m)$ expected amortized for insertion and deletion.

To analyze space we define the range of a group as the difference between the smallest and largest value. For l groups of size $n_i, 1 < i \leq l$ and range m_i, the total space required for the dictionary red-black trees is $O(\sum_{i=1}^{l}(n_i \log(m_i/n_i))$. This is because every key is stored relative to keys above it in the tree, so the effective range of the integers stored for each group is m_i. We need $\lg m$ bits to point to the root. The sum is maximized when all the m_i are approximately equal, which gives a total space of $O(n \log(m/n))$ bits. Along with the $O((n/\log n) \log m)$ bits required to store the leaders in the van Emde Boas tree, we have the desired bounds. □

7. CARDINAL TREES

A cardinal tree is a rooted tree in which every node has c slots for children any of which can be filled[3]. We generalize the standard definition of cardinal trees to allow each node v to have a different c, denoted as $c(v)$. For a node v we want to support returning the parent $p(v)$ and the i^{th} child $v[i]$ (for $1 \leq i \leq c(v)$), if any. We also want to support deleting or inserting a leaf node. As with graphs, we consider these partially dynamic operations since the updates might require relabeling of the nodes to maintain the space bounds.

The data structure we describe is based on labeling the vertices of the tree with integers and using difference codes to represent the tree "pointers" between nodes.

LEMMA 7.1. *Integer labeled cardinal trees with vertices V and labels in the range $[1, \ldots, k|V|]$ can be stored in $O(\sum_{v \in V}(\log c(p(v)) + \log |p(v) - v| + \log k))$ bits while supporting parent and child queries in $O(1)$ time and insertion and deletion of leaves in $O(1)$ expected amortized time.*

PROOF. To support parent queries we keep a variable-bit-length dictionary storing entries of the form $(v; p(v) - v)$, for each node v. The total number of bits stored in the keys and data of this dictionary is $\sum_{v \in V}(\log_2(k|V|) + \log_2 |p(v) - v|)$, which using Theorem 4.1 gives $O(\log k + \log |p(v) - v|)$ bits.

To support child queries we keep a dictionary storing entries as follows: for each node v, if v is the ith child of its parent, we store an entry $(p(v), i; v - p(v))$. The keys are kept by appending the $\log(k|V|)$-bits for $p(v)$ with the $\log(c(p(v))$ bits for i. The total number of bits stored in the keys and data of this dictionary is therefore $\sum_{v \in V}(\log_2(k|V|) + \log_2(c(p(v)) + \log_2 |p(v) - v|)$, which based on Theorem 4.1 gives $O(\log k + +log(c(p(v)) + \log |p(v) - v|)$ bits.

The total number of bits across the two dictionaries is therefore as stated. □

Any tree T can be separated into a set of trees of size at most $n/2$ by removing a single node. Recursively applying such a separator on the cardinal tree defines a separator tree T_s over the nodes. An integer labeling can then be given to the nodes of T based on the preorder traversal of T_s. We call this preorder-traversal labeling a *tree-separator labeling*.

For each node $v \in T_s$, we denote the degree of v by $d(v)$. We let $T_s(v)$ denote the subtree of T_s that is rooted at v. Thus $|T_s(v)|$ is the size of the piece of T for which v was chosen as a separator.

LEMMA 7.2. *For all tree-separator labellings of trees $T = (V, E)$ of size n,*

$$\sum_{(u,v) \in E} (\log |u - v|) < O(n) + 2 \sum_{(u,v) \in E} \log(\max(d(u), d(v)))$$

PROOF. Consider the separator tree $T_s = (V, E_s)$ on which the labeling is based, and the following one-to-one correspondence between the edges E and edges E_s. Consider an edge $(v, v') \in E_s$ between a node v and a child v'. This corresponds to an edge $(v, v'') \in T$, such that $v'' \in T_s(v')$. We need to account for the log-difference

[3]We use the definition from [Benoit et al. 2005].

$\log|v-v''|$. We have $|v-v''| < |T_s(v)|$ since all labels in any subtree are given by the preorder traversal. We partition the edges into two classes and calculate the cost for edges in each class.

First, if $d(v) > \sqrt{|T_s(v)|}$ we have for each edge (v, v''), $\log|v-v''| < \log|T_s(v)| < 2\log d(v) < 2\log\max(d(v), d(v''))$.

Second, if $d(v) \leq \sqrt{|T_s(v)|}$ we charge each edge (v, v'') to the node v. The most that can be charged to a node is $\sqrt{|T_s(v)|}\log|T_s(v)|$ (one pointer to each child). Note that for any tree in which for every node v: (A) $|T_s(v)| < |T_s(p(v))|/2$, and (B) $\text{cost}(v) \in O(|T_s(v)|^c)$ for some $c < 1$, we have $\sum_{v\in V}\text{cost}(v) \in O(n)$. Therefore the total charge is $O(n)$.

Summing the two classes of edges gives

$$O(n) + 2\sum_{(u,v)\in E}\log(\max(d(u), d(v)))$$

□

THEOREM 7.3. *Cardinal trees with a tree-separator labeling can be stored in* $O(\sum_{v\in V}\log(1+c(p(v))))$ *bits.*

PROOF. We are interested in the edge cost $E_c(T) = \sum_{v\in V}(\log|v-p(v)|)$. Substituting $p(v)$ for u in Lemma 7.2 gives:

$$\begin{aligned}
E_c(T) &< O(n) + 2\sum_{v\in V}\log(\max(d(v), d(p(v)))) \\
&< O(n) + 2\sum_{v\in V}(d(v) + \log d(p(v))) \\
&= O(n) + 4n + 2\sum_{v\in V}\log d(p(v)) \\
&< O(n) + 2\sum_{v\in V}\log(1 + c(p(v)))
\end{aligned}$$

With Lemma 7.1 this gives the required bounds. □

8. SIMPLICIAL MESHES

Using our variable-bit-length dictionary structure we can implement space-efficient representations of d-dimensional simplicial meshes (triangulated meshes). We describe a representation that supports a simplicial mesh in which every $(d-1)$-dimensional face belongs to one or two d-dimensional simplices. We will describe the structure for 3 dimensions but note that this can be generalized to d dimensions. Simplices are defined as sets of vertices, such that a triangle (2-dimensional simplex) is a set of three vertices and a tetrahedron is a set of four vertices. The mesh structure supports the following operations on a mesh M:

FINDSIMPLEX($\{a,b,c\}$): returns vertices d such that $\{a,b,c,d\} \in M$

INSERTSIMPLEX($\{a,b,c,d\}$): adds $\{a,b,c,d\}$ to M

DELETESIMPLEX($\{a,b,c,d\}$): deletes $\{a,b,c,d\}$ from M

```
findSimplex(S)
  (a, b, c) ← order(S)
  return lookup((a, b, c))

insertSimplex(S)
  for each ordered face (a, b, c) in S
    (e, 0) ← lookup((a, b, c))
    insert((a, b, c), (d, e))

deleteSimplex(S)
  for each ordered face (a, b, c) of S
    (d, e) ← lookup((a, b, c))
    if e ∈ S then swap(d, e)
    if e = 0 then delete((a, b, c))
    else insert((a, b, c), (e, 0))
```

Fig. 2. Pseudocode to support simplicial mesh operations.

We represent a simplicial mesh as a dictionary of simplices. Each face $\{a, b, c\}$ in the mesh may belong to two tetrahedra, $\{a, b, c, d\}$ and $\{a, b, c, e\}$. For each face in the mesh we store the entry $(a, b, c; d, e)$. We assume a canonical ordering on the vertex labels, for example $a < b < c$, and only store the face in that order. We will refer to this as an *ordered face*. If a face belongs to only one tetrahedra (on a boundary) then we store the special character 0 in the second slot. The operations can then be implemented as shown in Figure 2.

The representation can be compressed by labeling the vertices with integers and encoding b, c, d, and e relative to a. That is, the representation stores tuples of the form $(a, b-a, c-a; d-a, e-a)$ in the variable-bit-length dictionary.

Note that there is one tuple stored per face. To analyze the space usage of this structure, we charge the cost of storing $b-a$ and $c-a$ to the face (a, b, c); we charge the cost of $d-a$ to the face (a, b, d) and of $e-a$ to the face (a, b, e). Assuming the integer labels on the vertices V are $O(|V|)$, the quotienting of the dictionary absorbs the $(\log |V| + O(1))$-bit cost of representing a. Each face is charged $O(1)$ times, and each time the charge is $O(\max(\log |a-b|, \log |a-c|, \log |b-c|)) = O(\log |a-b| + \log |a-c| + \log |b-c|)$. This gives a bound of $O(\sum_{(a,b,c)\in F}(\log |a-b| + \log |a-c| + \log |b-c|))$, where F is the 2-skeleton (that is, the set of faces) of the mesh.

This structure generalizes to d dimensions. A similar argument to the above shows that the space usage in that case is $O(\sum_{f\in F}(d \max_{a,b\in E(f)} \log |a-b|))$, where F is the set of faces ($(d-1)$ simplices) in the mesh, and $E(f)$ is the set of edges in the face. For the special case of three dimensions we prove the following bound based on just the edges.

Theorem 8.1. *A 3-dimensional simplicial mesh with n vertices with integer labels bounded by $O(n)$ and edges E can be implemented with $O(\sum_{(a,b)\in E} \log |a-b|)$ bits while supporting lookups in $O(1)$ time, and updates in $O(1)$ expected amortized time.*

PROOF. The time bounds follow directly from the times for the compressed dictionary.

For space we begin with the space bound $O(\sum_{(a,b,c)\in F}(\log|a-b|+\log|a-c|+\log|b-c|))$ bits. To show the stronger bound, we wish to charge the cost of each face $(a,b,c)\in F$ to one of its adjoining edges. An edge (a,b) can be assigned a charge of $\log|a-b|$. We define the *heaviest edges* of (a,b,c) to be the two edges with the greatest difference between their vertices. For example, if $a<b<c$ and $c-b<b-a$, then (a,b) and (a,c) are the heaviest edges. Note that $\log|c-a|>\log|c-b|$ and $\log|c-a|\leq 1+\log|b-a|$, so we can charge the $O(\log|b-a|+\log|c-a|+\log|c-b|)$ cost of the face to either of its heaviest edges. However, we must ensure that no edge is charged more than $O(1)$ times.

To this end we describe an *edge-to-face mapping* so that every face maps to two of its edges, and every edge is mapped to a constant number of faces. One of the two edges a face maps to must be heavy, so therefore we can charge that face to the heavy edge, and every edge will be charged at most a constant number of times. This gives the desired space bound. Note that the mapping is purely for analysis and is not needed by the data structure.

For any vertex $v_i\in V$, consider all the tetrahedra that include the vertex. The surface created from the opposite faces of these tetrahedra from v_i form a two-dimensional surface, and is called the link $L(v_i)$ of v_i. Euler's rule applies to this surface, and hence we know that $|E(L(v_i))|<3|V(L(v_i))|$.

We will now direct all of the edges in $E(L(v_i))$ in such a way as to ensure that no vertex of $L(v_i)$ has in-degree greater than 5. This can be done iteratively: At each step, find a vertex $v\in V(L(v_i))$ of degree 5 or less. (Euler's rule guarantees that this is possible.) Direct all edges containing v into v, and then delete v and all its edges from $L(v_i)$. At termination, all edges have been directed, and no vertex has received in-degree greater than 5.

Now, consider the edges from v_i to the vertices in $V(L(v_i))$, and the faces formed by the vertex v_i and the edges in $E(L(v_i))$. For a face (v_i,v_j,v_k), w.o.l.g. assume the edge (v_j,v_k) is directed from v_j to v_k by the procedure above, add the edge (v_i,v_k) and face (v_i,v_j,v_k) to the edge-to-face mapping. Note that (v_i,v_k) will be assigned to at most 5 faces from v_i and at most another 5 when applying the same procedure from the other end. Therefore it is mapped to $O(1)$ faces. This is true for all edges.

Also note that when applying the procedure to v_j, at least one of the two edges (v_j,v_i) or (v_j,v_k) will be added to the edge-to-face mapping. Therefore the triangle (v_i,v_j,v_k) will be mapped to at least two of its edges. This will be true for all triangles. This gives our desired mapping and shows that every face can be charged to one of its heavy edges and that every edge gets charged $O(1)$ times. □

If the 1-skeleton of the mesh (that is, the graph induced by the edges) has a k-compact labeling, then the representation of the mesh will use $O(n)$ bits. We note that well-shaped meshes with bounded degree have small separators [Miller et al. 1997] and are therefore k-compact for fixed dimension.

9. CONCLUSIONS

We have described data structures for maintaining a dynamic dictionary where both the keys and associated data are bit-strings of varying length. The VLD structure is compact in that it's size is $O(n + I(n, m))$ where $I(n, m)$ is the information-theoretic lower-bound for table with n elements and m total bits across the keys and data. The time for operations match the best known results for dictionaries in general. We also described a compact array structure that supports variable-bit-length strings. The VLA structure is an important part of the implementation of the dictionary.

We described several data structures for various applications that made use of the VLD. All these data structures are based on standard pointer-based techniques, but pointers are stored as the difference between integer labels or integer values between elements. This approach seems to be a reasonably general technique that might be applied to many problems where labels can be assigned with some locality.

We leave some open questions. In regards to space, none of the structures we develop are succinct—*i.e.*, use $I(n, m) + o(I(n, m))$ bits. A succinct static VLA follows directly from succinct structures for the select operation [Munro 1996]. It seems that developing a succinct static VLD should also be possible. Developing a succinct dynamic VLA or VLD seem more difficult.

In regards to the applications, we described how to support dynamic operations on graphs, cardinal trees and simplicial meshes, but the effectiveness of these operations depends on maintaining a locality on the labels. In particular it is not clear how to maintain the space bounds under arbitrary sequences of updates. It would be interesting to study how to dynamically maintain labels so that space bounds can be maintained in the presence of arbitrary or perhaps certain classes of update operations.

REFERENCES

Baker, H. G. 1978. List processing in real-time on a serial computer. *Communications of the ACM 21,* 4, 280–94.

Benoit, D., Demaine, E. D., Raman, J. I. M. R., Raman, V., and Rao, S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4 (Dec.), 275–292.

Blandford, D. and Blelloch, G. 2004. Compact representations of ordered sets. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 11–19.

Blandford, D., Blelloch, G., and Kash, I. 2003. Compact representations of separable graphs. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 342–351.

Blandford, D., Blelloch, G., and Kash, I. 2004. An experimental analysis of a compact graph representation. In *Proceedings of the 6th Workshop on Workshop on Algorithm Engineering and Experiments (ALENEX)*.

Blandford, D. K., Blelloch, G. E., Cardoze, D. E., and Kadow, C. 2005. Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications 15,* 1, 3–24.

Blelloch, G. E., Maggs, B., and Woo, M. 2003. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 374–383.

Brodnik, A. and Munro, J. I. 1999. Membership in constant time and almost-minimum space. *Siam Journal on Computing 28,* 5, 1627–1640.

Carter, L. and Wegman, M. 1979. Universal classes of hash functions. *Journal of Computer and System Sciences 18,* 2, 143–154.

Chuang, R. C.-N., Garg, A., He, X., Kao, M.-Y., and Lu, H.-I. 1998. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *Lecture Notes in Computer Science 1443*, 118–129.

Cleary, J. G. 1984. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers 9*, 828–834.

Cohen, S. and Matias, Y. 2003. Spectral bloom filters. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 241–252.

Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., auf der Heide, F. M., Rohnert, H., and Tarjan, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing 23,* 4, 738–761.

Elias, P. 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21,* 2 (March), 194–203.

Fotakis, D., Pagh, R., Sanders, P., and Spirakis, P. G. 2005. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems 38,* 2, 229–248.

Fredman, M. L., Komlos, J., and Szemerdi, E. 1984. Storing a sparse table with O(1) worst case access time. *Journal of the ACM 31,* 3, 538–544.

Grossi, R. and Vitter, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

He, X., Kao, M.-Y., and Lu, H.-I. 2000. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM Journal on Computing 30,* 3, 838–846.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Symposium on Foundations of Computer Science (FOCS)*. 549–554.

Keeler, K. and Westbrook, J. 1995. Short encodings of planar graphs and maps. *Discrete Applied Mathematics 58*, 239–252.

Knuth, D. E. 1973. *The Art of Computer Programming/Sorting and Searching, Volumes 3.* Addison Wesley.

Mehlhorn, K. and Naher, S. 1990. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters 35,* 4, 183–189.

Miller, G. L., Teng, S.-H., Thurston, W. P., and Vavasis, S. A. 1997. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM 44*, 1–29.

Munro, J. I. 1996. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 1180 of Lecture Notes in Computer Science. Springer-Verlag, 37–42.

Munro, J. I. and Raman, V. 2001. Succinct representation of balanced parentheses, static trees and planar graphs. *SIAM Journal on Computing 31,* 2, 762–776.

Pagh, R. 2001. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing 31,* 2, 353–363.

Pagh, R. and Rodler, F. F. 2004. Cuckoo hashing. *Journal of Algorithms 51,* 2, 122–144.

Raman, R., Raman, V., and Rao, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

Raman, R. and Rao, S. S. 2003. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata; Languages and Computation (ICALP)*. 357–36.

Turán, G. 1984. Succinct representations of graphs. *Discrete Applied Mathematics 8*, 289–294.

van Emde Boas, P., Kaas, R., and Zijlstra, E. 1976. Design and implementation of an efficient priority queue. *Math. Systems Theory 10,* 2, 99–127.