Monotone Minimal Perfect Hashing: Searching a Sorted Table with $O(1)$ Accesses

Djamal Belazzougui[*] Paolo Boldi[†] Rasmus Pagh[‡] Sebastiano Vigna[†]

Abstract

A minimal perfect hash function maps a set S of n keys into the set $\{0, 1, \ldots, n-1\}$ bijectively. Classical results state that minimal perfect hashing is possible in constant time using a structure occupying space close to the lower bound of $\log e$ bits per element. Here we consider the problem of *monotone* minimal perfect hashing, in which the bijection is required to preserve the lexicographical ordering of the keys. A monotone minimal perfect hash function can be seen as a very weak form of *index* that provides *ranking* just on the set S (and answers randomly outside of S). Our goal is to minimise the description size of the hash function: we show that, for a set S of n elements out of a universe of 2^w elements, $O(n \log\log w)$ bits are sufficient to hash monotonically with evaluation time $O(\log w)$. Alternatively, we can get space $O(n \log w)$ bits with $O(1)$ query time. Both of these data structures improve a straightforward construction with $O(n \log w)$ space and $O(\log w)$ query time. As a consequence, it is possible to search a sorted table with $O(1)$ accesses to the table (using additional $O(n \log\log w)$ bits). Our results are based on a structure (of independent interest) that represents a trie in a very compact way, but admits errors. As a further application of the same structure, we show how to compute the predecessor (in the sorted order of S) of an arbitrary element, using $O(1)$ accesses in expectation and an index of $O(n \log w)$ bits, improving the trivial result of $O(nw)$ bits. This implies an efficient index for searching a blocked memory.

1 Introduction

This paper addresses a series of problems that lie at the confluence of two streams of research: the study of *minimal perfect hash functions*, and the analysis of *indexing structures*. A minimal perfect hash functions maps bijectively a set S of n keys into the set $\{0, 1, \ldots, n-1\}$. The construction of such functions has been widely studied in the last years, leading to fundamental theoretical results such as [12, 13, 15].

From an application-oriented viewpoint, *order-preserving* minimal perfect hash functions have been used to retrieve the position of a key in a given list of keys [11, 20]. We start from the observation that all existing techniques for this task assume that keys can be provided in any order, incurring an unavoidable $\Omega(n \log n)$ lower bound on the number of bits required to store the function. However, very frequently the keys to be hashed are sorted in their intrinsic (i.e., lexicographical) order. This is typically the case of dictionaries of search engines, list of URLs of web graphs, etc. We call the problem of mapping each key of a lexicographically sorted set to its ordinal position *monotone minimal perfect hashing*. This problem has received, to the best of our knowledge, no attention in the literature. However, as we will shortly explain, it is tightly connected with other classical problems. It is, in a way, a very weak form of *ranking*: for instance, *partial ranking* on a set S is given by a function that returns the lexicographical position of an element x of S, but returns -1 if x is not in S. Instead, a monotone minimal perfect hash function is allowed to return any result on elements not in S.

In a classical paper, Yao [27] showed that if one uses no extra space in addition to a table of $n > 2$ keys from an ordered universe u, and if u is sufficiently large, any organization of the table requires $\log(n+1)$ worst case search time. In particular, sorting the table yields the best possible search time. The lower bound holds even if we are interested only in membership queries ("is x in the table or not?"), and it extends to more general data structures allowing pointers and repeated keys in $n^{O(1)}$ space. A number of researchers have investigated the amount of extra space needed to break Yao's lower bound, using hashing techniques to provide membership queries with $O(1)$ table accesses [9, 10, 25]. However, these schemes do not support range queries ("Which are the keys in the range $[\ell\,..\,r]$?") beyond the trivial reduction to membership queries that requires linear time in the size of the range.

Here we consider the scenario where the table is

[*]Institut National d'Informatique, Oued Smar, Algiers, Algeria
[†]Università degli Studi di Milano, Italy
[‡]IT University of Copenhagen, Denmark

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

sorted, and ask the question: What is the space usage of index structures that make it possible to search the table using $O(1)$ accesses in expectation? We consider two kinds of search for a key x:

1. Membership searches where we must find the position of x in the table, or report that x is not in the table, and
2. Predecessor searches where we must return the position of the largest key not greater than x.

The second type immediately implies efficient range queries: Find the predecessor of ℓ and scan the table until a key greater than r is found. This kind of scanning is very efficient on disks, as well as modern blocked memory architectures. Indeed, several recent papers on range queries propose to keep keys in a sorted table (with gaps), rather than in a normal search tree, exactly for this reason [24, 2, 1]. (We note that these papers are concerned with the dynamic case where the key set is changing, while we consider only the static case.)

The first type of search allows point queries, but also range queries of a special kind: If we know some element in the range, its position can be found in $O(1)$ accesses, after which reporting all elements in the range is trivial. This may be relevant for example in database applications, where referential integrity constraints ensure that the elements in one relation also exist in another relation.

Our results. Suppose that our universe has size 2^w, and $n \geq \log w$. Without loss of generality we assume that w is a power of two (if not, round it up). Our model of computation is a unit-cost word RAM with word size w. We describe a monotone minimal perfect hash function of size $O(n \log\log w)$ bits and query time $O(\log w)$, and one of size $O(n \log w)$ bits and constant query time. In both cases, this implies that we can find an element in a sorted table using just one access to the table. Both constructions are based on novel ways of separating a set of keys into $O(n/k)$ ordered groups of size $O(k)$. This improves on the space and time, respectively, of the classic solution that stores every k-th key explicitly in a predecessor data structure (e.g. [26]). It is known that $\Omega(n)$ bits is a lower bound, even when the set is not required to be stored in sorted order [21], so these results are (at least) close to best possible.

The main tool in the more space efficient solution for membership search is an approximate trie representation that allows us to store a set S of n keys in space $O(n \log w)$ so that for every element y its rank relative to S can be approximately determined in the following sense: The data structure returns a set of two integers in $\{0, 1, \ldots, n\}$ such that with probability $1 - 1/w$ one of them is the position of the predecessor of y. The data structure is an approximate version of a new van Emde Boas tree-like data structure that we call *z-fast trie* (named after its relationship to y-fast tries [26]).

We believe that our approximate trie result is of independent interest, and could potentially find applications in settings where the two possible predecessors could be searched in parallel (e.g. in hardware solutions for routing, and in B-trees on parallel disk systems.) We also show a lower bound implying that the approximate trie representation is close to optimal in the following sense: Every data structure that determines the correct rank of each element in the universe with probability more than $1/2$ must use space close to the space required for storing S itself. Our data structure allows the rank to be determined with probability slightly below $1/2$.

An implementation of the data structures presented in this paper is distributed under the Gnu LGPL as part of the Sux4J project (`http://sux4j.dsi.unimi.it/`). We study carefully the implementation problems and the practical behaviour of our algorithms in a forthcoming paper.

Related work. Mehlhorn [21] showed that minimal perfect hashing requires space $\Theta(n + \log w)$ bits. The lower bound holds even in the more general case where the range of the function is of size $O(n)$ rather than exactly n, and h is required to be injective. A succinct data structure representing a minimal perfect hash function, with $O(1)$ evaluation time, was constructed by Hagerup and Tholey [15].

Schmidt and Siegel [25] considered a generalization of perfect hashing where the hash function h returns a *set* of at most k values from $\{0, \ldots, n-1\}$. The requirement is that there should exist a matching between S and $\{0, \ldots, n-1\}$ such that every key $x \in S$ is matched to an element of $h(x)$. For constant k this still requires $\Omega(n)$ bits of space. Specifically, upper and lower bounds of $O(ne^{-k} + \log\log m)$ and $\Omega((n/k^2)e^{-k} + \log\log m)$ bits were shown.

Mairson [18, 19] considered a related generalization of minimal perfect hashing where the range is split into "pages" of size k, and the k possible positions for a key are always the positions of a single page. In other words, at most k keys of S should map to a single page. Mairson showed that $\Theta(n \log(k)/k)$ bits are necessary and sufficient for this problem. Allowing pages that have only $\Omega(k)$ keys on average does not help: Also in this case there is a lower bound of $\Omega(n \log(k)/k)$ [19]. All these results are for the case where n is not bounded by a function of k; indeed, for $k = \omega(\log n)$ a "paged" perfect hash function requires only $O(\log n + \log w)$ bits of space.

Fiat el al. [10] considered searching of a table that

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

s_0	0001001000000	s_3	0010011000000	s_6	0010011010100	s_9	0010011110110
s_1	0010010101100	s_4	0010011001000	s_7	0010011010101	s_{10}	0100100010000
$\mathbf{s_2}$	**0010010101110**	$\mathbf{s_5}$	**0010011010010**	$\mathbf{s_8}$	**0010011010110**		

Figure 1: A toy example: $S = \{s_0, \ldots, s_{10}\}$ is divided into three buckets of size three (except for the last one that contains just two elements), whose delimiters $D = \{s_2, s_5, s_8\}$ appear in boldface.

may be organized as any permutation of S. They showed that $O(\log n + \log w)$ bits of additional storage are sufficient to achieve constant-time membership search. A subset of the authors [9] later showed, by a nonconstructive argument, that in theory $O(\log w)$ additional bits is sufficient. These methods make use of the fact that information can be encoded as permutations of elements, which is not possible in our setting.

2 Definitions, notation, tools

Sets and integers. We use von Neumann's definition and notation for natural numbers: $n = \{0, 1, \ldots, n-1\}$. We thus freely write $f : m \to n$ for a function from the first m natural numbers to the first n natural numbers. We do the same with real numbers, with a slight abuse of notation, understanding a ceiling operator.

In the following, we will always assume that a universe $u = 2^w$ is fixed. The set u has a *natural order* which corresponds to the string lexicographical order of the w-bit left-zero-padded binary representation. We assume, for sake of simplicity, that w is a power of two.

Given $S \subseteq u$ with $|S| = n$, and given m, an *m-bucket hash function for S* is any function $h : S \to m$. We say that h is *perfect* iff it is injective; h is *minimal perfect* iff it is injective and $n = m$; h is *monotone* iff $x \leq y$ implies $h(x) \leq h(y)$ for all $x, y \in S$.

Rank and select. We will make extensive use of the two basic blocks of several succinct data structures—rank and select. Given a bit array (or bit string) $\boldsymbol{b} \in \{0, 1\}^n$, whose positions are numbered starting from 0, $\mathrm{rank}_{\boldsymbol{b}}(p)$ is the number of ones up to position p, exclusive ($0 \leq p \leq n$), whereas $\mathrm{select}_{\boldsymbol{b}}(r)$ is the position of the r-th one in $\boldsymbol{b}$, with bits numbered starting from 0 ($0 \leq r < \mathrm{rank}_{\boldsymbol{b}}(n)$). These operations can be performed in constant time on a string of n bits using additional $o(n)$ bits [16, 6]. When $\boldsymbol{b}$ is obvious from the context we shall omit the subscript.

Storing functions. In the rest of the paper will frequently need to associate values to a key set S; more precisely, we will need to store statically an r-bit function $f : S \to 2^r$. This problem has recently received renewed attention [8, 5, 23]. However, for the purposes of this paper we resort to the classical solution, which is to store a minimal perfect hash function on S and use the resulting value to index a table. Using for example the perfect hash function of Hagerup and Tholey [15], we are able to store an r-bit function in $rn + O(n + \log w)$ bits with constant-time access.

Bucketing. We now discuss here briefly a general approach to monotone minimal perfect hashing that we will use in this paper and that will be referred to as *bucketing*. The same idea has been widely used for non-monotone perfect hashing, and its extension proves to be fruitful.

Suppose you want to build a minimal perfect monotone hash function for a set S; you start with:

- a monotone hash function $d : S \to m$ mapping S to a space of m *buckets*, called the *distributor*;
- for each $i \in m$, a monotone minimal perfect hash function g_i on $d^{-1}(i)$;
- a function $s : m \to n$ such that $s(i) = \sum_{j<i} |d^{-1}(j)|$ for each $i \in m$.

Then, the function $h : S \to n$ defined by $h(x) = s(d(x)) + g_{d(x)}(x)$ is a monotone minimal perfect hash function for S. The idea behind bucketing is that the distributor will consume little space (as we do not require minimality or perfection), and that the functions hashing monotonically each element in its bucket will consume little space if the bucket size is small enough. If the sets $d^{-1}(i)$ are all of the same size, the function s can of course be omitted.

3 Bucketing with longest common prefixes

Our first monotone minimal perfect hash function (which will also be used as a building block in the rest of the paper) is based on *longest common prefixes*. This technique has the advantage of requiring just the evaluation of a fixed number of hash functions; on the other hand, it has the highest memory occupancy among the algorithms we discuss.

We use the bucketing approach described in Section 2. Let b be a positive integer, and divide the set S into buckets of size b preserving order. In other words, let $B_0, B_1, \ldots, B_{m-1}$ be the unique partition of S such that $m = \lceil n/b \rceil$, $|B_i| = b$ for all $i = 0, \ldots, m-2$ and, if $x \in B_i$ and $y \in B_j$ with $i < j$, then $x < y$. We start with a simple lemma:

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

s_0	2	s_4	8	s_8	11
s_1	2	s_5	8	s_9	1
s_2	2	s_6	11	s_{10}	1
s_3	8	s_7	11		

(a)

00	0
00100110	1
00100110101	2
0	3

(b)

Figure 2: Bucketing with least common prefix for the set S of Figure 1: (a) d_0 maps each element x of S to the length of the least common prefix of the bucket to which x belongs; (b) d_1 maps each least common prefix to the bucket index.

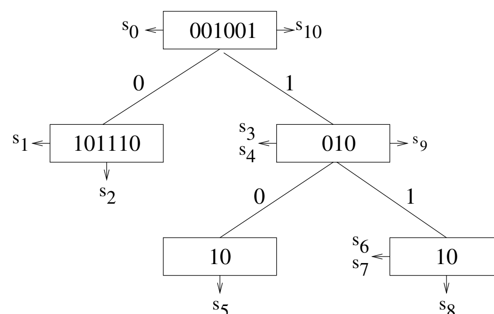

Figure 3: The standard compacted trie built from the set D of Figure 1. This data structure can be used to rank arbitrary elements of the universe with respect to D: when the trie is visited with an element not in D, the visit will terminate at an *exit node*, determining that the given element is to the left (i.e., smaller than) or to the right (i.e., larger than) all the leaves that descend from that node. The picture shows, for each element of S, the node where the visit would end.

LEMMA 3.1. *For $i = 0, \dots, m-1$, let p_i be the longest common prefix of B_i. Then all the p_i's are distinct.*

Proof. Suppose by contradiction that two longest common prefixes p_i and p_j for B_i and B_j, where $j > i$, are equal. If either $|B_i| = 1$ or $|B_j| = 1$ the two prefixes are obviously distinct. Otherwise, we examine the first and last keys of bucket i. Let $p = p_i = p_j$; we know that the last key of bucket i will begin with the string $p1$ and first key of bucket j will being with $p0$; that is because we know that the two keys necessarily differ in the bit immediately after the longest common prefix and last key is lexicographically larger than first key. Now since the bucket j has the same longest common prefix, its first key also begins with the string $p0$. This will mean that the first key of bucket j is lexicographically smaller than last key of bucket i (which begins with $p1$), leading to a contradiction. ∎

Thus, we will represent each bucket by the longest common prefix of its strings. To associate with each string the longest common prefix of the respective bucket, we simply store a function $d_0 : S \to w$ that assigns, to each $x \in S$, the length of the longest common prefix of the bucket containing x. Then, we store a function $d_1 : \{ p_0, p_1, \dots, p_{m-1} \} \to m$ mapping p_i to i.

To compute $d(x)$ for a given $x \in S$, we first apply d_0 obtaining the length ℓ of the longest common prefix of its bucket; from ℓ can compute the prefix (which is made of the first ℓ bits of x), and finally, applying d_1, we obtain $d(x)$. Figure 2 displays the functions d_0 and d_1 for the example of Figure 1 when $b = 3$.

The function d_0 requires $O(n \log w)$ bits, whereas d_1 requires $O((n/b)\log(n/b) + \log w)$ bits; all g_is together require $O(n \log b + \log w)$ bits. By choosing $b = \log n$ we thus obtain a space bound of $O(n(\log\log n + \log w)) = O(n \log w)$ bits, and evaluation is clearly constant time. We have shown the following.

THEOREM 3.1. *There is a monotone minimal perfect hash function that occupies $O(n \log w)$ bits and answers queries in constant time.*

4 Bucketing by relative ranking

In search for a better space bound, we note that an obvious approach to the bucketing problem is by *ranking*. Given a set of strings X, a ranking data structure provides, for each string $s \in u$, the number of strings in X that are smaller than s, that is, $|\{ x \in X \mid x < s \}|$. Consider the set D of *delimiters*, which is made of the lexicographically last string of each bucket. Clearly, the rank of an arbitrary string with respect to D is exactly the index of its bucket.

For instance, a trivial way to obtain such rank information is to build a compacted trie [17] containing the strings in D (see Figure 3). Much more sophisticated data structures are obviously available today (e.g., [14]), but they all fail to meet their purpose in our case: They occupy too much space, and we do not really need to rank *all* possible strings: we just need to rank strings in S. We call this problem the *relative ranking problem*: given sets $D \subseteq S \subseteq u$, we want to rank a string s w.r.t. D under the condition that s belongs to S.

The idea we use to solve this problem is to mimick the behaviour of a trie-based distributor: all we need to know is, for each node and for each key, which is the last node reached during a search (the *exit node*), what behaviour is exhibited at the exit node (exit on the left or right), and finally what is the rank associated to a given exit node and exit behaviour (e.g., the number of leaves on the left of, or on the left of and under, a given node). As we will see, this information can be coded in very little space.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

T			
	0010	→	$\langle 6, h(001001)\rangle$
	00100110	→	$\langle 10, h(0010011010)\rangle$

P	$\boldsymbol{b}$
001001**0**000000	1
001001**1**000000	0
0010011010**0**00	1
0010011010**1**00	1
0010011011000	0
0010100000000	0

Figure 4: The data making up a probabilistic z-fast trie based on the delimiter set D of Figure 1, and the associated ranking structure described in the proof of Theorem 4.2. Above, the map T, representing (just) the internal nodes of the compacted trie shown in Figure 3. Below, the set P and the associated bit vector.

4.1 A probabilistic trie.

In this section, we introduce a Monte-Carlo randomized data structure representing very compactly a trie with errors. To explain how this structure works, let us first introduce some notation and definitions related to compacted tries.

Notation for compacted tries. Consider the compacted trie built for the set $D \subseteq u$; the *string represented by a node* α is defined as the longest common prefix of all keys that are stored in the subtree rooted at α. Note that the association between nodes and strings is bijective: thus, we shall indifferently say that a string is represented by a node, or that the node represents the string. The *skip interval* of a node α representing the string p is defined as follows:

- if α is the root, the skip interval of α is $[0\,..\,|p|)$;
- otherwise, the skip interval of α is $[|q|\,..\,|p|)$, where q is the string represented by the parent of α.

The *path leading to node* α is defined as ε for the root, and as $p0$ and $p1$ for the left and right child of a node representing p, respectively. Note that the path leading to α is always a prefix of the string represented by α.

The main idea behind a z-fast trie is that, instead of representing explicitly a binary tree structure containing compacted paths, we will provide, given an input x, the longest string p represented by an internal node of the trie that is a prefix of x. The string p will be represented by the *parent* of the exit node of x. At that point, by inspecting the bit of x of index $|p|$, we shall be able to determine the edge leading to the exit node of x. The aim of this section is to show the following theorem.

THEOREM 4.1. *Consider the compacted trie for the set $D \subseteq u$, $|D| = m$; there exists a probabilistic data structure that, given an $x \in u$, will return with error probability ε the (length of) the string p represented by the unique internal node with the following properties:*

- *p is a prefix of x;*
- *for every other string q represented by an internal node, if q is a prefix of x, then $|q| < |p|$.*

The structure requires $O(m(\log w + \log(1/\varepsilon)))$ bits of space and has query time $O(\log w)$.

The *2-fattest* number in a nonempty interval of positive integers is the number in the interval whose binary representation has the highest number of trailing zeros.[1] To describe the probabilistic trie, we need some simple properties of 2-fattest numbers.

LEMMA 4.1. *Given an interval $[x\,..\,y]$ of strictly positive integers:*

1. *Let i be the maximal number such that there exists an integer b satisfying $2^i b \in [x, y]$. Then b is unique, and the number $2^i b$ is the* 2-fattest number *in $[x\,..\,y]$.*
2. *If $y - x < 2^i$, there exists at most a single value b such that $2^i b \in [x\,..\,y]$.*
3. *If i is such that $[x\,..\,y]$ does not contain any value of the form $2^i b$, then $y - x + 1 \le 2^i - 1$ and the interval may contain at most one single value of the form $2^{i-1} b$.*

Proof. We only prove (1), the other statements being trivial. Suppose that we have two distinct a and b with $b > a$ satisfying conditions $2^i a \in [x\,..\,y]$ and $2^i b \in [x\,..\,y]$. Since a and b are both odd, there is some c such that $a \le 2c \le b$. So we have $2^{i+1} c \in [x\,..\,y]$, contradicting the definition of i. ∎

A *probabilistic z-fast trie* is given by a function T (see Section 2) providing the following mapping: for every node α of the compacted trie built on D, let p be the string represented by α, $[\alpha_\ell\,..\,\alpha_r)$ the skip interval of α, and f the 2-fattest number in $(\alpha_\ell\,..\,\alpha_r]$ (note the change); if the interval is empty, which can happen only at the root, we set $f = 0$. Then, T maps the prefix of p of length f to the following data (see Figure 4):

1. the length of p ($\log w$ bits);

[1] We remark the fact that the 2-fattest number in the interval $(x\,..\,y]$ is $y \,\&\, -1 \ll \mathrm{MSB}(x \oplus y)$.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

2. a signature of p of length $\log\log w + \log(1/\varepsilon)$ computed using universal hashing [3].

The above function uses $O(m(\log w + \log(1/\varepsilon)))$ bits. We call the exact version, where the "signature" is the identity function, a *z-fast trie.*[2]

The data structure is queried as in Algorithm 1. The idea is to do a kind of binary search for the point where x exits the trie. Instead of testing against the arithmetic mean of the interval endpoints as in a traditional binary search, we test against the 2-fattest number in the interval (*fat binary search*). This ensures that T contains information to guide the search in a suitable way. We use the prefix q of x whose length is the 2-fattest number in the search interval as an input to T. If q is a prefix of a key in S we check whether there is a longer prefix matching x, corresponding to the branching node below q. If both these conditions are satisfied we update the lower bound ℓ, otherwise (assuming there is no false match on the signature) we may correctly update the upper bound r.

For instance, when querying the structure shown in Figure 4 with the string s_1 of Figure 1 we would first compute $T(00100101)$, as the 2-fattest number in $(0\,..\,13)$ is 8. Assuming that the signature correctly identifies our query as a failure, we would try again with $T(0010)$, and this time the signature would match, telling us that the parent of the exit node of s_1 represents the string 001001. By adding a further bit of s_1, we would conclude that 001001**0** is the path leading to the exit node of s_1

To determine the correctness and complexity of the algorithm, we first need some lemmata:

LEMMA 4.2. *The following invariants hold before and after each iteration of the loop in Algorithm 1:*

1. *There exists at most a single b such that $2^i b \in (\ell\,..\,r)$.*
2. *There exists no b such that $2^{i+1} b \in (\ell\,..\,r)$.*
3. *The length of the interval $(\ell\,..\,r)$ is less than 2^i.*

Proof. (1) Initially, when $i = \log w - 1$ we have $(\ell\,..\,r) = (0\,..\,w)$, and this interval contains a single value of the form $2^i b$, that is $w/2$. Now after some iteration suppose that we have at most a single b such that $2^i b \in (\ell\,..\,r)$. We have two cases:

- There is no b such that $2^i b \in (\ell\,..\,r)$. Then, the interval remains unchanged and, by Lemma 4.1 (3), it will contain at most a single value of the form $2^{i-1}b$.
- There is a single b such that $2^i b \in (\ell\,..\,r)$. The interval may be updated in two ways: either we set the interval to $(g\,..\,r)$ for some $g \geq 2^i b$ or we set the interval to $(\ell\,..\,2^i b)$. In both cases, the new interval will no longer contain $2^i b$. By invariant 3. of Lemma 4.1, the new interval will contain at most a single value of the form $2^{i-1}b$.

(2) Initially this is obviously true as the interval $(0\,..\,w)$ does not contain any value of the form bw. By (1), at the beginning of the iteration there was at most a single value of the form $2^j b$ in the interval $(\ell\,..\,r)$ and this single value is clearly eliminated from the interval at the end of the iteration.
(3) By the same argument as in (1) the interval will not contain a value of the form $2^i b$. Applying Lemma 4.1 (3) we deduce that length of interval is at most 2^{i-1}. ∎

Algorithm 1 Querying the probabilistic z-fast trie (represented by the function T).

```
input x ∈ u
i ← ⌈log w⌉ − 1
ℓ, r ← 0, w
while r − ℓ > 1 do
  if ∃b such that 2^i b ∈ (ℓ..r) then
              {2^i b is the 2-fattest number in (ℓ..r)}
    q ← prefix of x of length 2^i b
    ⟨g, s⟩ ← T(q)
    if g ≤ |x| and s is the signature of the prefix of
    x of length g then
      ℓ ← g              {Move from (ℓ..r) to (g..r)}
    else
      r ← 2^i b          {Move from (ℓ..r) to (ℓ..2^i b)}
    end if
  end if
  i ← i − 1
end while
return ℓ
```

The third invariant implies that the algorithm terminates when $i = 0$. Since i is initially $\log w - 1$ (an integer) and decreases by 1 in each iteration the algorithm never performs more than $\log w - 1$ iterations.[3]

[2] We remark that for the purposes of this paper it is sufficient to build a (probabilistic) z-fast trie only on internal nodes (leaves are necessary only for membership test in D, which we do not need). This simplification is tacitly applied in the rest of the paper.

[3] We remark that given the invariants established by Lemma 4.2 the test for the existence of a b such that $2^i b \in (\ell\,..\,r)$ can be replaced by the constant-time test $(1 \ll i) \mathbin{\&} \ell \neq (1 \ll i) \mathbin{\&} (r-1)$.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

LEMMA 4.3. *Let $X = \{ x_0 = \varepsilon, x_1, \dots, x_t \}$, where $x_1, x_2, \dots, x_t$ are the strings represented by nodes of the trie that are prefixes of x, ordered by increasing length. Suppose that there are no false positives in signature comparison (i.e., suppose that two matching signatures on two strings always imply equality of the two strings). Let $(\ell \mathinner{..} r)$ the interval maintained by the algorithm. Before and after each iteration the following invariant is satisfied: $\ell = |x_j|$ for some j, and $\ell \leq |x_t| < r$.*

Proof. We note that the invariant is trivially true at the start, as the initial interval is $(0 \mathinner{..} w)$. By Lemma 4.1 and 4.2 (1), at each step either we do nothing or we pick the 2-fattest number $g \in (\ell \mathinner{..} r)$, and change interval. We have two cases (we follow the notation of Algorithm 1):

- If $T(q)$ gives a positive result, we change our current interval to $(g \mathinner{..} r)$. We know that there is at least a string of length g that is a prefix of x, so certainly $g = |x_j|$ for some j, and $g \leq |x_t|$, and the invariant is preserved.
- If $T(q)$ gives a negative result, we know for sure that no prefix of x longer than $2^i b - 1$ can be represented by a node of the trie: otherwise, $2^i b$ would belong to the skip interval $[\alpha_\ell \mathinner{..} \alpha_r)$ of a node α representing some element of X, and $(\alpha_\ell \mathinner{..} \alpha_r]$ would be entirely contained in $(\ell \mathinner{..} r)$ (as $\ell = |x_j|$ for some j, and $|x_t| < r$). Thus, $2^i b$, being 2-fattest in $(\ell \mathinner{..} r)$, would be *a fortiori* 2-fattest in $(\alpha_\ell \mathinner{..} \alpha_r]$, and $T(q)$ would have returned a positive result—a contradiction. ∎

Thus, the algorithm is correct if there are no false positives, because at the end $\ell = |x_t|$. We now show that it fails with probability bounded by ε. Since we store T using perfect hashing, the signature returned by T is the signature of some string p that is in the domain on which T is defined. Therefore the probability of a false match is $2^{-\log\log w - \log(1/\varepsilon)} = \varepsilon / \log w$. The algorithm makes at most $\log w$ signature comparisons, so we conclude that the combined probability of having one or more false positives is bounded by ε. This concludes the proof of Theorem 4.1.

Variable-length keys Our algorithm works even for variable-length keys. However, care must be taken to get the best possible query time dependency on the key length. The problem is that hashing a key of l bits takes time $O(1 + l/w)$ if done naively. Our solution is to start each query by computing an $O(\log n)$-bit hash value for each word-aligned prefix of the query key. This can be done in time $O(1 + l/w)$ using an incremental hashing method, for instance the one in [7, Section 5]. With high probability there will be no collisions. Using the precomputed table, subsequent hash function evaluations can be done in constant time, meaning that the search itself uses time $O(\log l)$.

4.2 Ranking with errors using a probabilistic trie.

Using a probabilistic trie we can show that

THEOREM 4.2. *Let $D \subseteq u$ (with $|D| = m$) and $\varepsilon > 0$. There exists a data structure that for every $x \in u$ returns with error probability ε a pair of integers $\langle i, j \rangle$, one of which is the rank of x in D (i.e., $|\{ t \in D \mid t < x \}| \in \{ i, j \}$). The data structure uses $O(m(\log w + \log(1/\varepsilon)))$ bits of space and has query time $O(\log w)$.*

Proof. We build a probabilistic z-fast trie on D, and define a set P that, for each string p corresponding to an internal node in the compacted trie for D, contains the following bit strings of length w: $p00^{w-|p|-1}$, $p10^{w-|p|-1}$ and $p10^{w-|p|-1} + 0^{|p|}10^{w-|p|-1}$ (the latter only if $p \neq 111\cdots1$; plus denotes the standard arithmetic operator). We build a constant-time monotone minimal perfect hash function f on P (see Section 3), and we consider a bit vector $\boldsymbol{b}$ of $|P| = O(m)$ elements, endowed with a ranking structure, that has a bit set at position $f(x)$ for each $x \in P$ that leads to a leaf.

We note that, under this arrangement, if p is the string represented by an internal node α, by ranking $f(p00^{w-|p|-1})$ in $\boldsymbol{b}$ we obtain the number of elements of D smaller than every string starting with $p0$; by ranking $f(p10^{w-|p|-1})$ we obtain the number of elements of D smaller than every string starting with $p1$; by ranking $p10^{w-|p|-1} + 0^{|p|}10^{w-|p|-1}$ we obtain the the number of elements of D that start with $p1$ or are smaller than $p1$. Thus, to return the correct result for an input $x \in u$, we proceed as follows: we query the probabilistic trie, obtaining (the length of) a string p. Then, we examine the bit of x of index $|p|$: if it is zero, we return $\langle \mathrm{rank}_{\boldsymbol{b}}(f(p00^{w-|p|-1})), \mathrm{rank}_{\boldsymbol{b}}(f(p10^{w-|p|})) \rangle$; if it is one, but $p \neq 111\cdots1$, we return $\langle \mathrm{rank}_{\boldsymbol{b}}(f(p10^{w-|p|-1})), \mathrm{rank}_{\boldsymbol{b}}(f(p10^{w-|p|-1} + 0^{|p|}10^{w-|p|-1})) \rangle$; otherwise, we return $\langle \mathrm{rank}_{\boldsymbol{b}}(f(p10^{w-|p|-1})), m \rangle$. In each case, the first or the second answer are correct if x exits on the left or right, respectively, of α. ∎

Continuing our example, once we know that 001001**0** is the path leading to the exit node of s_1, by ranking the positions f(001001**0**000000) and f(001001**1**000000) we would know that the correct ranking for s_1 is either 0 or 1.

4.3 A lower bound for ranking with errors.

Theorem 4.2 can be interpreted as claiming that there

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

exists a data structure that is able to rank with respect to a set of m elements with success probability $1/2-\varepsilon$, requiring $O(m(\log w+\log(1/\varepsilon)))$ bits of space and time $O(\log w)$. This should be compared with the following lower bound:

THEOREM 4.3. *Let $D \subseteq u$ with $|D| = m \le 2^{w/3}$ and $\varepsilon < 1/2 - \Omega(1)$. Every probabilistic data structure that ranks D (i.e., that given $x \in u$ computes $|\{s \in D \mid s < x\}|$) with error probability bounded by ε requires in expectation at least*

$$\Omega\left(\frac{m\log(2^w/m)}{1+\log(w)/\log(1/\varepsilon)}\right) \text{ bits} \quad .$$

Proof. For sake of simplicity, assume that $1^w \notin D$. Suppose you have a probabilistic data structure $\mathscr{A}$ that ranks D with error probability $\varepsilon < 1/2-\Omega(1)$ and using s bits; let $\delta = 1/(2\varepsilon)-1$ and k be the smallest integer satisfying

$$\left(\frac{e^{1+\delta}}{(1+\delta)^{1+\delta}}\right)^{\varepsilon k} < \frac{1}{2w}.$$

Now, build a new data structure $\mathscr{B}$ that uses k independent instances of $\mathscr{A}$, and ranks D by using a majority criterion (if more than $k/2$ instances give the same output, $\mathscr{B}$ produces that output; otherwise, it produces a random value). By Chernoff bounds, $\mathscr{B}$ has an error probability bounded by $1/(2w)$: indeed, letting X be the number of instances that fail

$$P[\mathscr{B}\text{ fails}] \le P[X > k/2] = P[X > k(1+\delta)\varepsilon]$$
$$< \left(\frac{e^{k(1+\delta)-1}}{k(1+\delta)^{k(1+\delta)}}\right)^{\varepsilon} \le \left(\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\varepsilon k} < \frac{1}{2w}.$$

Given any i, we can use $\mathscr{B}$ to compute the element $x_i \in D$ whose rank in D is i: we proceed with a binary search on $u = 2^w$, that requires at most w steps. Since at every step the probability of error is less than $1/(2w)$, with probability at least $1/2$ we will compute the correct element x_i. Let D' be the set of elements obtained in this way, as i ranges from 0 to $m-1$: since for every i the probability that we obtain the correct element is $1/2$ or more, we have $|D \cap D'| \ge m/2$ in expectation.

The data structure $\mathscr{B}$ occupies ks bits, and implicitly encodes D'. With m additional bits (storing, for each i, whether x_i was correctly found or not), we can get $D \cap D'$. To obtain D from this we must store the set $D \setminus D'$ (that contains $m/2$ elements in expectation), requiring no more than $mw/2$ bits for its description. So we use

$$ks + mw/2 + O(m)$$

bits to represent D; since, by Kolmogorov complexity, D requires $m\log(2^w/m)+O(m) = mw - m\log m + O(m)$ bits to be represented, and $k = O(1+\log_{1/\varepsilon} w)$, we have (in expectation)

$$s \ge \frac{1}{k}\left(\frac{mw}{2} - m\log m + O(m)\right)$$
$$= \frac{m}{2k}\left(w - 2\log m + O(m)\right).$$

Since $2^w \ge m^3$, that is $w \ge 3\log m$, we have $w - 2\log m \ge w/2 + 3/2\log m - 2\log m = 1/2(w-\log m)$ so

$$s \ge \frac{m}{4k}\left(w - \log m + O(m)\right) = \Omega\left(\frac{m\log(2^w/m)}{1+\log(w)/\log(1/\varepsilon)}\right). \blacksquare$$

If we want the error probability to be, say, $\varepsilon \le 1/w$, we must essentially use $m\log(2^w/m)$ bits, that is, the *number of bits that are necessary to specify D*. The same is true if we want constant error probability, up to a $\log w$ multiplicative factor.

This result should be contrasted with the famous result of Bloom about the existence of a Monte-Carlo data structure for *membership* (with one-sided error) requiring $O(m\log(1/\varepsilon))$ bits, with a compression factor of $\Theta(\log(u/m)/\log(1/\varepsilon))$.

5 A relative trie

The probabilistic trie of Theorem 4.1 has some probability of failing to retrieve the correct node; this can be avoided if we know for sure that the structure is only queried for keys in a prescribed set $S \supseteq D$: we speak in this case of a *relative trie*. To implement a relative trie, we first solve the *relative membership* problem: given sets $E \subseteq S \subseteq u$, provide a succinct data structure that answers queries about membership in E in constant time, where the answer is guaranteed to be correct only for keys is in S.

THEOREM 5.1. *Given sets $E \subseteq S \subseteq u$ (with $|E| = t$, $|S| = n$), constant-time membership in E for elements of S can be implemented in $t\log(n/t)+O(t+\log w)$ bits.*

Proof. Our data structure is defined as follows: we store the elements of D in a constant-time approximate membership structure with error probability m/n. Concrete space-efficient implementation consists of storing a set of signatures [4] in a succinct dictionary [22] or in a hash table according to a minimal perfect hash function for D. Both methods require $m\log(n/m)+O(m+\log w)$ bits. The expected number of false positives in S is $O(n(m/n)) = O(m)$. By building the data structure an expected constant number of times we can get a worst-case bound of $O(m)$ false positives. Then, we store a static 1-bit function recording, for each element of S accepted by the approximate membership structure, whether it actually belongs to D. At query time,

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

we first interrogate the approximate membership structure; should the answer be positive, we double check using the 1-bit function. ∎

Theorem 5.2. *Consider the compacted trie for the set $D \subseteq S \subseteq u$, with $|D| = m$ and $|S| = n$; there exists a data structure that, given an arbitrary key $x \in S$, will return the (length of the) string p represented by an internal node with the following properties:*

- *p is a prefix of x;*
- *for every string q represented by an internal node, if q is a prefix of x, then $|q| \leq |p|$.*

The structure requires $O(m(\log(n/m) + \log w))$ bits of space and has query time $O(\log w)$.

Proof. We set $\varepsilon = m/n$ and build the probabilistic trie described in Theorem 4.1, which will require $O(m(\log(n/m)+\log w))$ bits. Let now $E \subseteq S$ be the set of elements of S that the structure misclassifies: since the data structure fails with probability ε, the expected size of E is $n\varepsilon = m$. Again, this can be turned into a worst-case guarantee of $O(m)$ by allowing the construction time to be a random variable with expected value $O(1)$ times larger. We now store the set E relatively to S, which requires $m \log(n/m) + O(m + \log w)$ by Theorem 5.1, and store explicitly the answer for all elements of E, which requires $O(m \log w)$ bits. Then, when we want to compute the answer for a key $x \in S$, we will first test whether $x \in E$, and, in that case, return the stored answer; otherwise, we query the probabilistic trie. ∎

5.1 Relative ranking without errors. The structure described in Theorem 5.2 makes the probabilistic trie exact, at the expense of giving a correct result only on elements of S. By coupling this with the ranking structure described in the proof of Theorem 4.2, we obtain a structure that provides (deterministically and without failures) two possible values for the rank of an element of S. At this point, we just have to record the subset $L \subseteq S$ for which the first integer of the pair is correct (i.e., essentially, which strings exit on the left). This can be done using a relative membership data structure (actually, even using a 1-bit function), leading to:

Theorem 5.3. *Let $D \subseteq S \subseteq u$ (with $|D| = m$ and $|S| = n$). There exists a data structure that, for every $x \in S$, returns the rank of x in D, requiring $O(m(\log(n/m) + \log w) + n)$ bits of space and providing query time $O(\log w)$.*

6 Monotone hashing with $O(n \log\log w)$ space

We finally show how to obtain a monotone minimal perfect hash function using just $O(n \log\log w)$ bits, and getting query time $O(\log w)$. We follow the setup of Section 2 with buckets of size $b = \log w$. The relative trie on the $n/\log w$ delimiters will occupy just $O(n)$ bits. Adding the space occupancy of the functions g_i ($O(n \log\log w)$ bits), we have the following

Theorem 6.1. *There is a monotone minimal perfect hash function that occupies $O(n \log\log w)$ bits of space and answers queries in time $O(\log w)$.*

7 Indexing a sorted table

The results we have proved on monotone minimal perfect hashing imply that we can find an element in a sorted table with just one access. In this section we provide further results on the indexing problem.

7.1 Ranking a sorted table with $O(1)$ accesses in expectation. Suppose we have an offline storage containing a (sorted) set of keys S. Our goal is to rank an element $x \in u$ using an expected constant number of accesses to the set S, and as little additional space as possible. The idea is to use a structure providing relative ranking with respect to S with errors, and check for errors using probes on S.

The error probability should be $o(1/w)$, which is possible without any cost in asymptotic space usage. This means that with probability $1 - o(1/w)$ we obtain two possible values i and j, $i < j$, such that either the element of rank $i - 1$ or the element of rank $j - 1$ in S is the predecessor of x (if $i = 0$ or $j = 0$ then x is before all strings in S). Using at most 3 probes we can determine whether the predecessor of x is indeed at rank $i - 1$ or $j - 1$: First probe the i-th element, and depending on its relation to x probe either position $i - 1$ or positions $\{ j - 1, j \}$. If this fails to identify the predecessor, we simply perform a binary search on S which requires $\log(n + 1) = O(w)$ probes. Since the binary search happens with probability $o(1/w)$, the expected contribution of this to the overall value is $o(1)$.

Theorem 7.1. *Given a stored sorted table S of n elements, there is a structure using additional $O(n \log w)$ space that computes the rank of an element $x \in u$ in S using $3 + o(1)$ accesses to S in expectation.*

7.2 Finding an element in blocked storage using at most two accesses. In this case, our sorted table S is divided in blocks of size b, and we want to find the block in which a key $x \in S$ is located reading as few blocks as possible. As done at the start of Section 5.1, we can build a relative trie structure on the delimiter set D (as usual, containing the last key of each block) paired with a node-ranking structure that will locate two possible blocks to which x might belong. The

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

structure will use $O(n/b(\log b + \log w) + n/b)$ bits of space, which for $b = O(w^c)$ is $O((n/b) \log w)$ (to be compared with the information-theoretical lower bound of $\Omega((n/b)(w - \log(n/b)))$ bits that are necessary to store the delimiters). After accessing the structure, we have at most two possible candidates, and in expectation we will detect the right one in 1.5 attempts.

We propose the name *Bee-tree* for an external memory search tree that uses probabilistic tries to guide searches. For some parameters, Bee-trees have depth that is asymptotically smaller than normal B-trees.

References

[1] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 53(2):115–136, 2004.

[2] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 39–48, New York, Jan. 6–8 2002. ACM Press.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18:143–154, 1979.

[4] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of Symposium on Theory of Computation (STOC '78)*, pages 59–65. ACM Press, 1978.

[5] D. Charles and K. Chellapilla. Bloomier filters: A second look. In *Proc. ESA 2008*, 2008.

[6] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391, 1996.

[7] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[8] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[9] A. Fiat and M. Naor. Implicit $O(1)$ probe search. *SIAM Journal of Computing*, 22(1):1–10, 1993.

[10] A. Fiat, M. Naor, J. P. Schmidt, and A. Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, 1992.

[11] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Sys.*, 9(3):281–308, 1991.

[12] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM J. Algebraic Discrete Methods*, 5(1):61–68, 1984.

[13] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, July 1984.

[14] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoret. Comput. Sci.*, 387(3):313–331, 2007.

[15] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS '01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. Springer–Verlag, 2001.

[16] G. Jacobson. Space-efficient static trees and graphs. In *In Proc 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[18] H. G. Mairson. The program complexity of searching a table. In *24th Annual Symposium on Foundations of Computer Science*, pages 40–47, Tucson, Arizona, 7–9 Nov. 1983. IEEE.

[19] H. G. Mairson. The effect of table expansion on the program complexity of perfect hash functions. *BIT*, 32(3):430–440, Sept. 1992.

[20] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996.

[21] K. Mehlhorn. On the program size of perfect and universal hash functions. In *23th Annual Symposium on Foundations of Computer Science (FOCS 1982)*, pages 170–175, Chicago, Illinois, USA, 1982. IEEE.

[22] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[23] E. Porat. An optimal bloom filter replacement based on matrix solving, 2008. arXiv:0804.1845v1.

[24] V. Raman. Locality preserving dictionaries: theory & application to clustering in databases. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 337–345, New York, NY, USA, 1999. ACM.

[25] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal of Computing*, 19(5):775–786, 1990.

[26] D. E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[27] A. C.-C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.