On Dynamic Range Reporting in One Dimension

(Extended Abstract) *

Christian Worm Mortensen†
IT U. Copenhagen
cworm@itu.dk

Rasmus Pagh
IT U. Copenhagen
pagh@itu.dk

Mihai Pătraşcu
MIT
mip@mit.edu

ABSTRACT

We consider the problem of maintaining a dynamic set of integers and answering queries of the form: report a point (equivalently, all points) in a given interval. Range searching is a natural and fundamental variant of integer search, and can be solved using predecessor search. However, for a RAM with w-bit words, we show how to perform updates in $O(\lg w)$ time and answer queries in $O(\lg \lg w)$ time. The update time is identical to the van Emde Boas structure, but the query time is exponentially faster. Existing lower bounds show that achieving our query time for predecessor search requires doubly-exponentially slower updates. We present some arguments supporting the conjecture that our solution is optimal.

Our solution is based on a new and interesting recursion idea which is "more extreme" that the van Emde Boas recursion. Whereas van Emde Boas uses a simple recursion (repeated halving) on each path in a trie, we use a non-trivial, van Emde Boas-like recursion on every such path. Despite this, our algorithm is quite clean when seen from the right angle. To achieve linear space for our data structure, we solve a problem which is of independent interest. We develop the first scheme for dynamic perfect hashing requiring sublinear space. This gives a dynamic Bloomier filter (a storage scheme for sparse vectors) which uses low space. We strengthen previous lower bounds to show that these results are optimal.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: sorting and searching; E.2 [**Data Storage Representations**]: hash-table representations

*A version of this paper containing all proofs is available as `arXiv:cs.DS/0502032`.

†Part of this work was done while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, as a Marie Curie doctoral fellow.

STOC'05, May 22-24, 2005, Baltimore, Maryland, USA.

General Terms

Algorithms, Performance, Theory

Keywords

range reporting, integer search, perfect hashing, Bloomier filters

1. INTRODUCTION

Our problem is to maintain a set S under insertions and deletions of values, and a range reporting query. The query $\text{FINDANY}(a, b)$ should return an arbitrary value in $S \cap [a, b]$, or report that $S \cap [a, b] = \emptyset$. This is a form of existential range query. In fact, since we only consider update times above the predecessor bound, updates can maintain a linked list of the values in S in increasing order. Given a value $x \in S \cap [a, b]$, one can traverse this list in both directions starting from x and list all values in the interval $[a, b]$ in constant time per value. Thus, the FINDANY query is equivalent to one-dimensional range reporting.

The model in which we study this problem is the word RAM. We assume the elements of S are integers that fit in a word, and let w be the number of bits in a word (thus, the "universe size" is $u = 2^w$). We let $n = |S|$. Our data structure uses Las Vegas randomization (through hashing), and the bounds stated will hold with high probability in n.

Range reporting is a very natural problem, and its higher-dimensional versions have been studied for decades. In one dimension, the problem is easily solved using predecessor search. The predecessor problem has also been studied intensively, and the known bounds are now tight in almost all cases [2]. Another well-studied problem related to ours is the lookup problem (usually solved by hashing), which asks to find a key in a set of values. Our problem is more general than the lookup problem, and less general than the predecessor problem. While these two problems are often dubbed "the integer search problems", we feel range reporting is an equally natural and fundamental incarnation of this idea, and deserves similar attention.

The first to ask whether or not range reporting is as hard as finding predecessors were Miltersen et al in STOC'95 [9]. For the static case, they gave a data structure with space $O(nw)$ and constant query time, which cannot be achieved for the predecessor problem with $n^{O(1)}$ space. An even more surprising result from STOC'01 is due to Alstrup, Brodal and Rauhe [1], who gave an optimal solution for the static case, achieving linear space and constant query time. In the dynamic case, however, no solution better than the predecessor problem was known. For this problem, the fastest known solution in terms of w is the classic van Emde Boas structure [13], which achieves $O(\lg w)$ time per operation.

For the range reporting problem, we show how to perform updates in $O(\lg w)$ time, while supporting queries in $O(\lg\lg w)$ time. The space usage is optimal, i.e. $O(n)$ words. The update time is identical to the one given by the van Emde Boas structure, but the query time is exponentially faster. In contrast, Beame and Fich [2, Theorem 3.7] show that achieving any query time that is $o(\lg w/\lg\lg w)$ for the predecessor problem requires update time $\Omega(2^{w^{1-\varepsilon}})$, which is doubly-exponentially slower than our update time. We also give an interesting tradeoff between update and query times; see theorem 4 below.

Our solution incorporates some basic ideas from the previous solutions to static range reporting in one dimension [9, 1]. However, it brings two important technical contributions. First, we develop a new and interesting recursion idea which is more advanced than van Emde Boas recursion (but, nonetheless, not technically involved). We describe this idea by first considering a simpler problem, the bit-probe complexity of the greater-than function. Then, the solution for dynamic range reporting is obtained by using the recursion for this simpler problem, on *every path* of a binary trie of depth w. This should be contrasted to the van Emde Boas structure, which uses a very simple recursion idea (repeated halving) on every root-to-leaf path of the trie. The van Emde Boas recursion is fundamental in the modern world of data structures, and has found many unrelated applications (e.g. exponential trees, integer sorting, cache-oblivious layouts, interpolation search trees). It will be interesting to see if our recursion scheme has a similar impact.

The second important contribution of this paper is needed to achieve linear space for our data structure. We develop a scheme for dynamic perfect hashing, which requires sublinear space. While the static version of the problem is understood, our result is the first sublinear-space solution in the dynamic case. In addition, we prove that our solution is optimal, closing the problem. An important application of perfect hashing is storing a sparse vector in small space, if we are only interested in querying non-null positions (the Bloomier filter problem). The stringent space requirements that our data structure can meet are also important in data-stream algorithms and database systems. We mention one application below, but believe others exist as well.

This extended abstract contains: a statement of our results (in the remainder of the introduction); our upper bound (Section 2) and lower bound (Section 3) for dynamic perfect hashing; and our main result for dynamic range reporting, introduced by the homologous result for the greater-than problem (Section 4).

1.1 Perfect Hashing

Perfect hashing is a fundamental variant of the dictionary problem. The problem is to maintain a set S of keys from the universe $\{1,\dots,u\}$, along with a perfect hash function to a small range (i.e. a function defined on the whole universe, which is one-to-one when restricted to S), and be able to evaluate the function efficiently. Ideally, the range is of size $O(n)$ and evaluation takes constant time. An important application of perfect hashing is Bloomier filters, which we discuss below.

In the static case, a perfect hash function can be represented with only $\Theta(\lg n+\lg\lg u)$ bits of space. Here we are concerned with the dynamic case, where elements can be inserted and deleted from S. An element needs to maintain the same hash value while it is in S. However, if an element is deleted and subsequently reinserted, its hash value may change.

THEOREM 1. *We can maintain a perfect hash function from a set $S\subset\{1,\dots,u\}$ with $|S|\le n$ to a range of size $n+o(n)$, under $n^{O(1)}$ insertions and deletions, using $O(n\lg\lg u)$ bits of space w.h.p., plus a constant number of machine words. The function can be evaluated in worst-case constant time, and updates take constant time w.h.p.*

This is the first dynamic perfect hash function that uses less space than needed to store S ($\lg\binom{u}{n}$ bits). Our lower bound for Bloomier filters (see below) implies that our space bound is essentially optimal regardless of the query and update time. Obtaining sublinear space for the dynamic problem is more surprising than for the static problem. In the static case, a small hash function can be constructed based on S, and then S is discarded. However, a dynamic data structures must assign unique identifiers to all elements in S without ever knowing the current set S.

Such stringent space requirements are typical of data-stream computation, where one needs to support a stream of updates and queries, but does not have space to hold the entire state of the data structure. Interestingly, our solution can achieve this goal without introducing errors (we use only Las Vegas randomization). We mention an independent application of our data structure. In a database we can maintain an index of a relation under insertions and deletions of tuples, using internal memory per tuple which is logarithmic in the length of the key for the tuple. If tuples have fixed length, they can be placed directly in the hash table, and need only be moved if the capacity of the hash table is exceeded.

1.2 Bloomier Filters

The Bloom filter is a classic data structure for testing membership in a set. If a constant rate of false-positives is allowed, the space *in bits* can be made essentially linear in the size of the set. Optimal bounds for this problem are obtained in [10].

Bloomier filters are an extension to Bloom filters for the dictionary problem. As an application of perfect hashing, the problem has been folklore, but this catchy name was only introduced in [3]. The problem is to represent a vector $V[1\,..\,u]$ with elements from $\{0,\dots,2^r-1\}$ which is nonzero in only n places (assume $n\ll u$, so the vector is sparse). Thus, we have a sparse set as before, but with values associated to the elements. The information theoretic lower bound for representing such a vector is $\Omega(nr+\lg\binom{u}{n}) = \Omega(n(r+\lg\frac{u}{n}))$ bits. However, in various applications, we only want correct answers when $V[x]\ne 0$. This can be the case either when we only intend to query nonzero positions, or when we have a way of verifying the correctness of an answer, and we can thus detect zero elements (this is the case in our range reporting structure).

Bloomier filters can be implemented using a perfect hash function, plus $O(nr)$ bits of space. In the static case, this gives a space of $O(nr+\lg\lg u)$ bits, which is optimal [3]. We are interested in the dynamic case, where the values of

V can change arbitrarily at any point. Our perfect hashing construction implies a bound of $O(n(r+\lg\lg u))$. Previously, no nontrivial upper bound was known. To detect whether $V[x]=0$ with probability of correctness at least $1-\varepsilon$, one can use a Bloom filter on top, which requires space $\Theta(n\lg\frac{1}{\varepsilon})$ even in the dynamic case [10]. Note that even for $\varepsilon=1$, randomization is essential, since any deterministic solution must use $\Omega(\lg\binom{u}{n})$ bits of space, i.e. it must essentially store the set of nonzero entries in the vector.

Our space bound is essentially optimal. It was previously shown in [3] that $\Omega(n(r+\min(\lg\lg\frac{u}{n^3},\lg n)))$ bits of space are needed, regardless of the query and update times. We improve this lower bound to the following:

Theorem 2. *Maintaining a dynamic Bloomier filter for $r\geq 2$ requires $\Omega(n(r+\lg\lg\frac{u}{n}))$ bits of space in expectation, regardless of the query and update time.*

Observe that our lower bound is an improvement both for small universes (below $n^{3+\varepsilon}$), and for large universes (where the previous space bound was capped to $\Omega(n\lg n)$).

1.3 Tradeoffs and the Greater-Than Problem

All tradeoffs, as well as the lower bounds for the greater-than problem, appear in the full version of this paper. In this extended abstract we concentrate on $t_u=O(\lg w)$ and $t_q=O(\lg\lg w)$ for the sake of space and clarity. However, a discussion of the tradeoffs affords some interesting insight into the scheme of things.

We begin with a discussion of the greater-than problem. Consider an infinite memory of bits, initialized to zero. Our problem has two stages. In the update stage, the algorithm is given a number $a\in\{1,\dots,n\}$. After seeing a, the algorithm is allowed to flip $O(t_u)$ bits in the memory. In the query stage, the algorithm is given a number $b\in\{1,\dots,n\}$. Now the algorithm may inspect $O(t_q)$ bits, and must decide whether or not $b>a$. The problem was previously studied by Fredman [6], who showed that $\max(t_u,t_q)=\Omega(\lg n/\lg\lg n)$. It is quite tempting to believe that one cannot improve past the trivial upper bound $t_u=t_q=O(\lg n)$, since, in some sense, this is the complexity of "writing down" a. However, as we show in this paper, Fredman's bound is optimal, in the sense that it is a point on our tradeoff curve. We give upper and lower bounds that completely characterize the possible asymptotic tradeoffs:

Theorem 3. *The bit-probe complexity of the greater-than function satisfies the tight tradeoffs:*

$$t_q\geq\lg\lg n,\ t_u\leq\lg n\quad:\quad t_u=\Theta(\lg_{t_q}n)$$
$$t_q\leq\lg\lg n,\ t_u\geq\lg n\quad:\quad 2^{t_q}=\Theta(\lg_{t_u}n)$$

As mentioned already, we use the same recursion idea for dynamic range reporting, except that we apply this recursion to every root-to-leaf path of a binary trie of depth w. Quite remarkably, the paths structures can be made to overlap in-as-much as the paths overlap, so only one update suffices for all paths going through a node.

An alternative explanation of our idea is rendered by Figure 1. As in the van Emde Boas structure, we represent the input set and the query points as root-to-leaf paths in a trie. We consider a series of "views" of this trie, at different levels of detail. Usually, the next level of detail halves the keys in the current level. The van Emde Boas structure examines each level of detail consecutively, halving w in each step. On the other hand, our query algorithm does a binary search on these representations of the trie, examining just a logarithmic number of them. The recursion on the levels of detail is equivalent to a van Emde Boas recursion on each path, as used by our solution for the greater-than problem. Due to these close relations between the greater-than problem and a natural approach to solve range reporting, we view the lower bounds for the greater-than function as giving an indication that our range reporting data structure is likewise optimal. In any case, the lower bounds show that quite different ideas would be necessary to improve our solution for range reporting.

Let t_{pred} be the time needed by one update and one query in the dynamic predecessor problem. The following theorem summarizes our results for dynamic range reporting:

Theorem 4. *There is a data structure for the dynamic range reporting problem, which uses $O(n)$ space and supports updates in time $O(t_u)$, and queries in time $O(t_q)$, for all t_u,t_q satisfying:*

$$t_q\geq\lg\lg w,\ \frac{\lg w}{\lg\lg w}\leq t_u\leq\lg w\quad:\quad t_u=O(\lg_{t_q}w)+t_{pred}$$
$$t_q\leq\lg\lg w,\qquad t_u\geq\lg w\quad:\quad 2^{t_q}=O(\lg_{t_u}w)$$

Notice that the most appealing point of the tradeoff is the cross-over of the two curves: $t_u=O(\lg w)$ and $t_q=O(\lg\lg w)$ – and indeed, this has been the focus of our discussion. Another interesting point is at constant query time. In this case, our data structure needs $O(w^\varepsilon)$ update time. Thus, our data structure can be used as an optimal static data structure, which is constructed in time $O(nw^\varepsilon)$, improving on the construction time of $O(n\sqrt{w})$ given by Alstrup et al [1].

The first branch of our tradeoff is not interesting with $t_{pred}=\Theta(\lg w)$, as given by the van Emde Boas structure. It may be possible to achieve $t_{pred}=O(\frac{\lg w}{\lg\lg w})$, matching the optimal bound for the static case. If this is true, the t_{pred} term can be ignored. Even without such a result, the first branch of the tradeoff is still interesting, because we can use $t_{pred}=O(\sqrt{\frac{\lg n}{\lg\lg n}})$, which is in many cases $o(\lg w)$.

We can also achieve bounds in terms of n, rather than w, by the classic trick of using our structure for small w and a fusion tree structure [7] for large w. In particular, we can achieve $t_q=O(\lg\lg n)$ and $t_u=O(\frac{\lg n}{\lg\lg n})$. Compared with the $\Theta(\sqrt{\frac{\lg n}{\lg\lg n}})$ bound for the predecessor problem, which holds even with $t_u=n^{O(1)}$, our data structure improves the query time exponentially by sacrificing the update time quadratically.

2. THE PERFECT-HASHING STRUCTURE

We denote by S be the set of values that we need to hash at present time. Our data structure has the following parts:

- A hash function $\rho:\{1,\dots,u\}\to\{0,1\}^v$, where $v=O(\lg n)$, from a family of universal hash functions with small representations (for example, the one from [4]).
- A hash function $\phi:\{0,1\}^v\to\{1,\dots,r\}$, where $r=\lceil n/\lg^2 n\rceil$, taken from Siegel's class of highly independent hash functions [12].

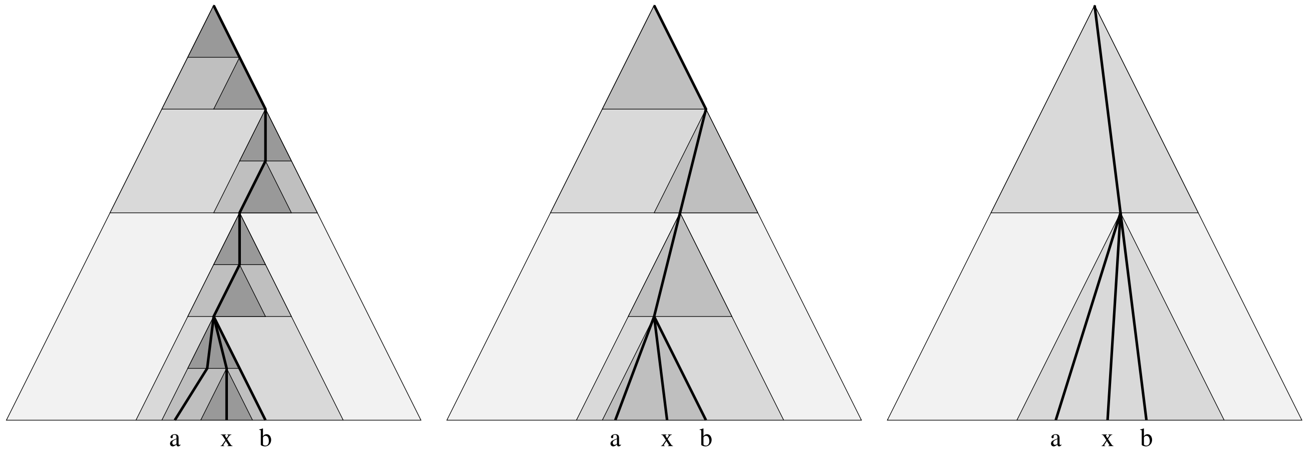

Figure 1: A trie and its representation on several levels of detail.

- An array of hash functions $h_1, \dots, h_r : \{0,1\}^v \to \{0,1\}^s$, where $s = \lceil (6+2c) \lg \lg u \rceil$, chosen independently from a family of universal hash functions; c is a constant specified below.

- A high performance dictionary [5] for a subset S' of the keys in S. The dictionary should have a capacity of $O(\lceil n/\lg u \rceil)$ keys (but might expand further). Along with the dictionary we store a linked list of length $O(\lceil n/\lg u \rceil)$, specifying certain vacant positions in the hash table.

- An array of dictionaries $D_1, \dots, D_r$, where D_i is a dictionary that holds $h_i(\rho(k))$ for each key $k \in S \setminus S'$ with $\phi(\rho(k)) = i$. A unique value in $\{0, \dots, j-1\}$, where $j = (1+o(1)) \lg^2 n$, is associated with each key in D_i. A bit vector of j bits and an additional string of $\lg n$ bits is used to keep track of which associated values are in use. We will return to the exact choice of j and the implementation of the dictionaries.

The main idea is that all dictionaries in the construction assign to each of their keys a unique value within a subinterval of $[1 \,..\, m]$. Each of the dictionaries $D_1, \dots, D_r$ is responsible for an interval of size j, and the high performance dictionary is responsible for an interval of size $O(n/\lg u) = o(n)$. If the number of items in the high performance dictionary exceeds $O(n/\lg u)$, which happens with polynomially small probability, we can revert to a brute-force solution which holds a dictionary with explicit perfect hash codes.

The hash function ρ is used to reduce the key length to v. The constant in $v = O(\lg n)$ can be chosen such that with high probability, over a polynomially bounded sequence of updates, ρ will never map two elements of S to the same value (the conflicts, if they occur, end up in S' and are handled by the high performance dictionary).

When inserting a new value k, the new key is included in S' if either:

- There are j keys in D_i, where $i = \phi(\rho(k))$, or

- There exists a key $k' \in S$ where $\phi(\rho(k)) = \phi(\rho(k')) = i$ and $h_i(\rho(k)) = h_i(\rho(k'))$.

Otherwise k is associated with the key $h_i(\rho(k))$ in D_i. Deletion of a key k is done in S' if $k \in S'$, and otherwise the associated key in the appropriate D_i is deleted.

To evaluate the perfect hash function on a key k we first see whether k is in the high performance dictionary. If so, we return the value associated with k. Otherwise we compute $i = \phi(\rho(k))$ and look up the value Δ associated with the key $h_i(\rho(k))$ in D_i. Then we return $(i-1)j + \Delta$, i.e. position Δ within the i-th interval.

Since $D_1, \dots, D_r$ store keys and associated values each of $O(\lg \lg u)$ bits, they can be efficiently implemented as constant depth search trees of degree $w^{\Omega(1)}$, where each internal node resides in a single machine word. This yields constant time for dictionary insertions and lookups, with an optimal space usage of $O(\lg^2 n \lg \lg u)$ bits for each dictionary. We do not go into details of the implementation as they are standard; refer to [8] for explanation of the required word-level parallelism techniques.

What remains to describe is how the dictionaries keep track of vacant positions in the hash table in constant time per insertion and deletion. The high performance dictionary simply keeps a linked list of all vacant positions in its interval. Each of $D_1, \dots, D_r$ maintain a bit vector indicating vacant positions, and additional $O(\lg n)$ summary bits, each taking the or of an interval of size $O(\lg n)$. This can be maintained in constant time per operation, employing standard techniques.

The preprocessing time is $o(n)$ – essentially to build tables needed for the word-level parallelism. The major part of the data structure is initialized lazily.

2.1 Analysis

Since evaluation of all involved hash functions and lookup in the dictionaries takes constant time, evaluation of the perfect hash function is done in constant time. As we will see below, the high performance dictionary is empty with high probability unless $n/\lg u > \sqrt{n}$. This means that it always uses constant time per update with high probability in n. All other operations done for update are easily seen to require constant time w.h.p.

We now consider the space usage of our scheme. The function ρ can be represented in $O(w)$ bits. Siegel's highly independent hash function uses $o(n)$ bits of space. The hash functions $h_1, \ldots, h_r$ use $O(\lg n + \lg\lg u)$ bits each, and $o(n \lg\lg u)$ bits in total. The main space bottleneck is the space for $D_1, \ldots, D_r$, which sums to $O(n \lg\lg u)$.

Finally, we show that the space used by the high performance dictionary is $O(n)$ bits w.h.p. This is done by showing that each of the following hold with high probability throughout a polynomial sequence of operations:

1. The function ρ is one-to-one on S.
2. There is no i such that $S_i = \{k \in S \mid \phi(\rho(k)) = i\}$ has more than j elements.
3. The set S' has $O(\lceil n/\lg u \rceil)$ elements.

That 1. holds with high probability is well known. To show 2. we use the fact that, with high probability, Siegel's hash function is independent on every set of $n^{\Omega(1)}$ keys. We may thus employ Chernoff bounds for random variables with limited independence to bound the probability that any i has $|S_i| > j$, conditioned on the fact that 1. holds. Specifically, we can use [11, Theorem 5.I.b] to argue that for any l, the probability that $|S_i| > j$ for $j = \lceil \lg^2 n + \lg^{5/3} n \rceil$ is $n^{-\omega(1)}$, which is negligible. On the assumption that 1. and 2. hold, we finally consider 3. We note that every key $k' \in S'$ is involved in an h_i-collision in S_i for $i = \phi(\rho(k'))$, i.e. there exists $k'' \in S_i \setminus \{k'\}$ where $h_i(k') = h_i(k'')$. By universality, for any i the expected number of h_i-collisions in S_i is $O(\lg^4 n/(\lg u)^{6+2c}) = O((\lg u)^{-(2+2c)})$. Thus the probability of one or more collisions is $O((\lg u)^{-(2+2c)})$. For $\lg u \geq \sqrt{n}$ this means that there are no keys in S' with high probability. Specifically, c may be chosen as the sum of the constants in the exponents of the length of the operation sequence and the desired high probability bound. For the case $\lg u < \sqrt{n}$ we note that the expected number of elements in S' is certainly $O(n/\lg u)$. To see than this also holds with high probability, note that the event that one or more keys from S_i end up in S' is independent among the i's. Thus we can use Chernoff bounds to get that the deviation from the expectation is small with high probability.

3. SPACE LOWER BOUND

We now prove that dynamic Bloomier filters require $\Omega(n(r + \lg\lg \frac{u}{n}))$ bits of space, regardless of the query and update times. This implies an $\Omega(n \lg\lg \frac{u}{n})$ bound for dynamic perfect hashing. First observe that when $r \geq \lg\lg \frac{u}{n}$, the lower bound is trivial. Otherwise, we may assume $r = 2$.

Following [3], we consider a two-set distinction problem. The central parameter of this problem (and the target of our lower bound) is the space S, which is easily seen to be bounded by the space for a dynamic Bloomier filter with $r = 2$. The problem has the following stages:

1. a random string R is drawn, which will be available to the data structure throughout its operation. This is equivalent to drawing a deterministic algorithm from a given distribution, and is more general than assuming each stage has its own random coins (we are giving the data structure free storage for its random bits).
2. the data structure is given $A \subset \{1, \ldots, u\}, |A| \leq n$. It must produce a representation $f_R(A)$, such that $(\forall)A, E_R[|f_R(A)|] \leq S$, where $|\cdot|$ denotes the space in bits required by the representation.
3. the data structure is given $B \subset \{1, \ldots, u\}$, such that $|B| \leq n, A \cap B = \emptyset$. Based on the old state $f_R(A)$, the data structure must produce a new state $g_R(B, f_R(A))$ with $E_R[|g_R(B, f_R(A))|] \leq S, (\forall)A, B$.
4. the data structure is given $x \in \{1, \ldots, u\}$ and its previous states $f_R(A)$ and $g_R(B, f_R(A))$. Now it must answer as follows with no error allowed: if $x \in A$, it must answer "A"; if $x \in B$, it must answer "B"; if $x \notin A \cup B$, it can answer arbitrarily.

Let $h_R(x, f_0, g_0)$ be the answer computed by the data structure, when the previous states are f_0 and g_0. Since a solution to the distinction problem is not allowed to make an error, we can assume w.l.o.g. that the computation of $h_R(x, f_0, g_0)$ is implemented as follows. If there exist appropriate A, B, with $x \in A$ and $f_R(A) = f_0,\ g_R(B, f_0) = g_0$, then $h_R(x, f_0, g_0)$ must be "A". Similarly, if there exists a plausible scenario with $x \in B$, the answer must be "B". Otherwise, the answer can be arbitrary.

Assume that the inputs $A \times B$ are drawn from a given distribution. We argue that if the expected sizes of f and g are allowed to be at most $2S$, the data structure need not be randomized. This uses a bicriteria minimax principle. We have $E_{R,A,B}\left[\frac{|f|}{S} + \frac{|g|}{S}\right] \leq 2$. Then, there exists a fixed string R_0 such that $E_{A,B}\left[\frac{|f|}{S} + \frac{|g|}{S}\right] \leq 2$. Since $|f|, |g| \geq 0$, this implies $E_{A,B}[|f|] \leq 2S,\ E_{A,B}[|g|] \leq 2S$. Thus, the data structure can use the deterministic sequence R_0 as its random bits. Below, we drop the subscript from f_{R_0}, g_{R_0}.

3.1 Lower Bound for Two-Set Distinction

Assume $u = \omega(n)$, since a lower bound of $\Omega(n)$ is trivial for universe $u \geq 2n$. Break the universe into n equal parts $U_1, \ldots, U_n$; w.l.o.g. assume n divides u, so $|U_i| = \frac{n}{u}$. The hard input distribution chooses A uniformly at random from $U_1 \times \cdots \times U_n$. We write $A = \{a_1, \ldots, a_n\}$, where a_i is a random variable drawn from U_i. Then, B' is chosen uniformly at random from the same product space; again $B' = \{b_1, \ldots, b_n\}, b_i \leftarrow U_i$. We let $B = B' \setminus A$. Note that $E[|B|] = n \cdot \Pr[A_1 \neq B_1] = (1 - \frac{n}{u}) \cdot n = (1 - o(1)) \cdot n$.

Let A_i^p be the plausible values of a_i after we see $f(A)$; that is, A_i^p contains all $a \in U_i$ for which there exists a valid A' with $a \in A'$ and $f(A') = f(A)$. Intuitively speaking, if $f(A)$ has expected size S, it contains on average S/n bits of information about each a_i. Since the range of a_i is $\frac{u}{n}$, we would expect that the average $|A_i^p|$ is quite large, around $\frac{u}{n}/2^{S/n}$. This intuition is formalized in the following lemma:

Lemma 5. *With probability at least a half over a uniform choice of A and i, we have $|A_i^p| \geq \frac{u/n}{2^{O(S/n)}}$.*

Proof. The Kolmogorov complexity of A is $n \lg \frac{u}{n} - O(1)$; no encoding for A can have an expected size less than this quantity. We propose an encoding for A consisting of two parts: first, we include $f(A)$; second, for each i we include the index of a_i in $|A_i^p|$, using $\lceil \lg |A_i^p| \rceil$ bits. This is easily decodable. We first generate all possible A' with $f(A') = f(A)$, and thus obtain the sets A_i^p. Then, we extract from each plausible set the element with the given index. The expected size of the encoding is $2S + \sum_i E_A[\lg |A_i^p|] + O(n)$, which must be at least $n \lg \frac{u}{n} - O(1)$. This implies $\lg \frac{u}{n} - E_{i,A}[\lg |A_i^p|] \leq \frac{2S}{n} + O(1)$. By Markov's inequality, with probability at least a half over i and A, $\lg \frac{u}{n} - \lg |A_i^p| \leq \frac{4S}{n} + O(1)$, so $\lg |A_i^p| \geq \lg \frac{u}{n} - O(\frac{S}{n})$. □

We now make a crucial observation which justifies our interest in A_i^p. Assume that $b_i \in A_i^p$. In this case, the data structure must be able to determine b_i from $f(A)$ and $g(B, f(A))$. Indeed, suppose we compute $h(x, f, g)$ for all $x \in A_i^p$. If that data structure does not answer "B" when $x = b_i$, it is obviously incorrect. On the other hand, if it answers "B" for both $x = b_i$ and some other $x' \in A_i^p$, it also makes an error. Since x' is plausible, there exist A' with $x' \in A'$ such that $f(A') = f(A)$. Then, we can run the data structure with A' as the first set and B as the second set. Since $f(A') = f(A)$, the data structure will behave exactly the same, and will incorrectly answer "B" for x'.

To draw our conclusion, we consider another encoding argument, this time in connection to the set B'. The Kolmogorov complexity of B' is $n \lg \frac{u}{n} - O(1)$. Consider a randomized encoding, depending on a set A drawn at random. First, we encode an n-bit vector specifying which indices i have $a_i = b_i$. It remains to encode $B' \setminus A = B$. We encode another n-bit vector, specifying for which positions i we have $b_i \in A_i^p$. For each $b_i \notin A_i^p$, we simply encode B_i using $\lceil \lg \frac{u}{n} \rceil$ bits. Finally, we include in the encoding $g(B, f(A))$. As explained already, this is enough to recover all b_i which are in A_i^p. Note that we do not need to encode $f(A)$, since this depends only on the random coins choosing the encoding and decoding algorithms.

The expected size of this encoding will be $O(n+S) + n \cdot \Pr_{A,B',i}[b_i \notin A_i^p] \cdot \lg \frac{u}{n}$. We know that with probability a half over A and i, we have $|A_i^p| \geq \frac{u/n}{2^{O(S/n)}}$. Thus, $\Pr_{A,B',i}[b_i \in A_i^p] \geq \frac{1}{2} \cdot 2^{-O(S/n)}$. Thus, the expected size of the encoding is at most $O(n+S) + (1 - 2^{-O(S/n)}) \cdot n \lg \frac{u}{n}$. Note that by the minimax principle, randomness in the encoding is unessential and we can always fix A guaranteeing the same encoding size, in expectation over B. We now get the bound:

$$
\begin{aligned}
& O(n+S) + (1 - 2^{-O(S/n)}) \cdot n \lg \frac{u}{n} \geq n \lg \frac{u}{n} - O(1) \\
\Rightarrow \quad & O\left(\frac{S}{n}\right) \geq 2^{-O(S/n)} \lg \frac{u}{n} - O(1) \\
\Rightarrow \quad & 2^{O(S/n)} O(S/n) \geq \lg \frac{u}{n} \Rightarrow \frac{S}{n} = \Omega\left(\lg \lg \frac{u}{n}\right)
\end{aligned}
$$

4. DYNAMIC RANGE REPORTING

4.1 Warm-up: the Greater-Than Function

We start with a simple upper bound of $t_u = O(\lg n)$, $t_q = O(\lg \lg n)$ for the greater-than problem. This gives a simpler context for understanding the recursion used on every path of the trie in the range reporting structure.

Our upper bound uses a trie structure. We consider a balanced tree with branching factor 2, and with n leaves. Every possible value of the update parameter a is represented by a root-to-leaf path. In the update stage, we mark this root-to-leaf path, taking time $O(\lg n)$. In the query stage, we want to find the point where b's path in the trie would diverge from a's path. This uses binary search on the $\lg n$ levels, as follows. To test if the paths diverge on a level, we examine the node on that level on b's path. If the node is marked, the paths diverge below; otherwise they diverge above. Once we have found the divergence point, we know that the larger of a and b is the one following the right child of the lowest common ancestor.

4.2 Description of the Data Structure

Let S be the current set of values stored by the data structure. Without loss of generality, assume w is a power of two. For an arbitrary $t \in [0, \lg w]$, we define the trie of order t, denoted T_t, to be the trie of depth $w/2^t$ and alphabet of 2^t *bits*, which represents all numbers in S. We call T_0 the *primary trie* (this is the classic binary trie with elements from S). Observe that we can assign distinct names of $O(w)$ bits to all nodes in all tries. We call *active paths* the paths in the tries which correspond to elements of S. A node v from T_t corresponds to a subtree of depth 2^t in the primary trie; we denote the root of this subtree by $r_0(v)$. A node from T_t corresponds to a 2-level subtree in T_{t-1}; we call such a subtree a *natural subtree*. Alternatively, a 2-level subtree of any trie is natural iff its root is at an even depth.

A root-to-leaf path in the primary trie is seen as the leaves of the tree used for the greater-than problem. The paths from the primary trie are broken into chunks of length 2^t in the trie of order t. So T_t is similar to the t-th level (counted bottom-up) of the greater-than tree. Indeed, every node on the t-th level of that tree held information about a subtree with 2^t leaves; here one edge in T_t summarizes a segment of length 2^t bits. Also, a natural subtree corresponds to two siblings in the greater-than structure. On the next level, the two siblings are contracted into a node; in the trie of higher order, a natural subtree is also contracted into a node. It will be very useful for the reader to hold these parallels in mind, and realize that the data structure from this section is implementing the old recursion idea *on every path*.

The root-to-leaf paths corresponding to the values in S determine at most $n-1$ branching nodes in any trie. By convention, we always consider roots to be branching nodes. For every branching node from T_0, we consider the extreme points of the interval spanned by the node's subtree. By doubling the universe size, we can assume these are never elements of S (alternatively, such extreme points are formal rationals like $x + \frac{1}{2}$). We define $\overline{S}$ to be the union of S and the two special values for each branching node in the primary trie; observe that $|\overline{S}| = O(n)$. We are interested in holding $\overline{S}$ for navigation purposes: it gives a way to find in constant time the maximum and minimum element from S that fits under a branching node (because these two values should be the elements from S closest to the special values for the branching node).

Our data structure has the following components:

1. a linked list with all elements of S in increasing order, and a predecessor structure for S.

2. a linked list with all elements of $\overline{S}$ in increasing order, accompanied by a navigation structure which enables us to find in constant time the largest value from S smaller than a given value from $\overline{S} \setminus S$. We also hold a predecessor structure for $\overline{S}$.

3. every branching node from the primary trie holds pointers to its lowest branching ancestor, and the two branching descendants (the highest branching nodes from the left and right subtrees; we consider leaves associated with elements from S as branching descendants). We also hold pointers to the two extreme values associated with the node in the list in item 2. Finally, we hold a hash table with these branching nodes.

4. for each t, and every node v in T_t, which is either a branching node or a child of a branching node on an active path, we hold the depth of the lowest branching ancestor of $r_0(v)$, using a Bloomier filter.

We begin by showing that this data structure takes linear space. Items 1-3 handle $O(n)$ elements, and have constant overhead per element. We show below that the navigation structure from 2. can be implemented in linear space. The predecessor structure should also use linear space; for van Emde Boas, this can be achieved through hashing [14].

In item 4., there are $O(n)$ branching nodes per trie. In addition, there are $O(n)$ children of branching nodes which are on active paths. Thus, we consider $O(n \lg w)$ nodes in total, and hold $O(\lg w)$ bits of information for each (a depth). Using our solution for the Bloomier filter, this takes $O(n(\lg w)^2 + w)$ bits, which is $o(n)$ words. Note that storing the depth of the branching ancestor is just a trick to reduce space. Once we have a node in T_0 and we know the depth of its branching ancestor, we can calculate the ancestor in $O(1)$ time (just ignore the bits below the depth of the ancestor). So in essence these are "compressed pointers" to the ancestors.

We now sketch the navigation structure from item 2. Observe that the longest run in the list of elements from $\overline{S} \setminus S$ can have length at most $2w$. Indeed, the leftmost and rightmost extreme values for the branching nodes form a parenthesis structure; the maximum depth is w, corresponding to the maximum depth in the trie. Between an open and a closed parenthesis, there must be at least one element from S, so the longest uninterrupted sequence of parenthesis can be w closed parenthesis and w open parenthesis.

The implementation of the navigation structure uses classic ideas. We bucket $\Theta(\sqrt{w})$ consecutive elements from the list, and then we bucket $\Theta(\sqrt{w})$ buckets. Each bucket holds a summary word, with a bit for each element indicating whether it is in S or not; second-order buckets hold bits saying whether first order buckets contain at least one element from S or not. There is also an array with pointers to the elements or first order buckets. By shifting, we can always insert another summary bit in constant time when something is added. However, we cannot insert something in the array in constant time; to fix that, we insert elements in the array on the next available position, and hold the correct permutation packed in a word (using $O(\sqrt{w} \lg w)$ bits). To find an element from S, we need to walk $O(1)$ buckets. The time is $O(1)$ per traversed bucket, since we can use the classic constant-time subroutine for finding the most significant bit [7].

We also describe a useful subroutine, TEST-BRANCHING(v), which tests whether a node v from some T_t is a branching node. To do that, we query the structure in item 4. to find the lowest branching ancestor of $r_0(v)$. This value is defined if v is a branching node, but the Bloomier filter may return an arbitrary result otherwise. We look up the purported ancestor in the structure of item 3. If the node is not a branching node, the value in the Bloom filter for v was bogus, so v is not a branching node. Otherwise, we inspect the two branching descendants of this node. If v is a branching node, one of these two descendants must be mapped to v in the trie of order t, which can be tested easily.

4.3 Implementation of Updates

This extended abstract only discusses insertions; deletions follow parallel steps uneventfully. We first insert the new element in S and $\overline{S}$ using the predecessor structures. Inserting a new element creates exactly one branching node v in the primary trie. This node can be determined by examining the predecessor and successor in S. Indeed, the lowest common ancestor in the primary trie can be determined by taking an xor of the two values, finding the most significant bit, and them masking everything below that bit from the original values [1].

We calculate the extreme values for the new branching node v, and insert them in $\overline{S}$ using the predecessor structure. Finding the branching ancestor of v is equivalent to finding the enclosing parentheses for the pair of parentheses which was just inserted. But $\overline{S}$ has a special structure: a pair of parentheses always encloses two subexpressions, which are either values from S, or a parenthesized expression (i.e., the branching nodes from T_0 form a binary tree structure). So one of the enclosing parentheses is either immediately to the left, or immediately to the right of the new pair. We can traverse a link from there to find the branching ancestor. Once we have this ancestor, it is easy to update the local structure of the branching nodes from item 3. Until now, the time is dominated by the predecessor structure.

It remains to update the structure in item 4. For each $t > 0$, we can either create a new branching node in T_t, or the branching node existed already (this is possible for $t > 0$ because nodes have many children). We first test whether the branching node existed or not (using the TEST-BRANCHING subroutine). If we need to introduce a branching node, we simply add a new new entry in the Bloomier filter with the depth of the branching ancestor of v. It remains to consider active children of branching nodes, for which we must store the depth of v. If we have just introduced a branching node, it has exactly two active children (if there exist more than two children on active paths, the node was a branching node before). These children are determined by looking at the branching descendants of v; these give the two active paths going into v. Both descendants are mapped to active children of the new branching node from T_t. If the branching node already existed, we must add one active child, which is simply the child that the path to the newly inserted value follows. Thus, to update item 4., we spend constant time per T_t. In total, the running time of an update is $T_{pred} + O(\lg w) = O(\lg w)$.

4.4 Implementation of Queries

Remember that a query receives an interval $[a, b]$ and must return a value in $S \cap [a, b]$, if one exists. We begin by finding the node v which is the lowest common ancestor of a and b in the primary trie; this takes constant time [1]. Note that v spans an interval which includes $[a, b]$. The easiest case is when v is a branching node; this can be recognized by a lookup in the hash table from item 3. If so, we find the two branching descendants of v; call the left one v_L and the right one v_R. Then, if $S \cap [a, b] \neq \emptyset$, either the rightmost value from S that fits under v_L or the leftmost value from S that in fits under v_R must be in the interval $[a, b]$. This is so because $[a, b]$ straddles the middle point of the interval spanned by v. The two values mentioned above are the two values from S closest (on both sides) to this middle point, so if $[a, b]$ is non-empty, it must contain one of these two. To find these two values, we follow a pointer from v_L to its left extreme point in $\overline{S}$. Then, we use the navigation structure from item 2., and find the predecessor from S of this value in constant time. The rightmost value under v_R is the next element from S. Altogether, the case when v is a branching node takes constant time.

Now we must handle the case when v is not a branching node. If $S \cap [a, b] \neq \emptyset$, it must be the case that v is on an active path. Below we describe how to find the lowest branching ancestor of v, *assuming that v is on an active path*. If this assumption is violated, the value returned can be arbitrary. Once we have the branching ancestor of v, we find the branching descendant w which is in v's subtree. Now it is easy to see, by the same reasoning as above, that if $[a, b] \cap S \neq \emptyset$ either the leftmost or the rightmost value from S which is under w must be in $[a, b]$. These two values are found in constant time using the navigation structure from item 2., as described above. So if $[a, b] \cap S \neq \emptyset$, we can find an element inside $[a, b]$. If none of these two elements were in $[a, b]$ it must be the case that $[a, b]$ was empty, because the algorithm works correctly when $[a, b] \cap S \neq \emptyset$.

It remains to show how to find v's branching ancestor, assuming v is on an active path, but is not a branching node. If for some $t > 0$, v is mapped to a branching node in T_t, it will also be mapped to a branching node in tries of higher order. We are interested in the smallest t for which this happens. We find this t by binary search, taking time $O(\lg \lg w)$. For some proposed t, we check whether the node to which v is mapped in T_t is a branching node (using the TEST-BRANCHING subroutine). If it is, we continue searching below; otherwise, we continue above.

Suppose we found the smallest t for which v is mapped to a branching node. In T_{t-1}, v is mapped to some z which is *not* a branching node. Finding the lowest branching ancestor of v is identical to finding the lowest branching ancestor of $r_0(z)$ in the primary trie (since z is a not a branching node, there is no branching node in the primary trie in the subtree corresponding to z). Since in T_t z gets mapped to a branching node, its natural subtree in T_{t-1} must contain at least one branching node. We have two cases: either z is the root or a leaf of the natural subtree (remember that a natural subtree has two levels). These can be distinguished based on the parity of z's depth. If z is a leaf, the root must be a branching node (because there is at least another active leaf). But then z is an active child of a branching node, so item 4. tells us the branching ancestor of $r_0(z)$. Now consider the case when z is the root of the natural subtree. Then z is above any branching node in its natural subtree, so to find the branching ancestor of $r_0(z)$ we can find the branching ancestor of the node from T_t to which the natural subtree is mapped. But this is a branching node, so the structure in item 4. gives the desired branching ancestor. To summarize, the only super-constant cost is the binary search for t, which takes $O(\lg \lg w)$ time.

Acknowledgement. The authors would like to thank Gerth Brodal for discussions in the early stages of this work, in particular on how the results could be extended to dynamic range counting.

5. REFERENCES

[1] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[2] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. See also STOC'99.

[3] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proc. 15th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–39, 2004.

[4] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 569–580, 1996.

[5] M. Dietzfelbinger and F. M. auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc 17th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 6–19, 1990.

[6] M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, 1982.

[7] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90.

[8] T. Hagerup. Sorting and searching on the word RAM. In *Proc. 15th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 366–398, 1998.

[9] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998. See also STOC'95.

[10] A. Pagh, R. Pagh, and S. S. Rao. An optimal Bloom filter replacement. In *Proc. 16th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2005. To appear.

[11] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995. See also SODA'93.

[12] A. Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004.

[13] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. See also FOCS'75.

[14] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.