Improved data structures for predecessor queries in integer sets

Rajeev Raman*

August 1995 (revised March 1996)

Abstract

We consider the problem of maintaining a dynamic ordered set of n integers in the range $0\,..\,2^w-1$, under the operations of insertion, deletion and predecessor queries, on a unit-cost RAM with a word length of w bits. We show that all the operations above can be performed in $O(\min\{\log w, 1+\log n/\log w\})$ expected time, assuming the updates are *oblivious*, i.e., independent of the random choices made by the data structure. This improves upon the (deterministic) running time of $O(\min\{\log w, \sqrt{\log n}\})$ obtained by Fredman and Willard. We also give a very simple deterministic data structure which matches the bound of Fredman and Willard. Finally, from the randomized data structure we are able to derive improved deterministic data structures for the *static* version of this problem.

1 Introduction

The *dynamic predecessor* problem is that of maintaining a subset S of an ordered universe U under the operations of insertion, deletion and the *predecessor query* operation, which, given some $x \in U$ as input, returns $pred(x,S) = \max\{y \in S \mid y \leq x\}$. In the comparison-based setting, where we may obtain information about the relative order of two elements of U only through pairwise comparisons, it has long been known that all three operations above can be performed in $O(\log n)$ time, where $n = |S|$ [1]. However, additional knowledge about the universe U can lead to more efficient solutions than would be possible in the comparison-based framework: for instance, if the universe U consists of character strings then a data structure such as a trie, which is not a comparison-based data structure, is likely to outperform a comparison-based one.

Another important case is when the universe U consists of integers in a certain range. Dynamic predecessor data structures customized for this case were first proposed by van Emde Boas [10] and have found many applications in the literature. Furthermore, it turns out that a data structure for integer sets on existing machines can also often handle floating-point numbers without modification: many internal representations of floating-point numbers have the property that relative order of two floating-point numbers is the same as the relative order of the bit sequences representing them, considered as integers (the IEEE 754 floating-point standard was specifically designed to allow floating-point numbers and integers to be compared by the same hardware).

*Algorithm Design Group, Department of Computer Science,King's College London, Strand, London WC2R 2LS, U. K,raman@dcs.kcl.ac.uk

The machine model we use in this paper is the unit-cost RAM with a word length of w bits. Accordingly, the universe U consists of the integers in the range $0\,..\,2^w - 1$. It is assumed that the RAM can perform addition, subtraction, bitwise logical operations, comparison, unrestricted bit shifts and multiplication in constant time on $O(w)$-bit operands. Furthermore, we postulate access to a constant-time generator of a random integer uniformly distributed over $\{0, .., 2^w - 1\}$.

The priority queues of van Emde Boas [10, 11] perform all the required operations in $O(\log w)$ time. Fredman and Willard [13] gave a linear-space data structure that performs all operations in $O(\log n / \log\log n)$ time, and a variant of their data structure performs all operations in $O(\sqrt{\log n})$ time when $w = 2^{\Omega(\sqrt{\log n})}$. The best previous time bound for the dynamic predecessor problem was therefore $O(\min\{\log w, \sqrt{\log n}\})$.

We give two new data structures, both of which use techniques developed in [4]. The first is extremely simple and performs all operations in $O(\min\{\log w, \sqrt{\log n}\})$ time. The second is randomized, and assuming that updates are independent of the random choices made by the data structure, processes all operations in $O(1 + \log n / \log w)$ expected time, which is faster than known methods when $w = 2^{\omega(\sqrt{\log n})}$. The dynamic predecessor problem can therefore be solved in $O(\min\{\log w, 1+\log n / \log w\})$ expected time, and in particular, when $n = w^{O(1)}$, we can process queries and oblivious updates in constant expected time.

A data structure for the predecessor problem on sets of maximum size $w^{O(1)}$ is referred to as a "small priority queue". Some of the many applications of small priority queues can be found in [7, 6, 13, 14, 17]. Ajtai, Fredman and Komlos [2] gave a dynamic small priority queue that performed all operations in $O(1)$ time on the the powerful cell-probe model, but implementing it on a RAM would require non-standard unit-time instructions. Fredman and Willard [13] gave a static priority queue that processed queries in $O(1)$ time using standard RAM instructions, after polynomial-time preprocessing, which also required polynomial space. Another small priority queue given by the same authors in [14] allows constant-time updates, but requires pre-computed tables of size exponential in M, the maximum size of the priority queue. This normally restricts the value of M to $(\log n)^{O(1)}$, where n is the input size, in order to keep the cost of pre-computing reasonable.

Our result therefore gives a small priority queue that uses standard RAM instructions yet processes queries and oblivious updates in $O(1)$ expected time, without the size limitations of [14]. Two new applications of this small priority queue are as follows:

Priority Queues with Decrease-Key: plugging this result into the AF-heap data structure of Fredman and Willard [14] we get a data structure which supports insert and decrease-key in constant expected time, while delete and delete-min take $O(1 + \log n / \log w)$ expected time, which is faster than the data structure of [14] when $w = (\log n)^{\omega(1)}$. A priority queue recently described by Thorup [17] has a faster running time, but requires that operations on the priority queue be *monotone*, i.e., the minimum value in the data structure should be a non-decreasing function of time. Thorup's result suffices, however, for some important applications such as computing single-source shortest paths.

Monotone Priority Queues: We can use our small priority queue in place of that of [14] in Thorup's reduction from monotone priority queues to sorting [17, Section 3.1]. This simplifies the data structure as one component [17, Section 3.1, component (iii)] is no longer needed. Although randomization is thereby introduced into the reduction, this does not affect the main application of the reduction which is anyway a randomized result.

Furthermore, by de-randomizing the above data structure we obtain a small priority queue which can be preprocessed in time polynomial in n to answer queries in $O(1)$ time. This data structure uses $n+2$ locations to store a set of size $n \leq w^{1/3}$ and hence $n(1+O(n^{-1/3}))$ space when $n = w^{O(1)}$, in contrast to the small priority queue of Fredman and Willard which uses space that is polynomial in n. As a corollary we obtain that n keys can be preprocessed in $O(n^{1+\epsilon})$ time for any fixed $\epsilon > 0$ to obtain a static linear-space data structure with $O(\min\{\log w, 1+\log n/\log w\})$ query time.
REMARK: A. Andersson [3] has shown that a running time of $O(\min\{\log w, \sqrt{\log n}\})$ can be achieved by a conceptually simple algorithm even when unit-time multiplication is not part of the instruction set. Theorem 1 is not, therefore, of interest *per se*. However, as the main components of Theorem 1 are either of independent interest or else are used elsewhere in our paper, we have chosen to present this result.

2 Overview

Let Σ^* denote the set of all strings over some finite alphabet Σ, and let Σ^k denote the set of all strings over Σ of length k, for any integer $k \geq 0$. Let $S \subset \Sigma^*$ be some finite set. For $x, y \in \Sigma^*$ let $lcp(x,y)$ denote the longest common prefix of x and y, and let $pre(S) = \{lcp(x,y) \mid x,y \in S, x \neq y\} \cup \{\Lambda\}$, where Λ denotes the empty string. The *compressed trie* for S is obtained from the usual trie (or digital search tree) by eliminating all nodes of degree 1 (see e.g. [15]). Each external node (leaf) of the compressed trie corresponds to a member of S, while each internal node can be identified with a member of $pre(S)$. Furthermore, each member of $pre(S)$ is determined by at least one pair of elements of consecutive rank in S (except possibly Λ). Our data structures for solving the predecessor problem all have the same structure. For a given parameter k, we view each integer key in S of w bits as a character string of length k over the alphabet $\{0, \ldots, 2^{w/k}-1\}$, and maintain three logically distinct data structures (which need not be physically distinct) which are:
(1) A *prefix* data structure which, given the query key x, quickly computes:

$$fail(x,S) = \max\{y \in pre(S) \mid y \text{ a prefix of } x\}$$

(this is the member of $pre(S)$ corresponding to the last internal node of the compressed trie for S visited during a search for x).
(2) A *compressed trie* on S, with all the standard information associated with the nodes and edges of a compressed trie—each internal or leaf node of the trie has a pointer to the smallest and largest elements stored under it and each edge going out from an internal node has associated with it a label, which is a single character, and the "length" of the edge. Furthermore, all leaves are linked together in a doubly-linked list.
(3) A *siblings list* data structure, which stores with with each internal node of the trie the information associated with each outgoing edge in a data structure ordered by the one-character label of the edge.

In order to do a search, the prefix data structure is queried, and $fail(x,S)$ is computed; using this information we then go to the trie internal node associated with $fail(x,S)$ and locate the appropriate character of x with respect to the labels of the outgoing edges of this node in the siblings list. Using the maximum and minimum pointers, it is now easy to compute one of $pred(x,S)$ or $succ(x,S)$; using the linked list we can compute the other one.

In order to do an insertion, we proceed as above to compute $fail(x, S)$ and $pred(x, S)$. Computing $pred(x, S)$ allows us to insert into the linked list (presumably in constant time). The rest of the insertion depends upon whether or not the character of x that was located in the siblings list of $fail(x, S)$ was already present as a label of an outgoing edge; if yes, then an edge of the trie must be "split", introducing a new element into $pre(S)$; if not, then the siblings list of $fail(x, S)$ needs to be updated and $pre(S)$ is unchanged. The minimum pointers in the compressed trie need to be updated as follows: for every node that is a common ancestor of x and $succ(x, S)$, wherever $succ(x, S)$ is pointed to by the minimum pointer, it is replaced by a pointer to x, and similarly for the maximum pointers.

Note that the computation of $fail(x, S)$ reduces the problem of locating x (a w-bit key) in S to the problem of locating a character of x (a w/k-bit key) among a set of characters. This constitutes a "range reduction" for searching. In the context of sorting [4] combined range reductions for sorting with "packed sorting" algorithms, which work by packing several reduced keys into a word and thus operating upon many keys in constant time. In Section 4 we introduce the notion of a *packed B-tree* which is the counterpart of packed sorting. In Section 5 we combine this with a simple deterministic range reduction. In Section 6 the randomized data structure is described; Section 7 contains remarks regarding de-randomization.

3 Preliminaries

In this section we will introduce some notation to facilitate the expression of bit-level operations, based on the notation developed in [4]. The (M, f)-*representation* partitions the rightmost Mf bits of a word into M *fields* of f bits each, while ignoring any other bits present in the word. The fields are numbered $1, \ldots, M$ from right to left, and the leftmost bit of each field, called the *test bit*, is required to be zero. The contents of these fields will be variously interpreted as integers, boolean values (if all field values are in $\{0, 1\}$) or records with a number of named *components*. Hence a word can represent a sequence of integers, booleans, or records. A word in (M, f) representation with $m < M$ non-empty fields is said to be *compact* if the non-empty fields are those numbered $1, \ldots, m$.

The built-in bitwise boolean operations will be denoted by AND and OR, respectively, and the shift operator is rendered as $\uparrow$ or $\downarrow$: When x and i are integers, $x \uparrow i$ denotes $\lfloor x \cdot 2^i \rfloor$, and $x \downarrow i = x \uparrow (-i)$. We denote the multiplication of integers x and y by either xy or $x \cdot y$, using the latter only when the desired effect cannot otherwise be obtained in $O(1)$ time (using shifts, for example).

We now describe a set of constant-time operations, many of which operate on sequences of integers, booleans or records on a componentwise basis, whose implementation can be found in [4]. In the following, assume the (M, f)-representation used throughout, for integers $M, f \geq 2$. The constant $1_{M,f} = \sum_{i=0}^{M-1} 2^{if}$ is of fundamental importance and can be calculated in $O(\log M)$ time given M and f.

For operating on boolean sequences, the operations for componentwise logical conjunction, disjunction and negation are denoted by $\wedge$, $\vee$ and $\neg$ respectively. Let OP denote any relational operator from the set $\{<, \leq, =, \neq, \geq, >\}$. Given two integer sequences $X = (x_1, \ldots, x_M)$ and $Y = (y_1, \ldots, y_M)$, the operation $[X \text{ OP } Y]$ returns the boolean sequence $(b_1, \ldots, b_M)$ with $b_i = true$ if and only if x_i OP y_i, for $i = 1, \ldots, M$. For sequences of records we extend the above notation to compare records based on a particular component

with name *name*, denoted by $[X.name \text{ OP } Y.name]$. Finally, given a single record or integer z, we use the notation $[X \text{ OP } z]$ or $[X.name \text{ OP } z.name]$ to simultaneously compare each element of X with z; this is implemented by obtaining the sequence $(z, z, \ldots, z)$ as $z \cdot 1_{M,f}$.

Another useful operator is the *extract* operator $|$. When $X = (x_1, \ldots, x_M)$ is a sequence of integers or records and $B = (b_1, \ldots, b_M)$ is a boolean sequence, $X \mid B$ denotes the sequence of integers or records $(y_1, \ldots, y_M)$ such that for $i = 1, \ldots, M$, $y_i = x_i$ if $b_i = true$ and $y_i = 0$ otherwise. Finally, given M, f such that $f \geq \log M + 2$ and the quantity $1_{M,f}$, we can compute $leftmost(X)$, which is the index of the leftmost true, non-zero or non-empty field in X, when X is respectively a sequence of booleans, integers or records, in constant time [4, Lemma 1] (see also [13]).

This enables us to do "associative lookup" in a sequence of records, where one of the records satisfying some (simple) condition on some component is extracted from the sequence and returned (in the first field of an output word). For instance, given a sequence of records X we can extract a record whose *name* component equals *value* by computing $Y := X \mid [X.name = value]$ and returning $Y \downarrow ((leftmost(Y) - 1)f)$.

4 Packed B-trees

Given a set of n keys of w/k bits each, we show how to solve the dynamic predecessor problem in $O(\log n / \log k)$ time. The main data structure we use is very simple and is called a *packed priority queue*. It maintains a set S of at most M records, ordered with respect to some key, under updates and *rank* and *index* queries, where the rank query computes the rank of a search key in the set of all keys in S, and the index query takes an integer and returns a record whose key has a rank equal to that integer. We assume each record fits in a field of f bits with the leftmost bit of the field equal to zero.

Lemma 1 *Provided that $Mf \leq w$ and $f \geq \log M + 2$ we can perform all the above operations in constant time.*

PROOF. A "data" word D stores the records of S in the compact (M, f) representation and a "rank" word R stores in its ith field the rank of the key of the ith record in the set of keys of S, for $i = 1, \ldots, |S|$. In order to process a rank query for an input record x we compute $A := [D.key \leq x.key]$ and compute the rank $r := (A \cdot 1_{M,f}) \downarrow (M-1)f$; essentially as described in [13]. An index query is done by computing $i := leftmost([R = r])$ and returning $(D \mid [R = r]) \downarrow (i-1)f$ if $i > 0$, and zero otherwise. In order to do an insertion, we place the new record x and its rank into fields $|S|+1$ of D and R, respectively, and update R as $R + [D.key > x.key]$. □

A *packed B-tree* is obtained by taking a B-tree with large branching factor and storing the keys at each internal node using Lemma 1. Specifically, the branching factor b is maintained at $\Theta(\sqrt{k})$, pointers to the children of an internal node are stored in an array, and the packed priority queue stores records consisting of (key, array index) pairs, such that if the predecessor of the search key is the first component of a pair, the second component gives the child pointer to be followed. These pairs are stored in a packed priority queue with $M = \sqrt{k}$, with each field consisting of $f = w/k + \log M + 2$ bits; for k sufficiently large this implies $Mf \leq w$. Note that an internal node can be split in $O(b)$ time, implying that insertions take $O(1)$ amortized time given the position of the key to be inserted. We summarize:

Lemma 2 *The dynamic predecessor problem on a set of n keys, each of w/k bits, can be solved in $O(\log n/\log k)$ time for all operations.*

5 A simple deterministic data structure

We now give a simple data structure that solves the dynamic predecessor problem in $O(\min\{\log w, \sqrt{\log n}\})$ time. The constituent parts of this data structure are as follows.

The prefix data structure is essentially just the *uncompressed* trie for $pre(S)$ (which can be viewed as a generalization of the "silly" version of the van Emde Boas data structure [11], or of Willard's Y-fast tries [18]). Each node in the trie that is a member of $pre(S)$ is considered "marked". The task of computing $fail(x, S)$ is simply the task of finding the first marked node on the path from the leaf corresponding to x to the root. This trie is implemented by an array indexed by all possible strings over the alphabet of length at most k. Each array entry has both a mark bit and a counter, initialized respectively to false and zero, but in general storing whether or not the node is marked, and the count of marked nodes below this node, including itself. Note that $pre(S)$ is closed under the *lcp* operation: this means that the lowest common ancestor of two marked nodes is also a marked node.

To compute $fail(x, S)$ we do a binary search along the path to the root, to find the first node whose counter value is greater than zero. If this node is marked, then all is well, otherwise a binary search is performed from this node up to the root to find the deepest node with the a larger counter value than this node, which is $fail(x, S)$. This takes $O(\log k)$ time.

Insertions and deletions into $pre(S)$ are trivial: for each prefix of the string to be inserted (deleted), increment (decrement) the counter of the corresponding array entry, taking $O(k)$ time, and marking/unmarking the last node, as appropriate.

The compressed trie for S is stored in the usual manner (see [15]), furthermore, the leaves of the data structure (corresponding to the members of S) are linked in a doubly linked list. Updates to the trie can be performed in $O(k)$ time using standard algorithms.

The value of k is set to $\min\{w, 2^{\sqrt{\log n}}\}$. If $k = w$ the siblings lists are trivial and queries take $O(1)$ time, otherwise they are are represented using Lemma 2, with all operations taking $O(\sqrt{\log n})$ time. The query time of this data structure is $O(\log k) = O(\min\{\log w, \sqrt{\log n}\})$, but the update time is $O(k)$: to improve this to $O(\log k)$ without affecting the query time, the set S is stored in buckets of size $\Theta(k)$ represented as conventional balanced search trees.

Theorem 1 *The dynamic predecessor problem on n keys of w bits each can be solved in $O(\min\{\log w, \sqrt{\log n}\})$ time on a unit-cost RAM with w bits.*

6 The randomized dynamic data structure

The randomized data structure follows the general outline given before, but adapts ideas first developed in [4] to compute $fail(x, S)$ quickly. As before, we consider each key as a string of length k over the alphabet $\Sigma = \{0, \ldots, 2^{w/k} - 1\}$, for some parameter k to be chosen later. Given a *signature* function $g : \Sigma \to \{0, \ldots, 2^l - 1\}$, for some $l \leq w/k$, and any string $y = (y_1, \ldots, y_k)$ we denote by $g(y)$ the word which contains the values $g(y_1), \ldots, g(y_k)$ in consecutive l-bit fields, and call $g(y)$ the *concatenated signature* of y (the concatenated signature of integers representing strings of length $< k$ is defined analogously). Also, for

any $T \subseteq \Sigma^k$, let $g(T)$ denote $\cup_{y \in T} g(y)$. The signature function should ideally have the property that it is 1-1 on all the characters in the keys currently in the set S, but should have a small range, i.e., l should be small compared to w/k.

If g is 1-1 on all the characters of S, then for any $y \in pre(S)$, $g(y) \in pre(g(S))$, and each member of $pre(g(S))$ is the image under g of some element of $pre(S)$. Also, if g happens to be 1-1 on the characters in $S \cup \{x\}$ then $fail(g(x), g(S))$ is simply the image of $fail(x, S)$ under g, and it suffices to compute the former. The concatenated signatures of the keys in $S \cup \{x\}$ are easier to work with, being much shorter than the keys themselves.

However, we would like to compute $fail(x, S)$ when g is known only to be 1-1 on the characters in S. In this case, we compute $y = fail(g(x), g(S))$ as before, and thus find $y' \in pre(S)$ such that $g(y') = y$. If y' is a prefix of the query key x, it follows that $y' = fail(x, S)$. Otherwise, if y' and x have a common prefix of length $i < |y'|$, then the $i+1$-st character of x must be the first character of x which is not in the set of characters currently in S. Let g' be obtained by replacing the $i+1$-st character of $g(x)$ by a value which is known not to be the image under g of any character in S. We then compute $z = fail(g', g(S))$ and determine $z' \in pre(S)$ such that $g(z') = z$; it can easily be checked that z' is indeed the desired value of $fail(x, S)$.

In order to compute $fail(x, S)$ in constant time, we therefore need to be able to compute $fail(y, g(S))$ for a (possibly modified) concatenated signature y, to maintain the mapping between $pre(S)$ and $pre(g(S))$ and to be able to compute $g(x)$ from x, all in constant time.

In [4] it is shown that if g is chosen from a universal class of hash functions defined by Dietzfelbinger et al. [9], then we can compute the concatenated signature of a key in constant time (for the values of k and l in which we are interested, at any rate). Furthermore, a randomly chosen function g from this class will be 1-1 on the characters in S with probability probability $1-(kn)^2/2^l$, where $n = |S|$. This probability can be made of the form $1-(kn)^{-c}$ for any constant $c > 0$ by choosing $l = \Theta(\log(kn))$. We will actually consider the signatures of characters under g as being $f = l+2$ bits long, with the two most significant bits always being equal to zero. (The most significant bit will be used as the test bit described in Section 3. Furthermore, this allows us to choose the value 2^{l+1} as being a value that is never the image under g of any "real" character.)

We now describe the remaining parts of the priority queues.

6.1 Efficient small priority queues

Recall that we are interested in maintaining dynamic sets of w-bit integers of maximum size w^c for some constant $c > 0$. We will for the rest of this section fix the maximum size M of a small priority queue at $w^{1/3}$ and the value of k at $\sqrt{w/\log w}$. This means that $l = \Theta(\log w)$ suffices to get a function g which is 1-1 with high probability.

6.1.1 Constituent parts

The small priority queue has an array of size M which holds the keys; we will index these array elements with indices from $1..M$. It also has the constituent parts outlined earlier, each of which is described in turn.

(1) The prefix data structure comprises of two parts. The first is a word containing the members of $pre(g(S))$, ordered by their length (ties between prefixes of equal length are broken arbitrarily). The set $pre(g(S))$ is represented in the compact $(M, 2kf)$-representation.

Each field of $2kf$ bits in this word which stores a member of $pre(g(S))$ is further partitioned into $2k$ sub-fields of f bits each, and the string stored in each field is stored right-justified with each subfield of f bits containing a single character, i.e., each string is represented essentially in the compact $(2k, f)$-representation. The total number of bits required is $O(Mkf) = O(w^{5/6} \log w) = o(w)$.

The second is an "associative lookup" table storing pairs of the form (x, i) where $x \in pre(g(S))$ and i is a unique integer label in the range $(M+1)\,..\,2M$. A new label is given to each member of $pre(g(S))$ as it is inserted into this data structure and is not changed for the lifetime of this data structure. These labels uniquely name each internal node of the trie, and have been chosen disjoint from the range $1\,..\,M$, as external nodes will be labelled with the array indices where the corresponding keys are stored. Each element of $pre(g(S))$ has $O(\sqrt{w \log w})$ bits and the labels are $O(\log M)$ bits long, so the associative table also fits into a single word, and is stored in a compact representation with an appropriate field width.

(2) We maintain a doubly linked list of the keys in sorted order in a single word (to conserve space) in compact representation. There are up to M fields each containing a pair $(prev, next)$ of array indices, the ith field containing the locations of the predecessor and successor in S of the key stored at the ith array location, for $i = 1, \ldots, M$ (0 denoting a null pointer). The pointers are stored in a single word, requiring $O(M \log M) = O(w^{1/3} \log w)$ bits.

(3) The compressed trie and siblings list are combined together in the following representation of the trie. The maximum and minimum pointers associated with each internal or external node v are stored as records consisting of triples of labels of the form (*node, max, min*), with the obvious interpretation. The maximum and minimum pointers of external nodes point to themselves. These triples are stored in the compact $(2M, f)$-representation with $f = O(\log M)$ (the compressed trie has at most $2M - 1$ nodes). Each triple is stored in each of two words. In the first word, called ${}^{+}P$, the order of vertices defines a pre-order, while in the second, called P^{+}, a post-order. As is well known, a node v is an ancestor of a node w iff v precedes w in a pre-order and succeeds w in a post-order.

Finally, a trie edge $(u \to v)$ is represented as a quadruple of the form (u, c, l, v), where c is the character label of the edge and l its length. Since there are at most $2M$ edges, we store these records in a packed priority queue, using the $(2M, f)$-representation, with a field width of $f = O(\log m + \log w + w/k) = O(\sqrt{w \log w})$ bits. The key for performing comparisons is the ordered pair (u, c). As $2Mf = O(w^{5/6} \log n)$ and $f > \log M + 2$ for sufficiently large w, packed priority queue operations take constant time by Lemma 1.

6.1.2 Queries

The computation of $fail(g(x), g(S))$ is essentially brute-force. Firstly, note that if two strings are stored in compact representation in two different words X and Y, we can determine if the string in X is a prefix of the string in Y by checking $leftmost([X \neq Y])$ against $leftmost(X)$. In fact, it turns out that given the string $g(x)$ we can simultaneously determine for each element of $pre(g(S))$, whether or not it is a prefix of $g(x)$ in constant time, by in essence performing all the required *leftmost* computations simultaneously. This follows from a more detailed look at the implementation of the *leftmost* operation given in [4, Lemma 1]. The salient points are that (a) the operations performed to compute *leftmost* are data-

independent (b) the intermediate results obtained during the computation are small enough that by storing strings of length kf in a field of size $2kf$, we can ensure that the separate computations do not interfere. Using the *leftmost* operation once again, we determine the leftmost, and hence longest, element of $g(S)$ which is also a prefix of $g(x)$, thus computing $\mathit{fail}(g(x), g(S))$. We then obtain the label of the trie internal node corresponding to the pre-image of $\mathit{fail}(g(x), g(S))$ in $\mathit{pre}(S)$. In order to find the prefix itself we simply compute the longest common prefix of the maximum and minimum nodes pointed to by this internal node. The rest of the computation of $\mathit{fail}(x, S)$ is done as indicated in the outline above.

We now move to the trie, where we locate the pair (u, c), where c is the appropriate distinguishing character of x, among the set of trie edge records, again in constant time. Now, if necessary, we can access the maximum and minimum pointers either of u or of the appropriate child of u to determine the predecessor, extracting the required values by associative lookup as described previously. This information is enough to compute the predecessor or the successor of x, using the linked list we compute the other.

6.1.3 Updates

For the update, we assume that g is $1-1$ on all keys in $S \cup \{x\}$ (this is at least plausible in light of the assumption of oblivious updates). Following the same method as for a query, we compute $\mathit{fail}(x, S)$, and as outlined above, determine if a new member needs to be added to $\mathit{pre}(S)$ as a result of this update. If so we generate a new label u' for this new member, and store the new pair in the associative lookup table. We now have to update the word containing $g(\mathit{pre}(S))$ ordered by length—in order to do this, we note that it is possible to simultaneously compute the lengths of each member of $g(S)$ in $O(1)$ time (the idea is essentially the same as was used to compute $\mathit{fail}(g(x), g(S))$). We now count the number of prefixes which are shorter than the new prefix to be inserted, after which it is easy to insert the new member of $g(S)$ in its appropriate place in the word. Updating the siblings list and the linked list is trivial.

We now address the problem of maintaining the words ${}^{+}P$ and P^{+} and the maximum/minimum pointers in them. If there is a new internal node u' to be added to the trie, it will be be a child of u. The new external node v will be a child u' if it is present, and of u otherwise. We first make a record containing $(u', max(u'), min(u'))$. Since u' inherits one of the minimum and maximum values from the endpoint of the edge that was split to introduce u, and since the other will be the new external node, this is easy. We also make a record (v, v, v) for the external node v.

First, the record for u' is added right before the record for u in P^{+} (by induction, the nodes appear in post-order in P^{+}). This is done by computing the index i of the field containing u as $i := \mathit{leftmost}([P^{+}.name = u])$ and let $F := 1_{M,f} \downarrow ((M - i + 1)f)$. By setting $P^{+} := (P^{+} \mid \neg F) \uparrow f + (P^{+} \mid F)$ we have moved the record of u and all those to its left to fields with index one greater than before and have vacated field i; the new record is then inserted in the newly vacant field. A similar method is used to add v into P^{+} and to add u' and v after u in ${}^{+}P$.

We also have to update the remaining maximum and minimum pointers, for which we define a new operation. Given integer sequences $X = (x_1, \ldots, x_M)$ and $Y = (y_1, \ldots, y_M)$, the operation $[X \in Y]$ returns the boolean sequence $b = (b_1, \ldots, b_M)$ where $b_i = \mathit{true}$ iff $x_i \in Y$ for $i = 1, \ldots, M$, where Y is considered a set. Note that $X \mid [X \in Y]$ selects those elements of X that belong to $X \cap Y$. When X and Y are sequences of records with

a component named *name*, the notation $[X.name \in Y.name]$ has the obvious meaning.

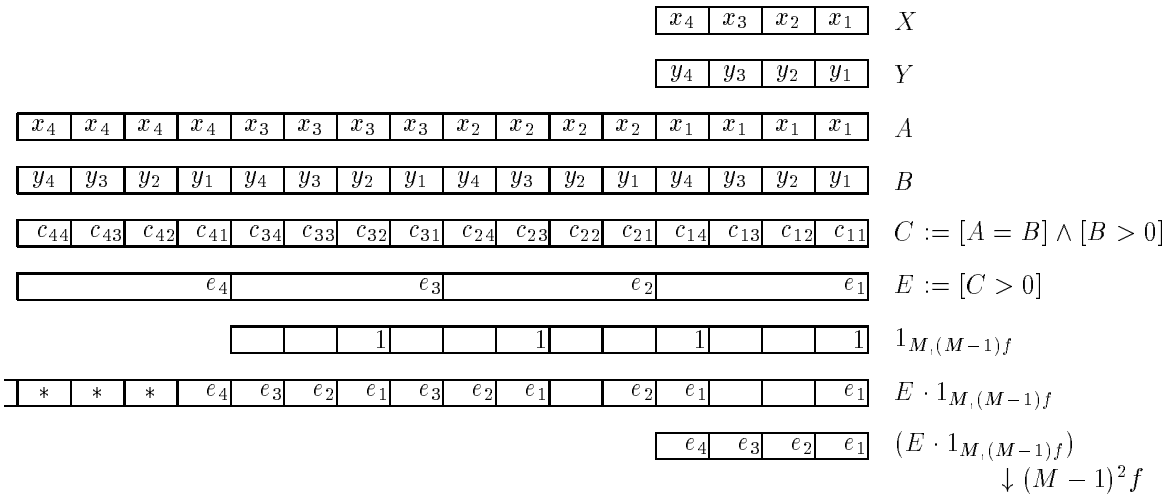

Figure 1: Computing $[X \in Y]$ ($*$ denotes a don't care value).

Lemma 3 *Suppose that we are given M, f such that $f \geq \log M + 2$ and integer sequences X, Y in the (M, f)-representation, and the constants $1_{M,f}$, $1_{M,(M-1)f}$ and $1_{(M-1)^2 f}$. Then in constant time, using a word length of $M^2 f$ bits, we can compute $[X \in Y]$.*

Proof. The algorithm is depicted in Figure 1. Temporarily using the (M^2, f)-representation, we create words A and B as shown above from X and Y respectively, in constant time [4, Lemma 3]. Now we compute $C := [A = B] \wedge [B > 0]$, resulting in the value c_{ij} being stored in field $(i-1)M + j$ of C, where $c_{ij} = true$ iff $x_i = y_j$ and both fields are nonempty, for $i, j = 1, \ldots, M$. As $x_i \in Y$ iff $\vee_{j=1}^{M} c_{ij}$, for $i = 1, \ldots, M$, we compute these quantities by viewing C as an integer sequence in the (M, Mf)-representation; the instruction $E := [C > 0]$ in the (M, Mf)-representation then achieves this. Finally we move these into consecutive locations by a multiplication followed by a shift. □

Now the minimum pointers are updated as follows. Let v' be the label of the successor of v in S; if there is none, no minimum pointers need to be updated. Otherwise, every common ancestor of v and v' that points to v' has to be modified to point to v. Let i the index of the field containing whichever of v and v' occurs earlier in ${}^{+}P$. We extract all fields indexed $i-1$ or less by setting ${}^{+}P_1 := {}^{+}P \mid (1_{M,f} \downarrow ((M-i+1)f))$. Similarly we extract P_1^{+} containing all the records that occur after both v and v' in P^{+}. Computing $F := [{}^{+}P_1 \in P_1^{+}]$ gives a mask which can be used to select those records in ${}^{+}P$ that correspond to common ancestors of v and v' and executing $F' := [({}^{+}P \mid F).min = v']$ gives a mask which can be used to extract exactly those records from ${}^{+}P$ that have to be updated (and to perform the updates as well). The updating of the minimum field values in P^{+}, as well as the maximum field values in both P^{+} and ${}^{+}P$, is similar. Note that $M = O(w^{1/3})$ and $f = O(\log w)$ so $M^2 f \leq w$ for large enough w.

6.2 Conclusion

All the constituent parts and algorithms for the small priority queue having now been described, we go over a couple of small details before claiming the following main result:

Theorem 2 *The dynamic predecessor problem on n keys of w bits each, where $n \leq w^\epsilon$ for some fixed $\epsilon > 0$ can be solved in $O(1)$ expected time on a unit-cost RAM with w bits assuming oblivious updates.*

PROOF. Firstly, whether or not g is 1-1 is irrelevant—so long as we get the right answer to the queries we are happy, and if we make sure that the sorted order on S given by the linked list is correct after each update, we can check the correctness of the answers to the queries, and re-build the data structure with a new f (in $O(n)$ time) when any query turns out wrong. On the other hand, if g is 1-1 on the current set S and the input key x, the answer will be correct. As g is 1-1 on the set of characters in $S \cup \{x\}$ with probability greater than $1 - 1/n$, the probability that we will re-build is bounded by $1/n$, giving an expected update cost of $O(1)$.

As described above, the data structure needs $O(\log w)$ preprocessing time to obtain constants of the form $1_{M,f}$. By setting $M = \min\{w^{1/3}, n\}$ and $k = \min\{\sqrt{w/\log w}, n\}$ this cost becomes $O(\log n)$ which can be absorbed into the cost of updates (by periodically rebuilding the data structure, e.g.). □

Corollary 1 *The dynamic predecessor problem on n keys of w bits each can be solved in $O(\min\{\log w, \log n/\log w + 1\})$ expected time on a unit-cost RAM with w bits assuming oblivious updates.*

PROOF. We maintain a B-tree with branching factor $\min\{w^{1/3}, n\}$, storing the splitter keys using Theorem 2 for $w \geq 2^{\sqrt{\log n}}$, switching to the van Emde Boas data structure for smaller values of w. Note that updates to all B-tree nodes are oblivious as the randomization is not used to determine the structure of the B-tree in any way. □

7 Static deterministic data structures

In this section we will show how to obtain a static deterministic small priority queue based on the the data structure of Theorem 2. The main obstacle is that the data structure of Theorem 2 uses randomization to obtain a signature function, which we overcome in Subsection 7.1 by giving an efficient deterministic method to find a "good" hash function from a class defined by Dietzfelbinger et al. [9]; some consequences for the complexity of computing a "perfect" hash function are also mentioned. In Subsection 7.2 we then return to the main theme of this paper and give deterministic data structures for the predecessor problem.

7.1 Deterministic Hashing

Let $b \geq 0$ be an integer and let $U = \{0, \ldots, 2^b - 1\}$. A class $\mathcal{H}_{b,s}$ of hash functions from U to $\{0, \ldots, 2^s - 1\}$ for any integer s, $b \geq s \geq 0$ was defined by Dietzfelbinger et al. [9] as follows: $\mathcal{H}_{b,s} = \{h_a \mid 0 < a < 2^b, \text{ and } a \text{ is odd}\}$ and for all $x \in U$:

$$h_a(x) = (ax \bmod 2^b) \operatorname{div} 2^{b-s}.$$

For any $T \subseteq U$ and any $h : U \to \{0, \ldots, m-1\}$ let $\mathrm{coll}(h, T)$ be the number of *collisions* when h is applied to T, that is,

$$\mathrm{coll}(h, T) = |\{(x, y) \mid x, y \in T, x < y \text{ and } h(x) = h(y)\}|.$$

Given a set $T \subseteq U$ and $s \geq 0$ we are interested in deterministically picking $h_a \in \mathcal{H}_{b,s}$ such that $\mathrm{coll}(h_a, T)$ is "small". Specifically, for some value t satisfying $t \geq \mathrm{E}[\mathrm{coll}(h_a, T)]$, where the expectation is calculated assuming h_a is picked uniformly at random from $\mathcal{H}_{b,s}$, we wish to find an $h_a \in \mathcal{H}_{b,s}$ such that $\mathrm{coll}(h_a, T) \leq t$. This can of course always be done—by individually testing each hash function in $\mathcal{H}_{b,s}$ for example—but in order to do this more efficiently, we use the *method of conditional probabilities* [16], which we briefly review below. First, however, we introduce some notation. Let $a_0, \ldots, a_{b-1}$ denote the bits comprising a, with a_0 and a_{b-1} being the least and most significant bits respectively. We denote the expectation of a random variable X by $\mathrm{E}[X]$, and its expectation conditioned on A as $\mathrm{E}[X \mid A]$. For any string $\alpha \in \{0,1\}^*$ of length $b-1$ or less, we abbreviate $\mathrm{E}[X \mid a_0 = 1 \text{ and } a_1 \ldots a_{|\alpha|} = \alpha]$ by $\mathrm{E}[X \mid \alpha]$.

Suppose $\widehat{f}$ is a real-valued function on the set of all strings over $\{0,1\}$ of length $b-1$ or less, satisfying the following properties:

(1) $\widehat{f}(\Lambda) \leq t$.

(2) For all $\alpha \in \{0,1\}^*$, $|\alpha| \leq b-1$, $\mathrm{E}[\mathrm{coll}(h_a, T) \mid \alpha] \leq \widehat{f}(\alpha)$.

(3) For all $\alpha \in \{0,1\}^*$, $|\alpha| < b-1$, $\min(\widehat{f}(\alpha 0), \widehat{f}(\alpha 1)) \leq \widehat{f}(\alpha)$.

(4) $\widehat{f}$ is efficiently computable.

Given such a $\widehat{f}$ we can find h_a such that $\mathrm{coll}(h_a, T) \leq t$ as follows: we set a_0 to 1 and for $i = 1, \ldots, b-1$, we set a_i to $x \in \{0,1\}$ such that $\widehat{f}(\alpha x) = \min(\widehat{f}(\alpha 0), \widehat{f}(\alpha 1))$, where α denotes the string $a_1 \ldots a_{i-1}$. Note that each such step maintains the invariant $\mathrm{E}[\mathrm{coll}(h_a, T) \mid \alpha] \leq \widehat{f}(\alpha) \leq t$, as $\mathrm{E}[\mathrm{coll}(h_a, T) \mid \alpha x] \leq \widehat{f}(\alpha x) \leq \widehat{f}(\alpha) \leq t$. Proceeding this way we obtain a setting for all bits of a such that the number of collisions is bounded by t. We now show:

Lemma 4 *Given integers $b \geq s \geq 0$ and $T \subseteq \{0, \ldots, 2^b - 1\}$ with $|T| = n$, and $t \geq 2^{-s+1}\binom{n}{2}$, a function $h_a \in \mathcal{H}_{b,s}$ can be chosen in $O(n^2 b)$ time such that $\mathrm{coll}(h_a, T) \leq t$.*

Proof. By the discussion above, all we have to do now is to define the function $\widehat{f}$ and show it satisfies properties (1)–(4) above. In what follows, expressions such as $p \bmod q \in S$ should be interpreted as meaning $p \equiv r \pmod q$, for some $r \in S$. For any $x, y \in T$ and $\alpha \in \{0,1\}^*$, $|\alpha| \leq b-1$, let $\widehat{f}(\alpha) = \sum_{x,y \in T, x<y} \delta_{xy}(\alpha)$, where

$$\delta_{xy}(\alpha) = \Pr[a(x-y) \bmod 2^b \in \{-2^{b-s}+1, \ldots, 0, \ldots, 2^{b-s}-1\} \mid \alpha].$$

Dietzfelbinger et al. [9] noted that $\Pr[h_a(x) = h_a(y) \mid \alpha] \leq \delta_{xy}(\alpha)$, and showed that $\delta_{xy}(\Lambda) \leq 2^{-s+1}$. Properties (1)–(3) can now be verified respectively as follows:

(1′) $\widehat{f}(\Lambda) = \sum_{x,y \in T, x<y} \delta_{xy}(\Lambda) \leq \binom{n}{2} 2^{-s+1} \leq t$.

(2′) $\mathrm{E}[\mathrm{coll}(h_a, T) \mid \alpha] = \sum_{x,y \in T, x<y} \Pr[h_a(x) = h_a(y) \mid \alpha] \leq \sum_{x,y \in T, x<y} \delta_{xy}(\alpha) = \widehat{f}(\alpha)$.

(3′) For all $x, y \in T, x < y$, $\alpha \in \{0,1\}^*$, $|\alpha| < b-1$, $\delta_{xy}(\alpha) = \frac{1}{2}(\delta_{xy}(\alpha 0) + \delta_{xy}(\alpha 1))$, from which Property (3) follows.

It remains to show how to compute $\delta_{xy}(\alpha)$ quickly. Firstly:

$$\begin{aligned} \delta_{xy}(\alpha) &= \Pr[\Big((a' 2^{|\alpha|+1} + 2\alpha + 1)(x-y)\Big) \bmod 2^b \in \{-2^{b-s}+1, \ldots, 0, \ldots, 2^{b-s}-1\}] \\ &= \Pr[\Big((a' 2^{|\alpha|+1})(x-y)\Big) \bmod 2^b \in \{l, \ldots, u\}], \end{aligned}$$

where a' is chosen uniformly over $\{0,\ldots,2^{b-|\alpha|-1}-1\}$ and $l=-2^{b-s}+1-(2\alpha+1)(x-y)$ and $u=2^{b-s}-1-(2\alpha+1)(x-y)$. We write $x-y$ as $z2^i$ where i,z are integers such that $i\geq 0$ and z is odd. Then $((a'2^{|\alpha|+1})(x-y)) \bmod 2^b = (a'z2^{|\alpha|+i+1}) \bmod 2^b$ is identically 0 with probability one if $|\alpha|+i+1\geq b$. Otherwise this quantity can be re-written as $((a'z) \bmod 2^{b-|\alpha|-i-1})\cdot 2^{|\alpha|+i+1}$. Since z, being odd, has a multiplicative inverse over the numbers modulo $2^{b-|\alpha|-i-1}$ it is easy to see that $(a'2^{|\alpha|+1})(x-y)$ is uniform over all integer multiples of $2^{|\alpha|+i+1}$ in the range $0\,..\,2^b-1$. Computing $\delta_{xy}(\alpha)$ therefore simply reduces to counting the number of integral multiples of $2^{b-|\alpha|-i-1}$ in the range $l\,..\,u$ (considered modulo 2^b) and can be done in $O(1)$ time given the quantity i above. Since the value i can be computed in $O(b)$ time for each pair x,y prior to picking the bits of a, it follows that a function h_a can be found with the required property in $O(n^2b)$ time. □

Corollary 2 *Given an integer $b\geq 0$ and $T\subseteq\{0,\ldots,2^b-1\}$ with $|T|=n$, a function $h\in\mathcal{H}_{b,s}$ can be found in $O(bn^2)$ time for which* $\mathrm{coll}(h,T)=0$, *for some s such that $2^s\leq 2n^2$.*

Proof. Choose $s=\lceil 2\log n\rceil$ and apply Lemma 4 with $t=2^{-s+1}\binom{n}{2}$. Since $2^s\geq n^2$, $t<1$ and by Lemma 4 a function $h\in\mathcal{H}_{b,s}$ can be found in $O(n^2b)$ time with $\mathrm{coll}(h,T)<1$ and hence with $\mathrm{coll}(h,T)=0$. Note that $2^s\leq 2n^2$. □

Given a set $T\subseteq U$ a hash function $h:U\to\{0,\ldots,m-1\}$ is said to be *perfect* for T if $\mathrm{coll}(h,T)=0$. What we also require is that $h(x)$ is computable in $O(1)$ time for any $x\in U$ (it is also desirable that the program to compute h be succinct, see e.g. [15, p. 127 ff.]).

Corollary 3 *Given an integer $b\geq 0$ and a set $T\subseteq U=\{0,\ldots,2^b-1\}$, a perfect hash function $h:U\to\{0,\ldots,m-1\}$ for some $m=O(n)$ can be computed in $O(n^2b)$ time.*

Proof. We use the two-level scheme of [12], modified to use the class of hash functions defined by [9]. Firstly we use Lemma 4 to find in $O(n^2b)$ time a "top-level" hash function $h[-1]$ such that $\mathrm{coll}(h,T)\leq n$; it is easy to see that $s=s_{top}=\log n+O(1)$ suffices. For $i=0,\ldots,2^{s_{top}}-1$, we let B_i denote the set of items from T mapped to the value i by $h[-1]$ and to find a "secondary" hash function $h[i]:U\to\{0,\ldots,m-1\}$ for some $m=O(|B_i|^2)$, such that $h[i]$ is perfect on B_i; by Corollary 2 this can be done in $O(b|B_i|^2)$ time. As shown in [12], $\sum_i|B_i|^2=O(n)$, implying both that the total space for the two-level data structure is $O(n)$ and that the running time for the second phase is $O(bn)$, yielding an overall $O(bn^2)$ running time. □

Remark: Corollary 3 improves the complexity of deterministically computing a perfect hash function from $O(bn^3)$ (as implied by [12, Lemma 3]) to $O(bn^2)$. Furthermore, our result does not require that the universe size be prime (see [8] for other results in this respect). On the other hand, the program computing the perfect hash function of [12] can be represented with $O(n\log n+\log b)$ bits [15, Theorem 13, pp. 138–139], while we do not know if the program computing our perfect hash function can be represented with fewer than $\Theta(n\log n+b)$ bits.

7.2 Deterministic small priority queues

We now return to the problem of constructing a small priority queue. The construction of Theorem 2 used randomization only to compute the signature function g. What we require

of the signature function in this context is that not only is it $1-1$ on all the fields occurring in the input set S (which is trivial to ensure during the pre-processing), but also that it should be possible to evaluate the concatenated signature of a search key in $O(1)$ time.

Given a set S of keys, we can therefore deterministically pre-process the characters of the individual keys to obtain a signature function g that is 1-1 on S using Corollary 2. While answering queries, we apply the same function g to the search key x and find the element y in $g(S)$ whose common prefix with $g(x)$ is the longest. If $lcp(g^{-1}(y), x)$ is shorter than $lcp(y, g(x))$, then we find the first character of x where f has failed and replace the signature in this position by a value that is known not to be the signature of any character in S, getting a modified signature $g'(x)$. Note that $fail(g'(x), g(S)) = g(fail(x, S))$, and since the rest of the search is not affected by f, the correct answer will be computed.

We thus obtain a deterministic small priority queue with $(n+w)^{O(1)} = w^{O(1)}$ preprocessing time. This priority queue occupies only $n+2$ locations for all $n \leq w^{1/3}$: all the intra-word data structures use $o(w)$ bits and can be stored together in a single word of w bits, and one extra word is needed to store the multiplier for the hash function. It is straightforward to thereby obtain a static data structure with linear space and $O(nw^{\epsilon})$ preprocessing time with query time $O(1 + \log n / \log w)$. Using Corollary 3 we can store the data structure of Theorem 1 (or other similar data structures [10, 18]) in a hash table to reduce its space usage to linear in n. Thus we can obtain a linear-space static data structure with $O(\min\{\log w, 1 + \log n / \log w\})$ query time.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Dokl. Akad. Nauk SSSR*, **146** (1962), pp. 263–266 (in Russian).

[2] M. Ajtai, M. Fredman and J. Komlós. Hash functions for priority queues. *Inform. and Control*, **63** (1984), pp. 217–225.

[3] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th IEEE FOCS*, 1995.

[4] A. Andersson, T. Hagerup, S. Nilsson and R. Raman. Sorting in linear time? In *Proc. 27th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 427–436, 1995.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, **18** (1979), pp. 143–154.

[6] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. WADS' 89*, LNCS 382, pp. 39–46, 1989.

[7] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM SODA*, pp. 77–87, 1991.

[8] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. To be presented at *STACS '96*.

[9] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Tech. Rep. no. 513, Fachbereich Informatik, Universität Dortmund, 1993.

[10] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, **6** (1977), pp. 80–82.

[11] P. van Emde Boas, R. Kaas, and E. Ziljstra. Design and implementation of an efficient priority queue. *Math. Sys. Theory*, **10** (1977), pp. 99–127.

[12] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, **31** (1984), pp. 538–544.

[13] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, **47** (1993), pp. 424–436.

[14] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, **48** (1994), pp. 533-551.

[15] K. Mehlhorn. *Data Structures and Algorithms, Vol. I: Sorting and Searching*. Springer-Verlag, Berlin, 1984.

[16] P. Raghavan. Lecture notes in randomized algorithms. TR RC 15340, IBM T. J. Watson Research Center, 1989.

[17] M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM SODA*, pp. 59–67, 1995.

[18] D. E. Willard. Log-logarithmic worst case range queries are possible in space $O(N)$. *Info. Proc. Lett*, **17** (1983), pp. 81–89.

[19] Andrew C. Yao. Should tables be sorted? *J. ACM*, **28** (1981), pp. 615–628.