Priority queues: small, monotone and trans-dichotomous

Rajeev Raman

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

Abstract. We consider two data-structuring problems which involve performing priority queue (PQ) operations on a set of integers in the range $0..2^w - 1$ on a unit-cost RAM with word size w bits.

A *monotone min-PQ* has the property that the minimum value stored in the PQ is a non-decreasing function of time. We give a monotone min-PQ that, starting with an empty set, processes a sequence of n insert and delete-mins and m decrease-keys in $O(m + n\sqrt{\log n \log\log n})$ time. As a consequence, the single-source shortest paths problem on graphs with n nodes and m edges and integer edge costs in the range $0..2^w - 1$ can be solved in $O(m + n\sqrt{\log n \log\log n})$ time, and n integers each in the range $0..2^w - 1$ can be sorted in $O(n\sqrt{\log n \log\log n})$ time. All the above results require linear space and assume that any unit-time RAM instructions used belong to the the class AC^0.

A *small (generalized) PQ* supports insert, delete and search operations (the latter returning the predecessor of its argument among the keys in the PQ), but allows only $w^{O(1)}$ keys to be present in the PQ at any time. We give a small PQ which supports all operations in constant expected time. As a consequence, we get that insert, delete and search operations on a set of n keys can be performed in $O(1+\log n/\log w)$ expected time. Derandomizing this small PQ gives a linear-space static deterministic small PQ.

1 Introduction

We consider two data-structuring problems which involve performing priority queue (PQ) operations on a set of keys:

(1) A *monotone min-PQ* has the property that the value of the minimum key stored in the PQ is a non-decreasing function of time. The monotone min-PQ should, starting with an empty set, process a sequence of insert, delete-min and decrease-key operations. Monotone min-PQs have numerous applications, most notably in greedy algorithms such as Dijkstra's algorithm for single-source shortest paths and Huffman's algorithm for computing optimal prefix-free codes.

(2) A *(generalized) PQ* supports insert, delete and search operations, where search returns the predecessor of its argument among the keys in the PQ. Also known as *search trees*, generalized PQs are among the most fundamental of data structures.

Most of the research regarding PQs has been in the *comparison-based* model, where one may obtain information about the relative order of keys exclusively through pairwise comparisons. The model used in this paper is the unit-cost RAM model with a word length of w bits, where w is a parameter of the model. The keys are assumed to be integers in the range $0..2^w - 1$.

We assume that the RAM can perform addition, subtraction, bitwise logical operations, comparison, arbitrary left and right bit shifts (with zero filling) and multiplication in constant time on $O(w)$-bit operands. We also assume that the RAM can address a total of 2^w words of memory, each of which contains w bits, and can also generate a random w-bit integer in constant time. Finally, we assume that the input fits into 2^{w-c} words of memory for some large enough constant $c > 0$, thus always leaving at least linear working space for the algorithms.

Reformulating classical problems in such *trans-dichotomous* models [12] is related to the older tradition of “finite-universe” reformulations of problems. However, we aim to obtain *strongly* trans-dichotomous algorithms, whose running times improve upon the running times of their classical counterparts *regardless* of the relative sizes of the input size, n, and the word length, w. The first example of such an algorithm was also given by Fredman and Willard [11].

A major reason for preferring this model to the (more elegant) comparison-based one is that many real-life instances of the PQ problems considered in this paper deal with integer or floating-point keys. Our model is clearly well-suited for handling integer keys, but it turns out that our algorithms would work without modification even with floating-point keys conforming to the common IEEE 754 standard [13]. As algorithms in our model are not fettered by the lower bounds that apply to their comparison-based counterparts, there is the potential for getting algorithms that are faster both theoretically and in practice.

However, the practical utility of our algorithms may be diminished if certain weaknesses in the model are exploited. Firstly, it may be unrealistic to assume constant-time access to 2^w words of memory, as the physical memory on most computers is far smaller than the address space. For this and other reasons, it is important that the algorithms minimize the amount of working space used. Another point of concern relates to the validity of the unit-cost assumption for long word sizes. Specifically, it would be desirable to use the *restricted* instruction set [5], which consists of all the instructions above except multiplication. This is because the restricted instruction set is comprised entirely of AC^0 functions, for which constant-depth circuits of polynomial size exist; multiplication is known not to be in AC^0. From the practical viewpoint as well, avoiding multiplication may be a good idea on some RISC architectures.

Both our main results use the top-level structure of [5], and combine the two notions of *packed* data structures and *range reductions*. A packed data structure obtains its speed by packing several keys into a single word and operating simultaneously on all of them at unit cost. This is only possible, of course, if several keys fit in one word. Range reduction, on the other hand, reduces a data structuring problem on long keys to one on shorter keys. The combination of the two techniques is straightforward: range reduction is applied to reduce the size of the keys sufficiently to make use of the speedups of packed data structures. We now describe our results and compare them with previous results.

Monotone min-PQs: We give a monotone min-PQ which stores a set of integer keys in the range $0 \,..\, 2^w - 1$, and which, starting with an empty set, processes a sequence of n insert and delete-mins and m decrease-keys in $O(m+n\sqrt{\log n \log\log n})$ time. This data structure requires linear space and uses the restricted instruction set, and is formed by a simple combination of four different PQs: the *radix heap* of Ahuja et al.[1], which we use as a range reduction; a packed PQ based on repeated merging developed by Thorup [20], the *AF-heap* of Fredman and Willard [12], which belongs to the Fibonacci heap [10] family and a new packed PQ based on PQs for linear systolic arrays. In the interests of brevity, we will not compare this result directly with previous results [20, 12, 10], focusing instead on its applications:

1. The *single-source shortest paths* problem on a graph with n nodes and m edges and integer edge costs in the range $0..2^w - 1$ can be solved in $O(m + n\sqrt{\log n \log\log n})$ time and linear space by using our data structure in Dijkstra's algorithm. We now compare this result with previous ones (a † indicates that the underlying PQ does not need the monotonicity assumption):

(a†) The fastest comparison-based algorithm runs in $O(m + n \log n)$ time [10].

(b) An algorithm of Ahuja et al. [1] runs in $O(m + n\sqrt{w})$ time and requires $O(m + \sqrt{w}2^{\sqrt{w}})$ space; ours is faster for $w \geq \log n \log\log n$ and requires linear space.

(c†) An algorithm of Fredman and Willard [12] runs in $O(m + n \log n / \log\log n)$ time and assumes unit-time multiplication; ours is faster and uses only AC^0 instructions.

(d) An algorithm of Thorup [20] runs in $O(m + n(\log n)^{1/2+\epsilon})$ expected time for any constant $\epsilon > 0$ and uses unit-time multiplication. Ours is deterministic, uses only AC^0 instructions, and at first sight, seems faster as well. However, as w increases beyond its minimum value of $\Theta(\log n)$, Thorup's algorithm speeds up, and becomes faster than ours if $w \geq (\log n)^{1+\epsilon}$ for some fixed $\epsilon > 0$.

(e†) Another of Thorup's algorithms [20] runs in $O(m \log\log n)$ time but requires $O(m + 2^{\epsilon w})$ space for any fixed $\epsilon > 0$. Ours is linear-space and runs faster on dense graphs.

Summarizing the above, in terms of speed alone, our algorithm is the fastest for graphs of a certain density and for a certain range of word-sizes. If we restrict ourselves to deterministic linear-space algorithms, we improve upon the algorithm of Ahuja et al. (which uses linear space for small w) when $w \geq \log n \log\log n$.

2. Our data structure can also be used to sort n integers in the range $0..2^w - 1$ in $O(n\sqrt{\log n \log\log n})$ time in linear space and using only AC^0 instructions. The only previous deterministic linear-space sorting algorithm, due to Fredman and Willard [11], runs in $O(n \log n / \log\log n)$ time and also assumes unit-time multiplication. Other sorting algorithms proposed in [5, 11] are either randomized or use super-linear space. Very recently, Andersson [4] has described a new data structure for searching, an immediate consequence of which is a deterministic $O(n\sqrt{\log n})$-time linear-space sorting algorithm. However, his result needs unit-time multiplication and so is not directly comparable with ours.

Small PQs: A *small* generalized PQ allows a maximum of $w^{O(1)}$ keys to be present in the PQ at any time. We give a small PQ which supports insert, delete and search operations in constant expected time, assuming the updates are *oblivious*, i.e., independent of the random choices made by the data structure. Previous work on this problem is as follows:

(f) Ajtai et al. [2] gave a small PQ which performs all operations in $O(1)$ time on the powerful *cell probe* model [22], where the complexity of an algorithm is measured solely in terms of the number of accesses it makes to a memory with an infinite number of w-bit words, with all other computation being free. They posed the question of whether a small PQ with the same complexity could be found for more realistic models.

(g) Fredman and Willard [11] gave a *static* small PQ which performs searches in $O(1)$ time after polynomial-time preprocessing and requires polynomial space.

(h) Another small PQ given by the same authors in [12] allows constant-time searches and updates, but requires pre-computed tables of size exponential in M, the maximum size of the PQ. 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.

Comparing our result with that of (h), our PQ can accommodate more keys at the expense of introducing randomization. Furthermore, derandomizing this small PQ gives a static deterministic small PQ which can be pre-processed in polynomial time to answer queries in constant time. This data structure uses $n+2$ locations to store a set of size $n \le w^{1/3}$ and hence $n(1+O(n^{-1/3}))$ space when $n = w^{O(1)}$, in contrast to the result of (g). The derandomization result also has some consequences to the problem of computing a perfect hash function deterministically. We now list some applications of these results:

1. As an immediate consequence, we obtain that a set of n integer keys can be maintained under the operations of insert, delete and search, with each operation taking $O(1+\log n/\log w)$ time. In essence, we can store the keys in a B-tree with branching factor $\Theta(w)$ and store the "splitter" keys which guide the searches in the internal nodes in a small PQ. Since constant expected time is spent at each level, a search operation takes time proportional to the height of the tree, or $O(1+\log n/\log w)$ time in all. The same holds for insertions or deletions.

This running time is better than the $O(\sqrt{\log n})$ running time of [11, 3] if $\log w = \omega(\sqrt{\log n})$. Note that this is also precisely the range of w for which the data structures of [11, 3] improve upon that of van Emde Boas [8]. A very recent deterministic data structure due to Andersson [4] matches our running time whenever $\log w = O(\log n/\log\log n)$.

2. Plugging the small PQ result into the AF-heap data structure of Fredman and Willard [12] we get a min-PQ with insert and decrease-key taking constant expected time, while delete and delete-min take $O(1+\log n/\log w)$ expected time (cf. Lemma 3). This is faster than the data structure of [12] when $w = (\log n)^{\omega(1)}$, but does not yield new results for the shortest-paths problem, as the *monotone* min-PQ of Thorup [20] is faster for all values of n relative to w.

3. Using our small PQ in place of that of [12] in Thorup's reduction from monotone min-PQs to sorting [20, Section 3.1] simplifies the reduction, as one component [20, 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 (result (d) above), which is already a randomized result.

2 Monotone Min-PQs

In this section we sketch a proof of the following main theorem.

Theorem 1. *There is a monotone min-PQ that, starting with an empty set, processes a sequence of n* insert *and* delete-min*s and m* decrease-key*s in $O(m + n\sqrt{\log n \log\log n})$ time. This data structure requires linear space and uses the restricted instruction set.*

The proof will be in four parts. First we give an overview of the range reduction, followed by a description of the packed data structure in two parts, after which we put the pieces together.

2.1 Range Reduction

The range reduction is effected by the *radix heap* data structure of Ahuja et al. [1]. In order to clarify its use as a range reduction, rather than as a PQ in its own right, as well as for the sake of a self-contained presentation, we now give a high-level description of their data structure. We also re-interpret their data structure in a manner inspired by [20]; making it, in our opinion, conceptually simpler. Let $1 \leq k \leq w$ be an integer whose value will be determined later. We consider each key as a string of k characters of w/k bits each, and denote the characters comprising an integer x by $x[1], \ldots, x[k]$, with $x[1]$ comprising the most significant bits and $x[k]$ the least significant. The radix heap associates an *index* with each key in the data structure, which is an ordered pair of numbers $\langle i, j \rangle$, $0 \leq i \leq k$ and $0 \leq j \leq 2^{w/k} - 1$. Let μ be the current minimum key value. The index of a key with value μ is defined to be $\langle 0, 0 \rangle$ by convention. For a key with value $x > \mu$, $index(x) = \langle k-i, x[i+1] \rangle$, where $i < k$ is the length of the longest common prefix of x and μ. Observe that for any two keys x, y:

$$index(x) < index(y) \Rightarrow x < y \qquad (*)$$

where indices are compared lexicographically. The set of key indices is maintained in a separate data structure called the *index PQ*. The data structure translates the top-level monotone PQ operations into (non-monotone) PQ operations on the index PQ. Since the indices may be considered as integers in the range $0 \mathinner{..} (k+1)2^{w/k} - 1$ and $(k+1)2^{w/k} - 1 \ll 2^w$ if k is chosen appropriately, this effects a range reduction for PQ operations. The operations on the top-level PQ are implemented as follows:

1. insert: the index of the new key x is computed and inserted into the index PQ. In order to do so, we check prefixes of x of length $1, 2, 3, \ldots$ to find the largest i such that $x[1] \mathinner{..} x[i]$ is a prefix of μ in $O(i+1)$ time.

2. $\mathsf{decrease\text{-}key}$: the index of the key x whose value is decreased is recomputed as follows: if the first component of the current index has value j, we check prefixes of the new value of x against μ, starting with $x[1] \mathinner{..} x[j]$, until we find the largest $i \geq j$ such that $x[1] \mathinner{..} x[i]$ is a prefix of μ, and thereby compute the new index. This takes $O(i - j + 1)$ time. If the new index is strictly smaller, a $\mathsf{decrease\text{-}key}$ is done on the index PQ.

3. $\mathsf{delete\text{-}min}$: a $\mathsf{delete\text{-}min}$ is performed on the index PQ, which gives a key with index $\langle 0, 0 \rangle$, and hence with minimum value. After this, a $\mathsf{find\text{-}min}$ is performed to get the value of the new minimum index ix. If $ix = \langle 0, 0 \rangle$, then the minimum value in the top-level PQ has not increased and nothing else happens.

Otherwise, the new minimum key(s) in the top-level PQ must also have index equal to ix, by (*) above. We locate *all* key(s) with index equal to ix and scan them to determine the new minimum key(s), and recompute indices relative to the new minimum. Note that keys with current index greater than ix do not have their index changed. Also, all keys whose current index equals ix get a new index which is *strictly* smaller in the first component, and these keys have their index recomputed as for the $\mathsf{decrease\text{-}key}$ operation. Keys which equal the new minimum value have their index set to $\langle 0, 0 \rangle$. Finally, a $\mathsf{decrease\text{-}key}$ operation is then performed on the index PQ for each key whose index has become smaller.

As can now be verified, a sequence of n insert and $\mathsf{delete\text{-}min}$ operations and m $\mathsf{decrease\text{-}key}$ operations at the top level results in at most $m + (k+1)n$ $\mathsf{decrease\text{-}key}$

operations (each key can have the first component of its index decreased at most $k+1$ times as a result of delete-min operations), n insert and delete-min operations and n find-min operations on a set of integers (indices) in the range $0\,..\,(k+1)2^{w/k}-1$. Re-computing indices for any particular item incurs a book-keeping cost of $O(k+d)$ time, taken over the entire period that it has been in the data structure, where d is the number decrease-keys performed on this item. This adds up to $O(m+kn)$ time summed over all items, which is negligible.

The space bound of the range reduction is linear, except that a delete-min at the top level may require that a list of all keys with the current minimum index be computed. It appears that [1] implement this by keeping a bucket for each possible index value. This incurs a space cost of $O(k2^{w/k})$, which we avoid by noting that our index PQ can provide this list in time proportional to its size.

2.2 The Packed PQ

We begin by giving a constant-time PQ for small integers which does not incorporate the decrease-key operation. Specifically we prove:

Lemma 1. *There is a data structure which stores N keys of $O\left(w/\log N \log\log N\right)$ bits each and supports* insert, delete, delete-min *and* find-min *in constant amortized time each. The data structure requires linear space and uses the restricted instruction set.*

Our data structure is obtained by modifying one due to Thorup [20, Theorem 3.1], which achieves the time bounds above but requires unit-time multiplication, and also uses non-linear space. We now describe in turn how to modify his data structure in order to overcome each of these problems. Making these modifications proves Lemma 1.

Thorup's packed PQ needs non-linear space because it handles deletes by maintaining a bit-vector of deleted keys of size $2^{\Theta(w)}$ (this suffices for his results). In order to keep the space linear, we change the specification of the delete operation slightly from that given by Thorup. When a key is inserted into the PQ, a tag (which is an integer whose value is $O(N)$) is returned to the user, who must refer to a key to be deleted by its tag. In essence, it suffices to keep track of deleted tags, which requires a bit-vector of size $O(N)$. Details can be found in [19].

Also, Thorup's packed PQ needs a PQ which performs insert, delete-min and delete in constant time on a set of $O(\log N)$ keys of $O\left(w/\log N \log\log N\right)$ bits each. Thorup uses a PQ of [12] for this purpose, which uses the multiplication instruction. Thorup has claimed an $O(\log^{**} N)$-time solution using the restricted instruction set. We obtain a constant-time solution using the restricted instruction set by adapting a PQ for a linear systolic array; such PQs are apparently part of the folklore, see e.g. [14, Problem 1.119, p254]. We show:

Lemma 2. *For any integer k, $0 < k < w$, there is a data structure which stores up to k keys of at most $\lfloor w/k \rfloor - 1$ bits each, and which supports* insert, delete-min *and* find-min *in constant amortized time each. All operations except* delete *can be supported in constant worst-case time. The data structure requires linear space and uses the restricted instruction set.*

Proof. We describe only the find-min, insert and delete-min operations, as the data structure can be augmented with the delete operation in linear space by using tags as above (a direct approach to handling deletes is also possible). We divide a word into k consecutive fields of $\lfloor w/k \rfloor$ bits each which are numbered $1, \ldots, k$ from right to left. A field may store one key in a right-justified manner, with the leftmost bit of the field equal to zero (this bit will be used to compare field values, among other things). Let κ denote the number of keys currently in the data structure. The keys are stored in fields $1, \ldots, \kappa$, and the key stored in the i-th field is denoted as x_i. Although the keys will not always be in sorted order, x_κ will be the smallest element in the PQ at the end of each operation.

Let $comp(i, j)$, for $1 \leq j < i \leq k$ denote the operation which moves $\min\{x_i, x_j\}$ to field i (the leftward field) and $\max\{x_i, x_j\}$ to field j (the rightward field). The *transpose* operation simultaneously executes $comp(i, i-1)$ for $i = \kappa, \kappa-2, \kappa-4, \ldots$ (if κ is odd, field number 1 is not involved in any *comp* operation). Using standard tricks, such as implementing multiple comparisons by subtraction [17], a transpose can be performed in constant time (see Appendix A).

A find-min is implemented by extracting the contents of field κ by means of a mask followed by a suitable shift. A delete-min saves the value returned by find-min, zeroes out field κ, decrements κ and performs a transpose. It then returns the saved value as the answer to the delete-min. An insert increments κ, places the new key into field number κ and performs a transpose. The proof of correctness of this algorithm is left as an exercise; a proof can also be found in [19]. □

2.3 A Packed PQ with decrease-key

The *AF-heap* is a variant of the Fibonacci heap [10] which was introduced by Fredman and Willard [12]. Given an integer parameter $4 \leq d \leq n$, the AF-heap maintains n keys in a forest of trees. Each tree is either a single node, or is a tree where all leaves are at the same depth, and all internal nodes have $\Theta(d)$ children. At any time, there are at most $d-1$ trees of the same height in the forest; since the number of different heights is $O(\log_d n)$, the total number of trees in the forest is $O(d \log n)$. Each (internal and external) node in the forest stores a key and the value stored at each internal node is no larger than the value stored at any of its children. The AF-heap uses a number of "atomic" PQs to make delete-min run in $o(\log n)$ time. All the keys stored at the roots of trees in the forest are kept in a top-level atomic PQ, and for every non-leaf node, the keys stored at all its children are also stored in an atomic PQ The complexities of the PQ operations of an AF-heap are given in the following lemma:

Lemma 3. *If the atomic PQ supports* insert, delete *and* find-min *operations on sets of size $O(d \log n)$ in constant amortized time, then the AF-heap supports* insert, decrease-key *and* find-min *in constant amortized time, and* delete-min *and* delete *in $O(\log n/ \log d)$ amortized time. Furthermore, all keys with minimum value can be listed in amortized time proportional to their number.*

Proof. The complexities are those given by Fredman and Willard [12] except for the operation of listing all keys with minimum key value. By performing a series of (constant-time) delete-min operations on the top-level atomic heap we obtain all the roots whose key value equals the minimum key. Placing these roots into a queue, we explore the forest in a breadth-first manner, using the atomic PQ associated with a node to determine if it has more children which need to be explored, all of whom we place in the queue. After each node has been explored, we restore the atomic PQ at that node by re-inserting all the key values that we deleted. Note that we do not modify the forest in any way, so the analysis of the running time is trivial. □

Plugging Lemma 1 into Lemma 3 and choosing $d = 2^{k/\log k}$ we get:

Corollary 1. *Given any $k \geq 16$ and a collection of n keys of $O(w/k)$ bits each, we can support* insert, decrease-key *and* find-min *in constant amortized time, and* delete-min *and* delete *in $O(\log n \log k/k)$ amortized time. Furthermore, all keys with minimum value can be listed in amortized time proportional to their number. The data structure requires linear space and uses the restricted instruction set.*

2.4 Conclusion

We are now in a position to prove Theorem 1. We use the radix heap range reduction with $k = \lceil \sqrt{\log n \log \log n} \rceil$. The range reduction then reduces the problem of performing n insert and delete-min and m decrease-key operations in a monotone PQ in $O(m + kn)$ time to the problem of performing $O(m + kn)$ decrease-key operations, n find-min operations and n insert and delete-min operations on keys in the range $0 \mathinner{..} (k+1)2^{w/k} - 1$, i.e., keys with $O(w/k)$ bits each. By Corollary 1, the overall running time is $O(m + kn + n \cdot \frac{\log n \log k}{k}) = O(m + n\sqrt{\log n \log \log n})$.

3 Dynamic Small PQs

In this section, we prove the following main theorem:

Theorem 2. *There is a randomized data structure which performs* search *and oblivious* insert *operations on a set of w-bit integer keys in constant expected time, provided the set has at most $M = \lfloor w^{1/3} \rfloor$ keys.*

We will not discuss deletes in this abstract. Note that the theorem can be extended trivially to allow sets of maximum size $w^{O(1)}$. We begin by introducing some notation, follow it up with an overview in Section 3.1, discuss the implementation in Section 3.2 and put the pieces together in Section 3.3

We now describe an extension to the notation developed in [5] to facilitate the expression of bit-level operations. 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 can be 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, OR etc., 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 $x \cdot y$.

We now describe a set of constant-time operations, whose implementations can be found in [5]. In the following, we assume the (M, f)-representation is 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 obtained in $O(\log M)$ time from M and f.

For operating on boolean sequences, the operations for element-wise 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 $b_i = \mathit{true}$ if and only if $x_i \text{ OP } y_i$, for $i = 1, \ldots, M$. Also, $[X.\mathit{name} \text{ OP } Y.\mathit{name}]$ denotes the element-wise comparison of two sequences of records based on a particular component with name *name*. Finally, given a single record or integer z, we use the notation $[X \text{ OP } z]$ or $[X.\mathit{name} \text{ OP } z.\mathit{name}]$ 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 $b_i = \mathit{true}$ 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 $\mathit{leftmost}(X)$, 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 [5, Lemma 1] (see also [11]). 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 $Y := X \mid [X.\mathit{name} = \mathit{value}]$ and returning $Y \downarrow ((\mathit{leftmost}(Y) - 1)f)$.

As a warm-up, we now describe a PQ which supports insert and search on a set S of at most M records. We assume each record fits in a field of f bits with the leftmost bit of the field equal to zero.

Lemma 4. *Provided that $Mf \leq w$ and $f \geq \log M + 2$ we can perform* insert *and* search *in constant time each.*

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 do a search for a given record x we first let $A := [D.\mathit{key} \leq x.\mathit{key}]$ and obtain the rank r of $x.\mathit{key}$ as $r := (A \cdot 1_{M,f}) \downarrow (M-1)f$ as described in [11]. We then compute $i := \mathit{leftmost}([R = r])$ and return $(D \mid [R = r]) \downarrow (i-1)f$ as the answer if $i > 0$, and zero otherwise. In order to do an insert, we place the new record x and its rank into fields $|S| + 1$ of D and R, respectively, and set $R := R + [D.\mathit{key} > x.\mathit{key}]$. □

3.1 Overview

Range reductions for searching: Let Σ^* denote the set of all strings over some finite alphabet Σ, and let Σ^i denote the set of all strings over Σ of length i, for any integer $i \geq 0$. Let $T \subset \Sigma^*$ be some finite set. For $x, y \in \Sigma^*$ let $\mathit{lcp}(x, y)$ denote the longest common prefix of x and y, and let $\mathit{pre}(T) = \{\mathit{lcp}(x, y) \mid x, y \in S, x \neq y\} \cup \{\Lambda\}$, where Λ denotes the empty string. The *compressed trie* for T is obtained from the usual trie (or digital search tree) by eliminating all nodes with only one child (see e.g. [15]). Each external node (leaf) of the compressed trie corresponds to a member of T, while each internal node can be identified with a member of *pre*(T). Furthermore, each member of *pre*(T) is determined by at least one pair of elements of consecutive rank in T (except possibly Λ).

We view a 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. Let S denote the current set of integers. We maintain three data structures which are:

(1) A *prefix* data structure which, given a query key x, quickly computes:

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

(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) The compressed trie for S, with all the standard information associated with the nodes and edges of a compressed trie—each internal node of the trie has a pointer to the smallest and largest elements stored under it and each edge emanating 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 an ordered doubly-linked list.

(3) A *siblings list* data structure, which stores 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 at least 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 of leaves 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 in S to the problem of locating a w/k-bit character of x among a set of w/k-bit characters. This constitutes a "range reduction" for searching.

Computing with signatures: We now adapt ideas first developed in [5] to compute *fail*(x, S) quickly. Recall that we view 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 $y \in \mathit{pre}(S)$, $g(y) \in \mathit{pre}(g(S))$, and each member of $\mathit{pre}(g(S))$ is the image under g of some element of $\mathit{pre}(S)$. Also, if g happens to be 1-1 on the characters in $S \cup \{x\}$ then $\mathit{fail}(g(x), g(S))$ is simply the image of $\mathit{fail}(x, S)$ under g, and it suffices to compute the former.

However, g is known only to be 1-1 on the characters in S. Therefore, we compute $y = \mathit{fail}(g(x), g(S))$ as before, and find $y' \in \mathit{pre}(S)$ such that $g(y') = y$. If y' is a prefix of the query key x, it follows that $y' = \mathit{fail}(x, S)$. 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 $z' \in \mathit{pre}(S)$ such that $g(z') = z$; it can easily be checked that z' is indeed the desired value of $\mathit{fail}(x, S)$.

In order to compute $\mathit{fail}(x, S)$ in constant time, we therefore need to compute $g(x)$ from x, to maintain the mapping between $\mathit{pre}(S)$ and $\mathit{pre}(g(S))$ and to compute $\mathit{fail}(y, g(S))$ for a (possibly modified) concatenated signature y, all in constant time. The latter is made possible by the fact that the concatenated signatures of the keys in $S \cup \{x\}$ are much shorter than the keys themselves.

In [5] it is shown that if g is chosen from a universal class of hash functions defined by Dietzfelbinger et al. [7], 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 $l+2$ 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 above. 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.)

3.2 Implementation

We now describe the implementation of these operations. For the rest of this section we 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.

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 $\mathit{pre}(g(S))$, ordered by their length (ties between prefixes of equal length are broken arbitrarily). The set $\mathit{pre}(g(S))$ is represented in the compact $(M, 2kf)$-representation, where $f = l + 2$ is chosen just large enough to hold a signature. Each field of $2kf$ bits in this word which stores a member of $\mathit{pre}(g(S))$ 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 charac-

ter, 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 $x \in \mathit{pre}(g(S))$ and i is a unique integer label in the range $(M+1) \,..\, 2M$. A new label is given to each member of $\mathit{pre}(g(S))$ 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. To avoid confusion, the external trie nodes will be labelled with the array indices where the corresponding keys are stored, i.e. with labels from the range $1 \,..\, M$. Each member of $\mathit{pre}(g(S))$ 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 $(\mathit{prev}, \mathit{next})$ 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 4.

Queries: The computation of $\mathit{fail}(g(x), g(S))$ 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 $\mathit{leftmost}([X \neq Y])$ against $\mathit{leftmost}(X)$. In fact, it turns out that given the string $g(x)$ we can simultaneously determine for each element of $\mathit{pre}(g(S))$, 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 [5, 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.

Updates: As the updates are oblivious, we may assume that g is 1-1 on all keys in $S \cup \{x\}$ with high probability. 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 $i := \mathit{leftmost}([P^{+}.\mathit{name} = u])$ 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 up one position 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 $[X.\mathit{name} \in Y.\mathit{name}]$ has the obvious meaning.

Lemma 5. *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 $[X \in Y]$.*

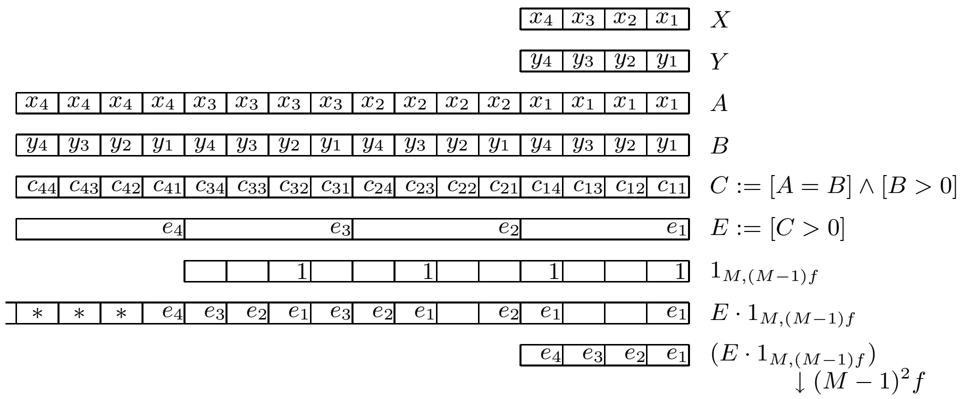

Fig. 1. Computing $[X \in Y]$ ($*$ denotes a don't care value).

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 [5, 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 $c_{ij} = \mathit{true}$ 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. $\square$

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 $F' := [({}^{+}P \mid F).\mathit{min} = v']$ 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.

3.3 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 Theorem 2. As the updates are oblivious, the probability that the signature function g is 1-1 on the current set S (including any newly-inserted keys) is at least $1 - 1/n$. Hence the probability that a query is answered correctly is also at least $1 - 1/n$. To convert this Monte Carlo procedure to a Las Vegas one, it is enough to note that errors can be detected easily: after every update we ensure that the linked list is in sorted order, which in turn enables us to check the answers to queries. If an error is detected we re-build the data structure with a new g (taking $O(n)$ expected time), but as the probability of this happening is bounded by $1/n$, the expected time for an update or query is $O(1)$.

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.).

4 Static Deterministic Small PQs

In this section show how to modify the data structure of Theorem 2 to obtain a *static* deterministic small PQ, which after polynomial-time preprocessing, answers queries in constant time. This PQ uses only $n+2$ locations to store a set of size $n \le w^{1/3}$, and hence $n(1+O(n^{-1/3}))$ space when $n = w^{O(1)}$. The main obstacle is that the signature function used in Theorem 2 is obtained by choosing a hash function at random from a class defined by Dietzfelbinger et al. [7]. We show to pick a "good" hash function deterministically from this class using the method of conditional probabilities [16].

We now describe the class of hash functions. Let $b \ge 0$ be an integer and let $U = \{0, \ldots, 2^b - 1\}$. This class $\mathcal{H}_{b,s}$ of hash functions from U to $\{0, \ldots, 2^s - 1\}$ is defined as follows: $\mathcal{H}_{b,s} = \{h_a \mid 0 < a < 2^b, \text{ and } a \text{ is odd}\}$ and for all $x \in U$:

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

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, i.e. the number of distinct pairs of elements of T mapped to the same value. Dietzfelbinger et. al showed that for any set T and a randomly chosen member h_a of $\mathcal{H}_{b,s}$, $\mathrm{E}[\mathrm{coll}(h_a, T)] \le 2^{-s+1}\binom{|T|}{2}$. Given a set T, we want to deterministically choose a multiplier a which gives no more than the expected number of collisions, and show:

Lemma 6. *Given integers $b \ge s \ge 0$ and $T \subseteq \{0, \ldots, 2^b - 1\}$ with $|T| = n$, and $t \ge 2^{-s+1}\binom{n}{2}$, a function $h_a \in \mathcal{H}_{b,s}$ can be chosen in $O(n^2 b)$ time such that* $\mathrm{coll}(h_a, T) \le t$.

Proof. 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 pick the bits of a sequentially beginning with a_0, which is always set to 1. 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 $\alpha \in \{0,1\}^*$ with $|\alpha| \le b-1$, we abbreviate $\mathrm{E}[X \mid a_0 = 1 \text{ and } a_1 \ldots a_{|\alpha|} = \alpha]$ by $\mathrm{E}[X \mid \alpha]$. Our pessimistic estimator for $\mathrm{E}[\mathrm{coll}(h_a, T) \mid \alpha]$ is $\widehat{f}(\alpha) = \sum_{x,y \in T, x<y} \delta_{xy}(\alpha)$, where for any $x, y \in T$,

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

Computing $\delta_{xy}(\alpha)$ simply reduces to counting the number of integral multiples of some powers of two in an interval of integers and can be done in $O(1)$ time, and hence $\widehat{f}(\alpha)$ can be computed in $O(n^2)$ time. Following [7] one can verify that this is indeed a pessimistic estimator. Details can be found in [18]. □

Using the same choice of parameters as Section 4.3, we can thus obtain a deterministic small PQ with $(n+w)^{O(1)}$ preprocessing time. This PQ 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. Reducing the preprocessing time to polynomial in n alone requires additional ideas not mentioned here. It is then easy to obtain a static data structure with linear space, $O(n^{1+\epsilon})$ preprocessing time and with $O(\min\{\log w, 1+\log n/\log w\})$ query time.

Given a set $T \subseteq U$ a hash function $h : U \to \{0,\ldots,m-1\}$ is said to be *perfect* for T if $\mathrm{coll}(h,T) = 0$; furthermore, we require that $h(x)$ is computable in $O(1)$ time for any $x \in U$ (see e.g. [15, p. 127 ff.]). Plugging Lemma 6 into the two-level scheme of [9], we obtain:

Corollary 2. *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 $O(n^2 b)$ time.*

REMARK: Corollary 2 improves the time complexity of deterministically computing a perfect hash function from $O(n^3 b)$, as implied by [9, Lemma 3], to $O(n^2 b)$. Furthermore, our result does not require knowledge of a prime larger than the universe size (see [6]).

References

1. R. K. Ahuja, K. Mehlhorn, J. B. Orlin and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, **37** (1990), pp. 213–223.
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. Faster sorting and searching in linear space. To be presented at *37th IEEE FOCS*, 1996.
5. A. Andersson, T. Hagerup, S. Nilsson and R. Raman. Sorting in linear time? In *Proc. 27th ACM STOC*, pp. 427–436, 1995.
6. M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. To be presented at *STACS '96*.
7. 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.
8. 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.
9. 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.
10. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization problems. *J. ACM*, **34** (1987), pp. 596–615.
11. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, **47** (1993), pp. 424–436.
12. 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.
13. J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publ., San Mateo, CA, 1994.

14. F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., San Mateo, CA, 1992.
15. K. Mehlhorn. *Data Structures and Algorithms, Vol. I: Sorting and Searching.* Springer-Verlag, Berlin, 1984.
16. R. Motwani and P. Raghavan. *Randomized Algorithms,* Cambridge University Press, Cambridge, 1996.
17. W. J. Paul and J. Simon. Decision trees and random access machines. In *Proc. International Symp. on Logic and Algorithmic*, Zürich, pp. 331–340, 1980.
18. R. Raman. Improved data structures for predecessor queries in integer sets. TR 96-07, King's College London, 1995 (revised 1996). `ftp://ftp.dcs.kcl.ac.uk/pub/tech-reports`.
19. R. Raman. Fast Algorithms for Shortest Paths and Sorting TR 96-06, King's College London, 1996. `ftp://ftp.dcs.kcl.ac.uk/pub/tech-reports`.
20. M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM SODA*, pp. 59–67, 1996.
21. D. E. Willard. Log-logarithmic worst case range queries are possible in space $O(N)$. *IPL*, **17** (1983), pp. 81–89.
22. Andrew C. Yao. Should tables be sorted? *J. ACM*, **28** (1981), pp. 615–628.

A The Transpose Operation

We now give a constant-time implementation of the transpose operation from Section 2.2. We use some of the notation from Section 3, but in order to make clear that we are using only AC^0 instructions, we give the code in terms of the built-in instructions rather than the more powerful operations defined there. The word p contains the keys, in f-bit fields, where $f = \lfloor w/k \rfloor$. Recall that a key is at most $f-1$ bits long, and the leftmost bit in each field of p is always zero. We assume the constants $k_1 = \sum_{i=1}^{k} 2^{if-1}$ and $k_2 = \sum_{i=1}^{\lfloor (k+1)/2 \rfloor} 2^{(2i-1)f-1}$ have been pre-computed. The code is as follows:

1 $\mathit{mask} \leftarrow$ **if** κ is even **then** k_2 **else** $k_2 \uparrow f$
2 $\mathit{res} \leftarrow$ NOT $(((p \uparrow f)$ AND $k_1) - p)$
3 $\mathit{res} \leftarrow \mathit{res}$ AND mask AND $(2^{\kappa} - 1)$
4 $\mathit{large} \leftarrow \mathit{res} - (\mathit{res} \downarrow (f-1))$
5 $\mathit{small} \leftarrow \mathit{large} \downarrow f$
6 $p \leftarrow p$ XOR $((p$ AND $\mathit{large}) \downarrow f)$
7 $p \leftarrow p$ XOR $((p$ AND $\mathit{small}) \uparrow f)$

After step (2), for $i = 2, \ldots, \kappa$, the leftmost bit of the i-th field is 1 iff $x_i > x_{i-1}$ [17]. In line (3) all but the rightmost κf positions are cleared, and only the results of the relevant comparisons are retained. Lines (4) and (5) compute a mask to extract large elements (which move to the right one position) and small elements (which move to the left one position) respectively, and lines (6) and (7) effect the movement.