PREPRINT. In Proc. 37th IEEE Symposium on Foundations of Computer Science, 1996.

Faster deterministic sorting and searching in linear space

Arne Andersson
Department of Computer Science, Lund University
Box 118, S–221 00 Lund, Sweden
arne@dna.lth.se

Abstract

We present a significant improvement on linear space deterministic sorting and searching. On a unit-cost RAM with word size w, an ordered set of n w-bit keys (viewed as binary strings or integers) can be maintained in

$$O\left(\min\left\{\begin{array}{l}\sqrt{\log n}\\ \frac{\log n}{\log w}+\log\log n\\ \log w\log\log n\end{array}\right\}\right)$$

time per operation, including insert, delete, member search, and neighbour search. The cost for searching is worst-case while the cost for updates is amortized. As an application, n keys can be sorted in linear at $O\left(n\sqrt{\log n}\right)$ worst-case cost.

The best previous method for deterministic sorting and searching in linear space has been the fusion trees which supports updates and queries in $O(\log n/\log\log n)$ amortized time and sorting in $O(n\log n/\log\log n)$ worst-case time.

We also make two minor observations on adapting our data structure to the input distribution and on the complexity of perfect hashing.

1 Introduction

Recently, a number of new findings on sorting and searching, including priority queues, have been presented [1, 2, 6, 11]. Most of them use superlinear space or randomization; the best deterministic method using linear space has been the fusion tree which supports dictionary operations in $O(\log n/\log\log n)$ amortized time [6]. Here, we present a significant improvement, showing that searches and updates can be performed in $O\left(\sqrt{\log n}\right)$ amortized time per operation and in linear space. This result also implies an improved worst-case bound on sorting in linear space, from $O(n\log n/\log\log n)$ to $O\left(n\sqrt{\log n}\right)$. (Very recently Raman [10] has developed a new priority queue with which it is possible to sort in deterministic linear space in $O(n\sqrt{\log n\log\log n})$ time. This complexity is slightly worse than the one presented here. On the other hand, it uses only AC^0 instructions.)

The complexity of our algorithm is dependent on the relation between n, the number of keys, and w, the word size. The upper bound of $O(\sqrt{\log n})$ is independent of w; when taking both n and w into account the cost is $o(\sqrt{\log n})$ in many cases. As an important example we achieve a cost of $O(\log w\log\log n)$; this shows that the complexity of van Emde Boas trees [12, 13] can almost be matched in deterministic linear space. (Previously, the only matching deterministic data structure was static [14].)

We also make two minor observations on the complexity of perfect hashing and on adapting our data structure to the input distribution.

2 Summary of new results

2.1 Main result

Theorem 1 *On a unit-cost RAM with word size w, an ordered set of n w-bit keys (viewed as binary strings or integers) can be maintained in*

$$O\left(\min\left\{\begin{array}{l}\frac{\log n}{\log w}+\log\log n\\ \log\left\lceil\frac{\log n}{\log^2 w}+1\right\rceil\log w\end{array}\right\}\right)$$

$$=O\left(\min\left\{\begin{array}{l}\sqrt{\log n}\\ \frac{\log n}{\log w}+\log\log n\\ \log w\log\log n\end{array}\right\}\right)$$

time per operation, including insert, delete, member search, and neighbour search. The cost for searching is worst-case while the cost for updates is amortized. For range queries, there is an additional cost of reporting the found keys. The data structure uses linear space.

The instruction repertoire is the same as for fusion trees, i.e. it includes double-precision multiplication but not division.

In the following, we will not discuss the instruction set in detail. In one case we will show how to avoid the use of division. Since we borrow some techniques from fusion trees, we need double-precision multiplication. Apart from traditional arithmetic and bitwise boolean operations, we use the shift operation, which is not explicitly used in fusion trees. Shifting is a simple operation, it can also be simulated with double-precision multiplication (as done in fusion trees).

We note that there is no real need for a double-precision multiplication; we can simulate that operation with a number of single-precision multiplications. (Another possibility is to split each w-bit key into two $w/2$-bit keys and to store them in a trie of height 2 such that each node in the trie is implemented as in Theorem 1. Then, a "double-precision" multiplication would be performed by a single-precision multiplication.)

2.2 Related observations

In Section 4 we also discuss some related results, stated below.

We give the first deterministic polynomial-time (in n) algorithm for constructing a linear space static dictionary with $O(1)$ worst-case access cost (cf. perfect hashing).

Observation 1 *A linear space static data structure supporting member queries at a worst case cost of $O(1)$ can be constructed in $O\left(n^{2+\epsilon}\right)$ worst-case time. Both construction and searching can be done without division.*

We also observe how to adapt our data structure to a favourable input distribution. For a more detailed discussion of the interpretation of the observation below, we refer to Section 4.2.

Observation 2 *There exist a linear-space data structure for which the worst-case cost of a search and the amortized cost of an update is $O(\log b \log\log n)$ where $b \leq w$ is the length of the query key's distinguishing prefix, i.e. the number of bits that need to be inspected in order to distinguish it from each of the other stored keys.*

Finally we show that Theorem 1 can be generalized to the case when the keys are longer than one word each.

Theorem 2 *Suppose each key occupies L words. Then, there exists a data structure occupying $O(Ln)$ space for which the worst-case cost of a search and the amortized cost of an update is the same as in Theorem 1 plus an additional cost of $O(L \log\log n)$ per operation.*

3 Proof of Theorem 1

3.1 Exponential search trees

Our basic data structure is a multiway tree where the degrees of the nodes decrease exponentially down the tree. For related techniques, see the references [3, 6, 8, 9].

Lemma 1 *Suppose a static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(S(d))$ worst-case time. Then, there exists a dynamic data structure storing n keys with the following costs:*

- *it uses $O(n)$ space;*
- *it can be constructed from a sorted list in $O(n)$ worst-case time and space;*
- *the worst-case cost of searching (including neighbour search) satisfies*

$$T(n) = O\left(S\left(n^{1/5}\right)\right) + T\left(n^{4/5}\right);$$

- *the amortized cost of restructuring during updates (i.e. the amortized update cost when the cost of locating the update position is not counted) is $O(\log\log n)$.*

Proof: We use an *exponential search tree*. It has the following properties:

- Its root has degree $\Theta(n^{1/5})$.
- The keys of the root are stored in a local (static) data structure, with the properties stated above. During a search, the local data structure is used to determine in which subtree the search is to be continued.
- The subtrees are exponential search trees of size $\Theta(n^{4/5})$.

First, we show that, given n sorted keys, an exponential search tree can be constructed in linear time and space. Since the cost of constructing a node of degree d is $O\left(d^4\right)$, the total construction cost $C(n)$ is given by

$$C(n) = O\left(\left(n^{1/5}\right)^4\right) + n^{1/5} \cdot C\left(n^{4/5}\right)$$

$$\Rightarrow \quad C(n) = O(n).$$

Furthermore, with a similar equation, the space required by the data structure can be shown to be $O(n)$.

Next, we derive the search cost, $T(n)$. It follows immediately from the description of exponential search trees that

$$T(n) = O\left(S\left(n^{1/5}\right)\right) + T\left(n^{4/5}\right).$$

Finally, we analyze the cost of updates. We only consider the cost of restructuring and ignore the cost of finding the update position; the latter cost equals the cost of searching.

Balance is maintained in a standard fashion by global and partial rebuilding. Let n_0 denote the number of present elements at the last global rebuilding. The next global rebuilding will occur when $|n - n_0| \geq n_0/2$. Hence, the linear cost of a global rebuilding is amortized over a linear number of updates and the amortized cost is $O(1)$.

At a global rebuilding, we set $n_0 = n$ and the degree of the root is chosen as $\left\lceil n_0^{1/5} \right\rceil$, while the size of each subtree is $\frac{n_0}{\lceil n^{4/5} \rceil} \pm 1$. Between global rebuildings, we ensure that the subtrees have size at least $\frac{n_0}{2\lceil n^{4/5} \rceil} \pm 1$ and at most $\frac{2n_0}{\lceil n^{4/5} \rceil} \pm 1$. When an update causes a subtree to violate this condition, we examine the sum of the sizes of that subtree and one of its immediate neighbours. This sum is between $\frac{n_0}{\lceil n^{4/5} \rceil} \pm 1$ and $\frac{4n_0}{\lceil n^{4/5} \rceil} \pm 1$. By reconstructing these two subtrees into one, two, three, or four new subtrees we can ensure that the sizes of the new trees will be far from the limits. In this way, we guarantee that a linear—in the size of the subtree—number of updates is needed before a subtree is reconstructed. Since the cost of constructing a subtree is linear in the size of the subtree, the amortized cost of reconstructing subtrees is $O(1)$.

Each time some subtrees are reconstructed, the degree of the root will change and the root must be reconstructed. The cost of this reconstruction is $O((n^{1/5})^4)$. Again, this is linear in the size of a subtree; hence, the amortized cost of reconstructing the root is $O(1)$. This gives us the following equation for the amortized restructuring cost $R(n)$:

$$R(n) = O(1) + R\left(n^{4/5}\right) \Rightarrow R(n) = O(\log\log n)$$

□

3.2 An improvement of fusion trees

Using our terminology, the central part of the fusion tree is a static data structure with the following properties:

Lemma 2 *(Fredman and Willard) For any d, $d = O\left(w^{1/6}\right)$, A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(1)$ worst-case time.*

Fredman and Willard used this static data structure to implement a B-tree where only the upper levels in the tree contain B-tree nodes, all having the same degree (within a constant factor). At the lower levels, traditional (i.e. comparison-based) weight-balanced trees were used. The amortized cost of searches and updates is $O(\log n/\log d + \log d)$ for any $d = O\left(w^{1/6}\right)$. The first term corresponds to the number of B-tree levels and the second term corresponds to the height of the weight-balanced trees.

Using an exponential search tree instead of the Fredman/Willard structure, we avoid the need for weight-balanced trees at the bottom at the same time as we improve the complexity for large word sizes.

Lemma 3 *A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O\left(\frac{\log d}{\log w} + 1\right)$ worst-case time.*

Proof: We just construct a static B-tree where each node has the largest possible degree according to Lemma 2. That is, it has a degree of $\min\left(d, w^{1/6}\right)$. This tree satisfies the conditions of the lemma. □

Corollary 1 *There is a data structure occupying linear space for which the worst-case cost of a search and the amortized cost of an update is $O\left(\frac{\log n}{\log w} + \log\log n\right)$*

Proof: Let $T(n)$ be the worst-case search cost. Combining Lemmas 1 and 3 gives that

$$T(n) = O\left(\frac{\log n}{\log w} + 1 + T\left(n^{4/5}\right)\right).$$

□

3.3 Tries and perfect hashing

In a binary trie, a node at depth i corresponds to an i-bit prefix of one (or more) of the keys stored in the trie. Suppose we could access a node by its prefix in constant time by means of a hash table, i.e. without traversing the path down to the node. Then, we could find a key x, or x's nearest neighbour, in $O(\log w)$ time by a binary search for the node corresponding to x's longest matching prefix. At each step of the binary search, we look in the hash table for the node corresponding to a prefix of x; if the node is there we try with a longer prefix, otherwise we try with a shorter one.

The idea of a binary search for a matching prefix is the basic principle of the van Emde Boas tree [12, 13, 14]. However, a van Emde Boas tree is not just a plain binary trie represented as above. One problem is the space requirements; a plain binary trie storing d keys may contain as much as $\Theta(dw)$ nodes. In a van Emde Boas tree, the number of nodes is decreased to $O(d)$ by careful optimization.

In our application $\Theta(dw)$ nodes can be allowed. Therefore, to keep things simple, we use a plain binary trie.

Lemma 4 *A static data structure containing d keys and supporting neighbour queries in $O(\log w)$ worst-case time can be constructed in $O\left(d^4\right)$ time and space.*

Proof: We study two cases.

Case 1: $w > d^{1/3}$. Lemma 3 gives constant query cost.

Case 2: $w \leq d^{1/3}$. In $O(dw) = o(d^2)$ time and space we construct a binary trie of height w containing all d keys. Each key is stored at the bottom of a path of length w and the keys are linked together. In order to support neighbour queries, each unary node contains a neighbour pointer to the next (or previous) leaf according to the inorder traversal.

To allow fast access to an arbitrary node, we store all nodes in a perfect hash table such that each node of depth i is represented by the i bits on the path down to the node. Since the paths are of different length, we use w hash tables, one for each path length. Each hash table contains at most d nodes. The algorithm by Fredman, Komlos, and Szemeredi [5] constructs a hash table of d keys in $O(d^3 w)$ time. The algorithm uses division, this can be avoided by simulating each division in $O(w)$ time. With this extra cost, and since we use w tables, the total construction time is $O\left(d^3 w^3\right) = O(d^4)$ while the space is $O(dw) = o(d^2)$.

With this data structure, we can search for a key x in $O(\log w)$ time by a binary search for the node corresponding to x's longest matching prefix. This search either ends at the bottom of the trie or at a unary node, from which we find the closest neighbouring leaf by following the node's neighbour pointer.

During a search, evaluation of the hash function requires integer division. However, as pointed out by Knuth [7], division with some precomputed constant p may essentially be replaced by multiplication with $1/p$. Having computed $r = \lfloor 2^w/p \rfloor$ once in $O(w)$ time, we can compute x DIV p as $\lfloor xr/2^w \rfloor$ where the last division is just a right shift w positions. Since $\lfloor x/p \rfloor - 1 < \lfloor xr/2^w \rfloor \leq \lfloor x/p \rfloor$ we can compute the correct value of x DIV p by an additional test. Once we can compute DIV, we can also compute MOD. □

An alternative method for perfect hashing without division is the one recently developed by Raman [10]. Not only does this algorithm avoid division, it is also asymptotically faster, $O(d^2 w)$.

Corollary 2 *There is a data structure occupying linear space for which the worst-case cost of a search and the amortized cost of an update is $O(\log w \log\log n)$.*

Proof: Let $T(n)$ be the worst-case search cost. Combining Lemmas 1 and 4 gives $T(n) = O(\log w) + T\left(n^{4/5}\right)$. □

3.4 Finishing the proof

If we combine Lemmas 1, 3, and 4, we obtain the following equation for the cost $T(n)$ of a search or update in an exponential search tree.

$$T(n) = O\left(\min\left(1 + \frac{\log n}{\log w}, \log w\right)\right) + T\left(n^{4/5}\right) \quad (1)$$

Ignoring the right part of the min expression gives Corollary 1 and ignoring the left part gives Corollary 2. Balancing the two parts of the min-expression gives

$$T(n) = O\left(\sqrt{\log n}\right) + T\left(n^{4/5}\right)$$

and hence

$$T(n) = O\left(\sqrt{\log n}\right).$$

Combining these three complexities yields

$$T(n) = O\left(\min\left\{\begin{array}{l}\sqrt{\log n}\\ \frac{\log n}{\log w} + \log\log n\\ \log w \log\log n\end{array}\right\}\right).$$

This expression is not tight for all possible combinations of n and w. To provide a tight asymptotic expression for the solution to Equation 1, we note that the parameter n will decrease as the recursion progresses (i.e. the size of subtrees decrease as we progress down the tree) while the word size w remains the same. Hence, the right part of our min expression will be used at some (maybe none) of the upper levels; at the lower levels the left part will be applied. Therefore, we distinguish two cases:

Case 1: $\log n \leq 2\log^2 w$. The left part will apply on all levels, giving a total cost of $O\left(\frac{\log n}{\log w} + \log\log n\right)$.

Case 2: $\log n > 2\log^2 w$. The right part will apply $O\left(\log\left\lceil\frac{\log n}{\log^2 w}\right\rceil\right)$ times, resulting in a cost of $O\left(\log\left\lceil\frac{\log n}{\log^2 w}\right\rceil \log w\right)$. Then, the left part will give a cost of $O\left(\sqrt{\log w} + \log\log n\right)$. In this case, the total cost will be (note that $\log n > 2\log^2 w$)

$$\begin{aligned} & O\left(\log\left\lceil\frac{\log n}{\log^2 w}\right\rceil \log w + \sqrt{\log w} + \log\log n\right) \\ = \quad & O\left(\log\left\lceil\frac{\log n}{\log^2 w} + 1\right\rceil \log w\right). \end{aligned}$$

Combining the two cases gives the following complexity:

$$T(n) = \begin{cases} O\left(\frac{\log n}{\log w} + \log\log n\right), & \log n \leq 2\log^2 w \\ O\left(\log\left\lceil\frac{\log n}{2\log^2 w} + 1\right\rceil \log w\right), & \log n > 2\log^2 w \end{cases}$$

The two expressions meet when $\log n = \Theta\left(\log^2 w\right)$. Thus, we can combine them as

$$T(n) = O\left(\min\left\{\begin{array}{l}\frac{\log n}{\log w} + \log\log n \\ \log\left\lceil\frac{\log n}{\log^2 w} + 1\right\rceil \log w\end{array}\right\}\right).$$

The proof of Theorem 1 is now complete.

4 Some related observations

We discuss the additional results stated in Section 2.2.

4.1 Observation 1: On perfect hashing

As mentioned earlier, a perfect hash table that supports member queries (neighbour queries are not supported) in constant time can be constructed at a worst-case cost $O\left(n^2 w\right)$ without division [10]. We show that the dependency of word size can be removed. The resulting data structure is not a pure "hash table".

We start by taking a closer look at the fusion tree. According to Fredman and Willard, a fusion tree node of degree d requires $\Theta\left(d^2\right)$ space. This space is occupied by a lookup table where each entry contains a rank between 0 and d. A space of $\Theta\left(d^2\right)$ is enough for the original fusion tree as well as for our exponential search tree. However, for the purpose of perfect hashing, we need to reduce the space. Fortunately, this reduction is straightforward. We note that a number between 0 and d can be stored in $\log d$ bits. Thus, since $d < w^{1/6}$, the total number of bits occupied by the lookup table is $O\left(d^2 \log d\right) = O(w)$. We conclude that instead of $\Theta\left(d^2\right)$, the space taken by the table is $O(1)$ ($O(d)$ would have been good enough). Therefore, the space occupied by a fusion tree node can be made linear in its degree.

We are now ready to prove Observation 1, stated in Section 2.2.

Proof of Observation 1: W.l.o.g we assume that $\epsilon < 1/6$.

Since Raman has shown that a perfect hash function can be constructed in $O\left(n^2 w\right)$ time without division) [10], we are done for $n \geq w^{1/\epsilon}$.

If, on the other hand, $n < w^{1/\epsilon}$, we construct a static fusion tree with degree $n^{1/3}$. This degree is possible since $\epsilon < 1/6$. The height of this tree is $O(1)$, the cost of constructing a node is $O\left(n^{4/3}\right)$ and the total number of nodes is $O\left(n^{2/3}\right)$. Thus, the total construction cost for the tree is $O\left(n^2\right)$. □

4.2 Observation 2: An adaptive data structure

In some applications, we may assume that the input distribution is favourable. These kind of assumptions may lead to a number of heuristic algorithms and data structures whose analysis are based on probabilistic methods. Typically, the input keys may be assumed to be generated as independent stochastic variables from some (known or unknown) distribution; the goal is to find an algorithm with a good expected behaviour. For these purposes, a deterministic algorithm is not needed.

However, instead of modeling input as the result of a stochastic process, we may characterize its properties in terms of a *measure*. Attention is then moved from the process of generating data to the properties of the data itself. In this context, it makes sense to use a deterministic algorithm; given the value of a certain measure the algorithm has a guaranteed cost.

We give one example of how to adapt our data structure according to a natural measure. An indication of how "hard" it is to search for a key is how large part of it must be read in order to distinguish it from the other keys. We say that this part is the key's *distinguishing prefix*. (In Section 3.3 we used the term longest matching prefix for essentially the same entity.) For w-bit keys, the longest possible distinguishing prefix is of length w. Typically, if the input is nicely distributed, the average length of the distinguishing prefixes is $O(\log n)$.

As stated in Observation 2, we can search faster when a key has a short distinguishing prefix.

Proof of Observation 2: We use exactly the same data structure as in Corollary 2, with the same restructuring cost of $O(\log\log n)$ per update. The only difference is that we change the search algorithm from the proof of Lemma 4. Applying an idea of Chen and Reif [4], we replace the binary search for the longest matching (distinguishing) prefix by an exponential-and-binary search. Then, at each node in the exponential search tree, the search cost will decrease from $O(\log w)$ to $O(\log b)$ for a key with a distinguishing prefix of length b. □

4.3 Theorem 2: Sorting and searching long keys

So far, we have discussed how to treat w-bit keys on a w-bit RAM. However, in some application the input may consist of multi-word keys. As a further application of the exponential search tree, we study the case when each key occupies L words.

Lemma 5 *Suppose each key occupies L words. Furthermore, suppose a static data structure containing d keys can be constructed in $O\left(dL + d^4\right)$ time and space such that it supports neighbour queries in $O(L+S(d))$ worst-case time. Then, there exists a dynamic data structure storing n keys with the following costs:*

- *it uses $O(nL)$ space;*
- *it can be constructed from a sorted list in $O(nL)$ worst-case time and space;*
- *the worst-case cost of searching for a key (including neighbour search) satisfies*

$$T(n) = O\left(L + S\left(n^{1/5}\right)\right) + T\left(n^{4/5}\right);$$

- *the amortized cost of restructuring during updates (i.e. the amortized update cost when the cost of locating the update position is not counted) is $O(L + \log\log n)$.*

Proof: (sketch) We alter the proof of Lemma 1 slightly. □

Lemma 6 *Suppose each key occupies L words. Then, a static data structure containing d keys and supporting neighbour queries in $O\left(L + \min\left(1 + \frac{\log n}{\log w},\ \log w\right)\right)$ can be constructed in $O(dL + d^4)$ time and space.*

Proof: We store the d keys in a trie of height L. In the trie, each unary node is stored in $O(1)$ space in the obvious way. For each non-unary node, we represent the set of outgoing edges in two ways. First, they are stored with constant lookup time according to Observation 1. Second, they are stored in a data structure supporting neighbour search according to Lemma 3 or Lemma 4. The sum of the degrees of the nun-unary nodes is at most $2d - 1$ and the number of unary nodes is less than dL. Hence, it follows from Lemmas 3 and 4 and Observation 1 that the trie can be constructed in $O(dL + d^4)$ time and space.

When searching for a key we traverse the trie; at each node we find the proper outgoing edge in constant time. The traversal either ends successfully at the bottom of the trie or at a missing edge. If the missing edge is at a unary node, we can determine the nearest neighbour immediately, otherwise we find the neighbour by making a local search among the outgoing edges of the last found node.

The cost of traversing the trie is $O(L)$ and the cost of a local search is $O\left(\min\left(1 + \frac{\log n}{\log w},\ \log w\right)\right)$ according to Lemmas 3 and 4. □

Proof of Theorem 2: (cf. Section 2.2) Combine Lemmas 5 and 6 and proceed as in the proof of Theorem 1. □

Corollary 3 *n multi-precision integers, each of length L words, can be sorted in $O(n\sqrt{\log n} + Ln\log\log n)$ time.*

When using a comparison-based sorting algorithm, each comparison would take $\Theta(L)$ time and the cost of sorting would be $\Theta(Ln\log n)$. Hence, for multi-word keys the asymptotic improvement over comparison-based algorithms is even larger than for single-word keys.

5 Comments

Sorting and searching in deterministic linear space is certainly a fundamental topic and we believe that many researchers have tried to find new improved bounds since the fusion trees were first presented.

For small (realistic) word sizes the bound of $O(\log w \log\log n)$ is appealing; this shows that the complexity of van Emde Boas trees can almost be matched in deterministic linear space.

It is interesting to note that our main result relies on "old" techniques, such as fusion trees, tries, and perfect hashing. None of the new techniques developed recently [1, 2, 11] are used.

Acknowledgements

The author would like to thank Bengt Nilsson for insightful comments on the manuscript and Mikkel Thorup and Rajeev Raman for their valuable comments, in particular for discussions on how to implement perfect hashing without division.

References

[1] A. Andersson. Sublogarithmic searching without multiplications. In *Proc.* 36^{th} *IEEE Symposium on Foundations of Computer Science*, pages 655–663. ACM Press, 1995.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings* 27^{th} *ACM Symposium on Theory of Computing*, pages 427–436. ACM Press, 1995.

[3] A. Andersson and C. Mattsson. Dynamic interpolation search in $o(\log \log n)$ time. In *Proc.* 20^{th} *International Colloquium on Automata, Languages and Programming*. Springer Verlag, 1993.

[4] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proceedings of the* 34^{th} *Annual IEEE Symposium on Foundations of Computer Science*, pages 104–112, 1993.

[5] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[6] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1994.

[7] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973. ISBN 0-201-03803-X.

[8] K. Mehlhorn and A. Tsakalidis. Dynamic interpolation search. *Journal of the ACM*, 49(3):621–634, 1993.

[9] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, 1983. ISBN 3-540-12330-X.

[10] R. Raman. Priority queues: small, monotone and trans-dichotomous. to appear in proc. European Symp. on Algorithms, 1996.

[11] M. Thorup. On RAM priority queues. In *Proc.* 7^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[12] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[13] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

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