

Faster deterministic sorting and searching in linear space (preliminary version)

Tech. report LU-CS-TR:95-160

Arne Andersson
Department of Computer Science
Lund University
Box 118, S-22100 Lund, Sweden
`arne@dna.lth.se`

December 15, 1995. Slightly modified December 18, 1995

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

$$O\left(\min\left(\sqrt{\log n}, \frac{\log n}{\log w} + \log \log n, \log w \log \log n\right)\right)$$

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. As an application, n keys can be sorted in linear space at a worst-case cost of $O(n\sqrt{\log n})$.

The best previous method for deterministic sorting and searching in linear space has been the fusion trees which supports 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, 9, 10]. 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 $O(\sqrt{\log n})$ 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 $O(n\sqrt{\log n})$.

The exact complexity of our algorithm is depending on the relation between n , the number of keys, and w , the word size. In many cases it is $o(\sqrt{\log n})$.

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

2 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*

$$O\left(\min\left(\frac{\log n}{\log w} + \log \log n, \quad \log \left\lceil \frac{\log n}{2(\log w)^2} + 1 \right\rceil \log w\right)\right)$$

$$\left[= O\left(\min\left(\sqrt{\log n}, \quad \frac{\log n}{\log w} + \log \log n, \quad \log w \log \log n\right)\right)\right]$$

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).

3 Proof

3.1 Exponential search trees

Our basic data structure is a multiway tree where the degrees of the nodes decrease exponentially down the tree. In order to support fast searches, some auxiliary information is stored in each node. For related techniques, see the references [3, 6, 7, 8].

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

- *it uses $O(n)$ space;*
- *it can be constructed 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 $\Theta\left(n^{1/5}\right)$.

- The keys of the root are stored in a local data structure. 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 $\Theta\left(n^{4/5}\right)$.

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 $O(d^4)$, the total construction cost $C(n)$ is given by

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

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

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

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 $\lceil n_0^{1/5} \rceil$, 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 contents of the root will change and so the root must be reconstructed. The cost of this reconstruction is $O\left(\left(n^{1/5}\right)^4\right)$. 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 $R(n)$:

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

□

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 , $d = O(w^{1/6})$, A static data structure containing d keys can be constructed in $O(d^4)$ 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 $d = O(w^{1/6})$. 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 $O(d^4)$ 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 $\min(d, w^{1/6})$. This tree satisfies the conditions of the lemma. \square

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

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

\square

3.3 van Emde Boas trees and perfect hashing

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

Proof: We study two cases.

Case 1: $\log w > \sqrt{\log d}$. The lemma follows immediately from lemma 3.

Case 2: $\log w \leq \sqrt{\log d}$. In this case, we use a variant of the van Emde Boas tree (VEB). A VEB is a recursive trie of height $\Theta(\log w)$. During a search (or update) we follow a

trie edge in constant time by using a part of the search key as address in a large array of outgoing edges. Normally, a VEB requires a considerable amount of space due to the large arrays used. However, if these arrays are represented as a perfect hash table, the total space occupied by the VEB is proportional to the total number of nodes, which is $O(d)$ [11, 12, 13]. (In the earliest version of the VEB the number of nodes is $\Theta(d \log w)$, which is good enough for our application.)

When the VEB is under construction, we can not afford to store it with perfect hashing. Instead, we construct a temporary pointer-based version of the VEB where each array of edges is replaced by a binary search tree. Here, the cost of following a trie edge is $O(\log d)$ and the cost of inserting a key is $O(\log d \log w)$. Hence the total cost of constructing the pointer-based VEB is $O(d \log d \log w)$.

After the pointer-based VEB has been constructed, we compute a perfect hash table for all $\Theta(d)$ trie edges. The method by Fredman, Komlos, and Szemerédi is used [5]. In its original form, the cost of this algorithm is $O(d^3 w)$ and it uses division. Since we wish to avoid division, we simulate each division in $O(w)$ time. With this extra cost the hash table can be constructed in $O(d^3 w^2)$ time. Also, we precompute constants that allows us to evaluate the hash function without division. In order to compute $x \bmod p$ for some precomputed value p we do the following preprocessing: In $O(w)$ time we compute another constant $r = 2^w \text{ DIV } p$. We can now compute $x \text{ DIV } p$ as $rx \text{ DIV } 2^w$ where the last division is just a right shift w positions (if we wish, this shift can be simulated by a double-precision multiplication). 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 [9]. Not only does this algorithm avoid division, it is also asymptotically faster, $O(d^2 w)$.

In total, the construction cost is $O(d \log d \log w + d^3 w^2)$. Since $w \leq 2^{\sqrt{\log d}}$ this cost is $O(d^4)$. \square

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*

$$T(n) = O(\log w \log \log n).$$

Proof: Combining Lemmas 1 and 4 gives

$$T(n) = O(\log w) + T(n^{4/5}) \quad \Rightarrow \quad T(n) = O(\log w \log \log n).$$

\square

3.4 Putting it together

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.

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

Ignoring the right part of the min expression gives Corollary 1 and ignoring the left part gives Corollary 2. Also, noting that the min expression is $O(\sqrt{\log n})$, we can derive that

$$T(n) = O\left(\sqrt{\log n} + \sqrt{\log(n^{4/5})} + \sqrt{\log(n^{16/25})} + \dots\right) = O(\sqrt{\log n}).$$

Combining these three complexities yields

$$O\left(\min\left(\sqrt{\log n}, \quad \frac{\log n}{\log w} + \log \log n, \quad \log w \log \log n\right)\right).$$

However, 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 sizes 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: $\log n \leq 2(\log w)^2$. 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: $\log n > 2(\log w)^2$. The right part will apply $O\left(\log\left[\frac{\log n}{2(\log w)^2}\right]\right)$ times, resulting in a cost of $O\left(\log\left[\frac{\log n}{2(\log w)^2}\right] \log w\right)$. Then, the left part will give a cost of $O(\sqrt{\log w} + \log \log n)$. In this case, the total cost will be

$$O\left(\log\left[\frac{\log n}{2(\log w)^2}\right] \log w + \sqrt{\log w} + \log \log n\right) = O\left(\log\left[\frac{\log n}{2(\log w)^2} + 1\right] \log w\right)$$

since $\log n > 2(\log w)^2$. Combining the two cases give us the following complexity:

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

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

$$O\left(\min\left(\frac{\log n}{\log w} + \log \log n, \quad \log\left[\frac{\log n}{2(\log w)^2} + 1\right] \log w\right)\right).$$

The proof of Theorem 1 is now complete.

4 Two observations

4.1 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*. 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 $O(\log n)$.

Our combination of van Emde Boas trees and perfect hashing can be modified to adapt nicely to the input distribution. Normally, searching in a VEB corresponds to locating the closest existing prefix of the query key by making a binary search on the length of the distinguishing prefix. Since the word length is w , the cost of a binary search is $O(\log w)$. As observed by Chen and Reif [4], the binary search may be replaced by an exponential-and-binary search. The corresponding data structure is similar to the VEB, but the search cost is $O(\log b)$ where b is the length of the distinguishing prefix. Using this data structure in the nodes of an exponential search tree, we obtain a linear-space data structure that allows searching in $O(\log b \log \log n)$ worst-case time for a key with a distinguishing prefix of length b .

This yields:

Observation 1 *There exist a linear-space data structure supporting search at a worst-case cost of $O(\log b \log \log n)$ where b 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.*

4.2 An observation 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 $O(n^2 w)$ without division [9]. We show that the dependency of word size can be removed. First, we need to take a closer look at the space requirements of fusion trees. According to Fredman and Willard, a fusion tree node of degree d requires $\Theta(d^2)$ space. This space is occupied by a lookup table where each entry contains a rank between 0 and d . A space of $\Theta(d^2)$ 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 is done just by taking a closer look at the Fredman/Willard construction. 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 $O(d^2 \log d) = O(w)$. We conclude that instead of $\Theta(d^2)$, 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 is linear in its degree.

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

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

Since Raman has shown that a perfect hash function can be constructed in $O(n^2w)$ time without division) [9], 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 $O(1)$ and the total number of nodes is $O(n^{2/3})$. Thus, the total construction cost for the tree is $O(n^2)$. \square

Acknowledgements

The author would like to thank 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. 36th IEEE FOCS*, pages 655–663, 1995.
- [2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings 27th ACM Symposium on Theory of Computing*, pages 427–436. ACM Press, 1995.
- [3] A. Andersson and Ch. Mattsson. Dynamic interpolation search in $o(\log \log n)$ time. In *Proc. ICALP '93, Springer Verlag*, 1993.
- [4] Shenfeng Chen and John H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 104–112, 1993.
- [5] M. L. Fredman, J. Komlos, 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] K. Mehlhorn and A. Tsakalidis. Dynamic interpolation search. *Journal of the ACM*, 49(3):621–634, 1993.
- [8] 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.

- [9] R. Raman. Improved data structures for predecessor queries in integer sets. manuscript, 1995.
- [10] M. Thorup. On RAM priority queues. to appear in proc. SODA, 1996.
- [11] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.
- [12] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.
- [13] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17:81–84, 1983.