Faster deterministic sorting and priority queues in linear space*

Mikkel Thorup
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk, http://www.diku.dk/~mthorup

Abstract

The RAM complexity of deterministic linear space sorting of integers in words is improved from $O(n\sqrt{\log n})$ to $O(n(\log\log n)^2)$. No better bounds are known for polynomial space. In fact, the techniques give a deterministic linear space priority queue supporting insert and delete in $O((\log\log n)^2)$ amortized time and find-min in constant time. The priority queue can be implemented using addition, shift, and bit-wise boolean operations.

1 Introduction

In this paper, we consider the problem of RAM sorting of integer keys, each stored in one word (briefly, a RAM is what we program in imperative programming languages such as C. The memory is divided into addressable words of length w. Addresses are themselves contained in words, so $w \geq \log n$. Moreover, we have a constant number of registers, each with capacity for one word. The basic assembler instructions are: conditional jumps, direct and indirect addressing for loading and storing words in registers, and some computational instructions, such as comparisons and addition, for manipulating words in registers. The space complexity is the maximal memory address used, and the time complexity is the number of instructions performed). Ever since Fredman and Willard [FW93] in 1990 surpassed the information theoretic sorting lower bound with fusion trees, there has been a substantial gap between deterministic and randomized linear space sorting. Fusion trees gave a deterministic bound of $O(n\log n/\log\log n)$ and a randomized bound of $O(n\sqrt{\log n})$. In 1995, Andersson et.al.[AHNR95] improved the complexity of randomized linear space sorting to $O(n\log\log n)$. In 1996, Raman improved the complexity of deterministic linear space sorting to $O(n\sqrt{\log n\log\log n})$ and, also in 1996, this was improved by Andersson [And96] to $O(n\sqrt{\log n})$. In this paper, we improve the complexity of deterministic linear space sorting to $O(n(\log\log n)^2)$. In fact we prove something stronger:

Theorem 1 *There is a linear space priority queue which on n keys supports insert and delete in $O((\log\log n)^2)$ amortized time and find-min in constant time. It may be implemented using addition, shift, and bit-wise boolean operations as only computational instructions.*

Note that the above mentioned instructions are all in AC^0, meaning that they can be implemented by polynomially sized constant depth circuits. The constant depth justifies that we pay only constant time per instruction. To compare with the previously mentioned techniques, those from [FW93] and [And96] are general dynamic search structures. They inherently use multiplication which is not in AC^0. The technique from [AHNR95] is only for sorting, but it is generalized to an $O(\log\log n)$ randomized linear space priority queue in [Tho96]. The techniques from [AHNR95, Tho96] use multiplicative hashing to achieve linear space. An implementation based on addition,

*Part of this research was done while the author visited the Max-Plank-Intitut für Informatik.

shift, and bit-wise boolean operations is, however, presented in [Tho97]. The construction from [Ram96] is a monotone priority queue, meaning a priority queue where the minimum has to be non-decreasing, and it is implemented using addition, shift, and bit-wise boolean operations.

It should be noted that Miltersen [Mil94] has proved that dynamic searching takes $\Omega(\sqrt[3]{\log n})$ time even if we allow for randomization. Hence our $O((\log\log n)^2)$ bound cannot be generalized to dynamic searching.

Techniques

Our techniques are strongly related to Andersson's [And96] exponential search trees giving the previous fastest deterministic linear space sorting and searching. An exponential search tree is a general method, transforming a polynomial time and space construction of a fast static search structure into a fast dynamic linear space search structure. Here, by *searching* a key y in a set X, we mean finding a pointer to the maximum key $x \in X$, $x \le y$; if $y < \min X$, a null-pointer is returned. From [And96], we have

Proposition 2 (Andersson) *Given d keys, suppose a static search structure on d keys can be constructed in $O(d^{k-1})$, $k \ge 2$, time and space so that it supports searches in $O(S(d))$ time. We can then construct a dynamic linear space search structure that with n keys supports insert, delete, and searches in amortized time $T(n)$ where $T(n) \le T(O(n^{1-1/k})) + O(S(O(n^{1/k}))$.*

In order to use the proposition, Andersson derandomizes the randomized $O(\sqrt{\log n})$ time and linear space searching technique of Fredman and Willard [FW93].

Proposition 3 (Andersson) *Given n keys, in $n^{O(1)}$ time and space, a static deterministic $O(\sqrt{\log n})$-time search structure can be constructed.*

Corollary 4 (Andersson) *There is a dynamic deterministic search structure supporting insert, delete, and searches in amortized time $O(\sqrt{\log n})$*

As mentioned above, from [Mil94], we have a lower-bound of $\Omega(\sqrt[3]{\log n})$ for Corollary 4, and hence for Proposition 3. In this paper, however, we show that an exponential speed-up per search is to be gained if instead of doing one search at the time, we process a batch of n searches. More precisely, we get the following alternative to Proposition 3,

Proposition 5 *Given a set X of n keys, a static data structure can be constructed in $n^{O(1)}$ time and space so that any batch of n searches in X can be performed in time $O(n \log\log n)$.*

Proposition 5 is complemented by showing

Proposition 6 *Given d keys, suppose a static structure can be constructed in $O(d^{k-1})$, $k > 2$, time and space, supporting any batch of d searches in time $O(d \cdot S(d))$. We can then construct a dynamic priority queue that with n keys supports find-min in constant time and insert and delete in amortized time $T(n)$ where $T(n) \le T(O(n^{1-1/k})) + O(S(O(n^{1/k}))$.*

Corollary 7 *There is a dynamic deterministic linear space priority queue supporting find-min in constant time and insert and delete in amortized time $O((\log\log n)^2)$*

Finally we combine with ideas from [Tho97] to implement everything using addition, shifts, and bit-wise boolean operations, thus completing the proof of Theorem 1. It is worth mentioning that in contrast to Andersson's technique [And96], we completely avoid the use of Fredman and Willard's fusion trees [FW93] that are rather complicated and can only be implemented for $n > 2^{2^{20}}$.

The paper is divided as follows. We prove first Proposition 6, then Proposition 5, and finally we consider the implementation using addition, shifts, and bit-wise boolean operations.

2 Proof of Proposition 6

In this section, we prove the statement of Proposition 6:

> Given d keys, suppose a static structure can be constructed in $O(d^{k-1})$, $k > 2$, time and space, supporting any batch of d searches in time $O(d \cdot S(d))$. We can then construct a dynamic priority queue that with n keys supports insert, delete, and find-min in amortized time $T(n)$ where $T(n) \leq T(O(n^{1-1/k})) + O(S(O(n^{1/k}))$.

As in [And96], we define an *exponential search tree* T over n keys as a leaf oriented search tree (all keys of the search tree are in the leaves and an internal node v with $d(v)$ children contains $d-1$ key values separating the keys in the different subtrees) where the the root has $\Theta(n^{1/k})$ children and where each subtree is an exponential subtree with $\Theta(n^{1-1/k})$ keys. Consequently, the depth of T is $O(\log\log n)$. We say T is *balanced* if the root has $(1 \pm o(1))(n^{1/k})$ children and the subtrees are balanced, each with $(1 \pm o(1))(n^{1-1/k})$ keys.

The static data structure from the condition of Proposition 6 is referred to as an *S-structure.* As in [And96], for each node v in T, we have an S-structure over the $d(v)-1$ key values separating the keys of the subtrees of v. In [And96], the S-structure allows search for *one* key in time $S(d(v))$, giving the recursive search time of $T(n) \leq T(n^{1-1/k}) + O(S(O(n^{1/k}))$ from Proposition 2. Our S-structures are faster, but they only allow for searching *many* keys at the time. We therefore have to employ some buffers for the searching. By careful management of these search buffers, we achieve the above recurrence for priority queues.

Besides their doubly logarithmic depth, exponential search trees have another essential feature capturing why we can spend polynomial time and space on building our static S-structures:

Lemma 8 (Andersson) *If T is an exponential search tree with n keys,*

$$\sum_{v \in T} d(v)^{k-1} = O(n).$$

Here $d(v)$ denotes the number of children of v.

Given a sorted list of keys it is trivial to build a balanced exponential search tree without S-structures in linear time. Afterwards, by the definition of S-structures, for each node v, the S-structure at v is built in time $O(d(v)^{k-1})$. Consequently, by Lemma 8,

Corollary 9 (Andersson) *Given a sorted list of n keys, a balanced exponential search tree, including S-structures at all nodes, can be constructed in linear time.*

Corollary 9 is used by [And96] for local rebalancing in linear time. Consequently, the cost of updates become that of searching plus the depth of T which is $O(\log\log n)$.

For all nodes $v \in T$ that are not on the left-most branch, we introduce a *buffer* $B(v)$ for up to $d(v)$ keys. All keys in $B(v)$ have to belong below v in T. Since $|B(w)| \leq d(w)$ for all w, the number of keys in the buffers is bounded by the number of nodes in T, which is less than twice the number of leaves. Thus, with n keys placed in the leaves of T, we can have $< 2n$ keys in the buffers. Consequently, the buffers do not asymptotically affect the number of keys in a subtree. In the following, for a node v, by $n(v)$ we denote the number of keys below v in T including those in the buffers at v and below. Then our definition of exponential search trees states,

$$d(v) = \Theta(n(v)^{1/k}) \text{ and if } u \text{ is the parent of } v,\ n(v) = \Theta(n(u)^{1-1/k}) \tag{1}$$

Note that since we have no buffers on the left-most branch, the smallest key is always the left-most leaf of T. Despite the buffers being unsorted, we have

Lemma 10 *Given an exponential search tree T with buffers B, we can sort all the keys in linear time.*

Proof: In a bottom-up traversal of T, for each node v, we will construct a standard balanced search tree [Tar83, §4] over all the keys below v, including those in the buffers. Suppose this has been done for all children v. Then, first we join the at most $d(v)$ search trees of the children. Second we insert the at most $d(v)$ keys from $B(v)$. The cost of processing v is hence $O(d(v) \log n(v))$. By (1), $d(v) = \Theta(n(v)^{1/k})$, so $\log n(v) = O(\log d(v))$. Thus, since $k > 2$, the cost of processing v is $O(d(v) \log d(v)) = O(d(v)^{k-1})$. From Lemma 8, it now follows that the total cost of the processing all nodes is linear. At the end, we have a balanced binary search tree with all keys, readily giving us the desired sorted list. ∎

In order to implement insert and extract-min, temporarily, we will allow buffers and subtrees to deviate from the previously mentioned bounds. The operations are then implemented as follows.

find-min Return a pointer to the left-most leaf.

insert(x) It takes one constant time comparison to decide if x belongs in the left-most subtree of a node. Hence, in $O(\log \log n)$ time, we can send x down the left-most branch until we reach a node v for which x does not belong in the left-most branch. Then x is put in $B(v)$ in constant time.

extract-min Return and remove the left-most leaf.

delete Deleted keys are just marked. Hence, whenever a marked key becomes minimum, we perform an internal extract-min. If more than half the keys are marked, we rebuild the whole structure. Thus, the amortized cost of delete is dominated by the cost of the other operations.

Before carrying out any more operations, we may have to update our buffered exponential search tree by carefully applying the following operations:

clean(v) Nodes are *cleaned* regularly. A node v is *dirty* if $n(v)$ has doubled or halved since last v was cleaned. We then sort all keys below v, including those in the buffers. If $n(v)$ was halved, v is on the left-most path, and then we also sort the keys of its neighboring subtree to the right. Let u be the parent of v. The sorted list is now divided into segments of sizes between $0.75\, n(u)^{1-1/k}$ and $1.5\, n(u)^{1-1/k}$, each of which is turned into a balanced exponential search tree. All nodes of these balanced subtrees are now clean. The subtrees are new subtrees of u so the S-structure of u has to be updated.

flush(v) A buffer $B(v)$ is *over-full* if $|B(v)| > d(v)$, and we then take exactly $d(v)$ keys from $B(v)$ and send through the S-structure of v in time $d(v) \cdot S(d(v))$. The keys are then placed in the root buffers of the relevant subtrees.

Unfortunately, the complexities of the two operations depend on each other: over full buffers may make it impossible to sort a subtree in linear time, and if there are dirty nodes around, we cannot be sure that the sizes of the different trees relate properly, and hence both flushing and sorting of subtrees may take longer time. To avoid chaos, we have to observe the following two rules: 1) we only flush if there are no dirty nodes, and 2) we only flush a buffer $B(v)$ if there is no over full buffers below it.

Lemma 11 *(1) is always satisfied for all nodes.*

Proof: After a node v has been cleaned, (1) is satisfied with the hidden constant between 0.75 and 1.5, and hence (1) remains satisfied as long as no nodes are dirty. If a node v gets dirty because of an insert or extract-min, the changes in the involved values are at most by one, so (1) is still satisfied. If v gets dirty because its parent buffer $B(u)$ is flushed, the change in $n(v)$ may be non-constant. The number of elements flushed from $B(u)$ is $d(u)$, which is hence an upper-bound on the number of elements arriving at $B(v)$. Before the flushing of $B(u)$, by rule 1, all nodes were non-dirty. Hence (1) was everywhere satisfied, so

$$d(u) = \Theta(n(u)^{1/k}) = \Theta(n(v)^{1/(1-1/k)})^{1/k}) = \Theta((n(v)^{1/(k-1)}).$$

Since $k > 2$, we conclude that $n(v)$ is not affected asymptotically, hence that (1) remains satisfied despite a dirty flush. ∎

Lemma 12 *Cleaning a dirty node v takes time $O(n(v))$.*

Proof: Suppose v is dirty because of a flush from its parents buffer $B(u)$. By rule 2, no buffers below u were over-full. Hence, if we for a moment ignore the keys flushed down, by Lemma 10, we can sort all keys below v in time $O(n(v))$.

As we saw in the proof of Lemma 11, the number of keys flushed to $B(v)$ is at most $d(u) = \Theta((n(v)^{1/(k-1)})$. Since $k > 2$, these keys can be sorted in $O((n(v)^{1/(k-1)} \log n(v)) = o(n(v))$. Finally, they can be merged with the other sorted list in linear time. Thus we can sort all keys below v, including the last arrivals, in $O(n(v))$ time.

Since we have now dealt with the buffers, the remainder of the proof follows [And96]. By Corollary 9, an empty-buffered balanced exponential search tree is built from the sorted list in time $O(n(v))$. Finally, we have to rebuild the S-structure at u. Since (1) is satisfied, as above, $d(u) = \Theta((n(v)^{1/(k-1)})$, and hence, building the S-structure takes time $O((n(v)^{1/(k-1)})^{k-1}) = O(n(v))$.

The case of a dirty insertion is just simpler than flushing since only one key arrives. Dirtiness due to extract-min is the case of halving. Here we just need to note that by (1), the neighboring subtree is of size $O(n(v))$. ∎

From Lemma 12, it follows that the total cleaning cost is proportional to the changes to the $n(v)$ between the cleaning sessions. Since T has depth $O(\log\log n)$, the number of changes due to an extract-min is $O(\log\log n)$. To analyze the cost of inserting a key x, we need to take a little bit of care because T may change shape while x descends through the buffers. Note that x only changes a value $n(v)$ when it is being put in $B(v)$. If v is subsequently cleaned, and the buffers emptied, x will no longer contribute to any changes. Hence the changes due to inserting x are dominated by the cost of flushing x.

Suppose $n(v) = n$ when x is put in $B(v)$. If x is going to be flushed from $B(v)$ to $B(w)$, v may not first get dirty, so by the time of flushing $n(v) = \Theta(n)$. Hence, by (1), $d(v) = \Theta(n(v)^{1/k}) = \Theta(n^{1/k})$, and $n(w) = \Theta(n(v)^{1-1/k}) = \Theta(n^{1-1/k})$. Thus, the total cost of flushing x follows the recurrence $T(n) \leq T(O(n^{1-1/k})) + O(S(O(n^{1/k})))$ where n is the number of keys at the time where x was inserted. Since this recurrence dominates $O(\log\log n)$, this completes our proof of Proposition 6.

3 Proof of Proposition 5

In this section, we prove the statement of Proposition 5:

> Given a set X of n keys, a static data structure can be constructed in $n^{O(1)}$ time and space so that any batch of n searches in X can be performed in time $O(n \log\log n)$.

In the proof we follow the general pattern of Andersson et.al. [AHNR95] of making an adequate range reduction, and then apply packed sorting. Let $q = \log n \log\log n$. Each key is viewed as consisting of (w/q)-bit characters. Our first step is to construct a trie t_X over the keys in X. That is, t_X is a rooted tree over all prefixes of keys in X. The leaves are the keys in X themselves and the root is the empty string. The parent of a non-empty prefix is obtained by removing its last character, and the edge from the parent is labeled with this character. The edges to the children of a node are ordered according to their labels. This induces the lexicographical ordering of the leaves of t_X, and these leaves are the keys of X.

Let Y be the set of keys to be searched for in X. In order to search $y \in Y$, we first find its longest common prefix z with any key in X. Let a be the next character in y. Thus, za is the shortest distinguishing prefix of y relative to X. By placing a between the the labels of edges leaving z in t_X, we find y's position in X. Note that there may be several different keys $y \in Y$ leading to the same distinguishing prefix za, hence ending at the same position in t_X. Fortunately, we are not asked to find the order between keys in Y.

In order to identify z, we build a dictionary over all prefixes of keys in X. A *dictionary* over a set A is a data structure that for any a decides if $a \in A$, and, in if so, returns a unique index $\in O(|A|)$ that can used for efficiently storing and retrieving information associated with a. From [And96], we have

Lemma 13 (Andersson) *A linear space dictionary over m keys with constant access time can be constructed in $m^{O(1)}$ time.*

Here we have $m = qn = n \log n \log\log n$ prefixes, so the dictionary is built in $n^{O(1)}$ time. The dictionary together with the trie forms our data structure for X. Using the dictionary, we can find the longest common prefix z of y with a prefix of a key in X by a binary search in time $O(\log k) = O(\log\log n)$. Also, the dictionary is used to give us the position of z in t_X.

Given the set Y, first, in time $O(n \log\log n)$, we identify the above prefixes z for all $y \in Y$. Afterwards, for each such z, we sort the multi-set $\{a : za \in \text{Prefixes}(X) \cup \text{Prefixes}(Y)\}$. This sorting gives us the the positioning of all keys y with z as longest common prefix with keys in X. As pointed out in [AHNR95], from [AH92] we have

Lemma 14 (Albers and Hagerup) *We can sort m keys of length $O(w/\log m \log\log m)$ in $O(m)$ time.*

Thus each of the $(w/\log n \log\log n)$-bit character sets $\{a : za \in \text{Prefixes}(X) \cup \text{Prefixes}(Y)\}$ are sorted in linear time. Each key $y \in Y$ contributes to only one such character set. If X contributes with more characters to a set than Y, it is with at least 2 characters, but X can provide at most $\leq 2n - 2$ such branchings (= number of edges in binary tree with n leaves), leading to a total of $4n$ characters over all multi-sets (which can be reduced to $3n$ by a tighter analysis). In conclusion, all character sets are sorted in $O(n)$ total time, completing the placement of all keys from Y between the keys in X. This completes the proof of Proposition 5

4 With addition, shift, and bit-wise boolean operations

By a *Practical RAM*, we mean a RAM where the only computation instructions are addition, shift, and bit-wise boolean operations. So far, the only place where we have used any other operations; namely multiplication, is in the dictionary from Lemma 13. The dictionary was used for a binary search of the longest common prefix of each key $y \in Y$. The results from [AMRT96] imply that asking one membership query takes time $\Omega(\sqrt{\log n / \log\log n})$. We can, however, right as well do all these binary searches in parallel. Increasing the buffer size $|Y|$ to $m = |\text{Prefixes}(X)| = n \log n \log\log n$, this means that we can make a batch of m queries at the time to a dictionary over

m elements. Using standard techniques, it is straightforward to derandomize the duplicate finding technique from [Tho97] to give

Lemma 15 (Thorup) *In $O(m^{2+\varepsilon}w)$ time we can construct a dictionary over m w/q-bit keys, so that m batched queries can be processed in $O(m(\log\log m)/q)$ time.*

Proof: In $O(m^{2+\varepsilon}w)$ time, we identify the $O(w\log n)$ random bits to make the randomized duplicate finding [Tho97] for X run within its expected time bound of $O(m(\log\log m)/q)$. Given a set Y of query keys, we run the duplicate finding on $X\cup Y$ with the previously found random bits. All keys from X and some keys from Y will be grouped correctly according to equality, but some keys from Y may be discarded. However, if a key y is discarded, it is not equal to any key in X. ■

The significance of the dividing the time with q is that if we make our binary search in the recursive style of van Emde Boas' data structure [vB77, vBKZ77], the total cost of finding the longest common prefixes becomes $O(\sum_{i=0}^{O(\log\log n)}(m(\log\log n)/2^i)) = O(m(\log\log m))$ instead of $O(m(\log\log m)^2)$.

The theoretical problem in using Lemma 15 is the dependence on w which may be unbounded in m. Suppose $w > m^{4+\varepsilon}$. In order to construct an efficient batched dictionary in time polynomial in m, we need to introduce some more techniques. First note

Lemma 16 *In $O(m)$ time and space we can construct a dictionary over m (w/q)-bit keys, $q = \log n\log\log n$, so that m batched queries can be processed in $O(m)$ time.*

Proof: Let $X = \{x_1,\dots,x_m\}$ be the dictionary keys and $Y = \{y_{m+1},\dots,y_{2m}\}$ be the query keys. First append the $1+\log_2 m$ bits of the unique index of each key to the key. Second sort all the keys-index pairs lexicographically in linear time using Lemma 14. Now all the membership queries can be answered by a simple linear scan of the sorted list. ■

In order to make use of Lemma 16, we will construct a *signature function* f from w-bit keys to (w/q)-bit keys which is 1-1 on X. Assuming such f, given any $y\in Y$, there is at most one $x\in X$ with $f(x)=f(y)$, and then $y\in X \iff y=x$.

Clearly we need to consider at most m^2 bit positions in order to distinguish any pair of keys from X. Since $w > m^{4+\varepsilon}$, any function picking out m^2 such distinguishing bits would be a perfect signature function. In fact, it suffices if we can get down to $m^4 = o(w/q)$ bits, and that is exactly what we will do below.

For each pair $(x_1,x_2)\in X^2$, we will be interested in the least significant distinguishing bit, which is the least significant bit (lsb) of $a = x_1\oplus x_2$. Computing the index of the least significant bit is know to take time $\Theta(\log\log w)$ time on the Practical RAM [BMM97], but for our purposes it suffices to compute lsb-only(a) denoting the key with all but the least significant bit of a set. In the proof of Lemma 8 in [BMM97] it is shown that lsb-only(a) can be computed in constant time. A slightly simpler construction than theirs is

$$\text{lsb-only}(a) \equiv a \wedge (\neg(a\oplus(\neg a+1))$$

Let α be the key with all the least significant distinguishing bits of X set, i.e. $\alpha = \bigvee_{(x_1,x_2)\in X^2, x_1\neq x_2}\text{lsb-only}(x_1\oplus x_2)$. Then $g_\alpha : y\mapsto y\wedge\alpha$ is a 1-1 function on X. Constructing α and hence the function g_α takes time $O(m^2)$, but computing g takes constant time.

Let high and low be the functions extracting the first and the last half of a key. Since there are $\leq m$ 1s in α, there can be at most $(m^2/2)^2 = m^4/4$ values of i such that $(\text{high}(\alpha)\lll i)\wedge\text{low}(\alpha)\neq 0$. Here $\lll i$ denotes a cyclic shift to the left by i positions. Since α has length $> m^4/4$, we conclude that there is an $i\leq m^4/4+1$ so that $(\text{high}(\alpha)\lll i)\wedge\text{low}(\alpha)\neq 0$. With this choice of i, $h_i : z\mapsto(\text{high}(z)\lll i)\vee\text{low}(z)$ is a 1-1 function on $g_\alpha(X)$, and hence $g_\alpha\circ h_i$ is a 1-1 function on X. Moreover, h_i halves the length of its input. Finding i and hence the function h_i takes time

$O(m^4)$, but computing h_i takes constant time. The above process can be repeated on $\alpha' = h_i(\alpha)$ constructing a new function h'_j which is 1-1 on $h_i(g_\alpha(X))$. Repeating $O(\log\log m)$ times, we arrive at signatures of the desired length $w/\log m \log\log m$. In order to be able to trace back from the signatures to the original keys, as in the proof of Lemma 16, we append indices to the keys and append a corresponding number of 1s to the original value of α. As a result, the above process will lead to signatures with the indices appended. Lemma 16 can now be applied directly to the signatures.

The recursive construction of the $\log\log m$ functions h_i takes $O(m^4 \log\log m) = O(m^{4+\varepsilon})$ time, which is hence the complexity of constructing our dictionary. Computing the signature of one key would take time $O(\log\log m)$. However, every time we apply a function h_i we half the size of the key. Packing the keys in words, this means that one word operation allows us to work on twice as many keys at the time. Hence, given $\log m$ keys, the signatures can be computed in constant time per key. The reader is referred to [Tho97] for details of such parallelization. In conclusion,

Lemma 17 *If $w > m^{4+\varepsilon}$, in $O(m^{4+\varepsilon})$ time and $O(m)$ space, we can construct a dictionary over m keys, so that m batched queries can be processed in $O(m)$ time.*

Combined with Lemma 15, we get

Proposition 18 *In $O(m^{4+\varepsilon}w)$ time we can construct a Practical RAM dictionary over m w/q-bit keys, so that m batched queries can be processed in $O(m(\log\log m)/q)$ time.*

This completes the proof of Theorem 1.

References

[AH92] S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, 1992.

[And96] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. FOCS*, 1996, pages 135–141.

[AHNR95] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. 27th ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[AMRT96] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static Dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *Proc. 37th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 441–450, 1996.

[BMM97] A. Brodnik, P.B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds, To appear at WADS'97.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993. See also STOC'90.

[Mil94] P.B. Miltersen, Lower bounds for union-split-find related problems on random access machines, *in* "Proc. 26th STOC," pp. 625–634, 1994.

[Ram96] R. Raman. Priority queues: small monotone, and trans-dichotomous. *Proc. ESA'96, LNCS 1136*, 1996, 121–137.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms* SIAM, 1983.

[Tho96] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

[Tho97] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 352–359, 1997.

[vB77] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inf. Proc. Lett.* **6** (1977), 80–82.

[vBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Syst. Th.* **10** (1977), 99–127.