

134134

# BRICS

Basic Research in Computer Science

## Dictionaries on $AC^0$ RAMs: Query Time $\Theta(\sqrt{\log n} / \log \log n)$ is Necessary and Sufficient

Arne Andersson  
Peter Bro Miltersen  
Søren Rils  
Mikkel Thorup



BIBLIOTEKET  
DATALOGISK SAMLING  
AARHUS UNIVERSITET  
Ny Munkegade, Bygn. 530

BRICS Report Series

ISSN 0909-0878

RS-97-14

June 1997

BRICS RS-97-14 Andersson et al.: Dictionaries on  $AC^0$  RAMs: Query Time...

Matematisk Institut  
Aarhus Universitet  
Trykkeriet

Copyright © 1997,

BRICS, Department of Computer Science  
University of Aarhus. All rights reserved.

Reproduction of all or part of this work  
is permitted for educational or research use  
on condition that this copyright notice is  
included in any copy.

See back inner page for a list of recent BRICS Report Series publications.  
Copies may be obtained by contacting:

**BRICS**  
Department of Computer Science  
University of Aarhus  
Ny Munkegade, building 540  
DK-8000 Aarhus C  
Denmark  
Telephone: +45 8942 3360  
Telefax: +45 8942 3255  
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide  
Web and anonymous FTP through these URLs:

<http://www.brics.dk>  
<ftp://ftp.brics.dk>  
This document in subdirectory RS/97/14/



134/134

**BIBLIOTEKET**  
DATALOGISK SAMLING  
AARHUS UNIVERSITET  
Ny Munkegade, Bygn. 530

# Dictionaries on $AC^0$ RAMs: Query time $\Theta(\sqrt{\log n / \log \log n})$ is necessary and sufficient\*

Arne Andersson  
Lund University  
arne@dna.lth.se

Peter Bro Miltersen†  
BRICS‡  
bromille@brics.dk  
Søren Riis §  
University of Leeds  
pmlsr@amsta.leeds.ac.uk

Mikkel Thorup  
University of Copenhagen  
mthorup@diiku.dk

## Abstract

In this paper we consider solutions to the dictionary problem on  $AC^0$  RAMs, i.e. random access machines where the only restriction on the finite instruction set is that all computational instructions are in  $AC^0$ . Our main result is a tight upper and lower bound of  $\Theta(\sqrt{\log n / \log \log n})$  on the time for answering membership queries in a set of size  $n$  when reasonable space is used for the data structure storing the set; the upper bound can be obtained using  $O(n)$  space, and the lower bound holds even if we allow space  $2^{\text{poly} \log n}$ . Furthermore, the lower bound holds even for static dictionaries, while the upper bound holds also for dynamic dictionaries; insertion and deletions can be accommodated in expected amortized time  $\Theta(\sqrt{\log n / \log \log n})$ .

Several variations of this bound is also obtained, including tight upper and lower bounds on the storage space if the query time must be constant and bounds valid for non- $AC^0$  RAMs if the execution time of an instruction computing a function is measured as the minimal depth of a polynomially sized unbounded fan-in circuit computing the function. We refer to this model as the *Circuit RAM*. As an example of the latter, we show that any RAM instruction set which permits a linear space, constant query time solution to the static dictionary problem must have an instruction of depth  $\Omega(\log w / \log \log w)$ , where  $w$  is the word size of the machine (and  $\log$  the size of the universe). This matches the depth of multiplication and integer division, used in the two level hashing scheme of Fredman, Komlós and Szemerédi.

## 1 Introduction

The most fundamental data structure problem is the static dictionary problem: Given a set  $A$  containing  $n$  keys, each being a bit vector of length  $w$ , store it as a data structure in the memory of a random access machine, using few memory registers, each containing  $w$  bits (a word), so that membership queries “Is  $x \in A$ ?” can be answered efficiently for any value of  $x$ . In addition, if  $x \in A$ , we might want to retrieve some information associated with  $x$ . We are

\*A preliminary version of this paper appeared at FOCS'98.

†This work was initiated while the author visited Copenhagen with support from the Danish Science Research Council. Most of the work was carried out while the author was employed by University of Toronto. Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

‡Centre of the Danish National Research Foundation, University of Aarhus, Ny Munkegade, Aarhus, Denmark.

§Supported by Carlsbergfondet. This work was initiated while the author visited Copenhagen with support from the Danish Science Research Council.



interested in tradeoffs between the storage space (measured by the number of registers used) and the query time.

The set  $A$  can be stored as a sorted table using  $n$  memory registers allowing queries to be answered using binary search in  $O(\log n)$  time. Yao [26] first considered the possibility of improving this solution and provided an improvement for certain cases. Fredman, Komlós and Szemerédi [13] showed that for all values of  $w$  and  $n$ , there is a storage scheme using  $O(n)$  memory registers, so that queries can be answered in constant time. Their technique is two level hashing based on the family of hash functions  $h_t(x) = (kx \bmod p) \bmod s$ . Thus, multiplication and integer division are used by the query program. Since these instructions are usually considered expensive, it is natural to ask whether their use can be avoided. The family may be replaced by a family due to Knuth [17, p.509] using only multiplication, bitwise Boolean operations, and shifts (see [10] for details), so integer division is not essential. But so far, the existence of optimal schemes for static dictionaries with multiplication replaced by cheaper instructions has remained an open problem.

One natural, robust, and generally accepted formalization of what it means for a function to be "cheap" is membership in  $AC^0$ , i.e. the function should be implementable by a constant depth, unbounded fan-in (AND,OR,NOT)-circuit of size  $w^{O(1)}$ . This is the approach taken in this paper; we consider solving the static dictionary problem on  $AC^0$  RAMs. Informally, an  $AC^0$  RAM is a RAM with an instruction set consisting of direct and indirect addressing, conditional jump, and a finite number of computational instructions each mapping a constant number of words to one word. All computational instructions belong to  $AC^0$ . All instructions are charged unit cost.

Our main results are the following.

**Theorem A.** *There is a solution to the static dictionary problem on an  $AC^0$  RAM using  $O(n)$  space and with a worst case query time of  $O(\sqrt{\log n}/\log \log n)$ . On the other hand, query time  $O(\sqrt{\log n}/\log \log n)$  is not possible on any  $AC^0$  RAM even if space  $2^{\text{poly} \log n}$  is allowed.*

**Theorem B.** *For any  $\epsilon > 0$ , there is a solution to the static dictionary problem on an  $AC^0$  RAM with constant query time using space  $O(2^{w^\epsilon})$  for storing sets of size  $n = 2^{w^{\epsilon(1)}}$ . On the other hand, there is no constant query time solution using space  $2^{w^{\epsilon(1)}}$  for storing sets of size  $n = 2^{(\log w)^\epsilon}$  on any  $AC^0$  RAM.*

The lower bounds above actually holds in a more general model than the  $AC^0$  RAM, the Circuit RAM. By a Circuit RAM with word size  $w$  we mean a RAM with direct and indirect addressing, conditional jump, and a number of computational instructions each mapping a constant number of words to one word. The computational instructions can be chosen arbitrarily, i.e. they do not have to be in  $AC^0$ . However, each instruction has an associated cost. The cost of direct and indirect addressing and conditional jump is 1. The cost of a computational instruction is  $d$ , where  $d$  is minimum depth of a circuit of size  $w^{O(1)}$  implementing the instruction. The time of an execution is the sum of the costs of all the instructions executed.

Note the similarity between this definition and the *log cost RAM*, the difference being that the log cost RAM captures hardware operating sequentially on the bits in the words, while the Circuit RAM captures hardware exploiting unbounded fan-in parallelism (but only inside the CPU).

If a problem can be solved by an  $AC^0$  RAM using time  $t$ , it can also be solved by a Circuit RAM using time  $O(t)$ . We believe the Circuit RAM is quite a natural model of computation, deserving further study. Note that we do not disallow expensive instructions such as multiplication, we just charge them more, using the well-studied depth measure from circuit complexity. In the Circuit RAM model, the two level hashing scheme of [13] has query time  $O(\log w/\log \log w)$ , since multiplication and integer division [7] are in  $NC^1$  and any function in  $NC^1$  has polynomial size circuits of depth  $O(\log w/\log \log w)$  [9]. We provide a matching lower bound that also generalizes the lower bound of Theorem A.

**Theorem C.** *There is no Circuit RAM solution to the static dictionary problem using space  $2^{\text{poly} \log n}$  and with query time  $o(\log w/\log \log w)$ . In fact, for any word length  $w$ , there is a fixed set  $X \subseteq \{0,1\}^w$  of size  $n = 2^{\Theta(\log^2 w/\log \log w)}$  so that any Circuit RAM dictionary for  $X$  using space  $2^{\text{poly} \log n}$  has worst case query time  $\Omega(\sqrt{\log n}/\log \log n) = \Omega(\log w/\log \log w)$ . The result holds even for Monte Carlo schemes and with worst case time replaced with average case time (where the average is over the set  $X$ ).*

## Techniques used

Our upper bounds are shown using a clustering structure. The data structure is based on two kinds of "hash-like" functions, clustering functions clustering the input into Hamming balls of small radius and cluster busters hashing each individual Hamming ball. The  $AC^0$  instruction set required is non-standard, but not unrealistic. We hope that the clustering structure might inspire future hardware design by pointing out potentially useful instructions.

Our lower bound relies on a recursive construction of a difficult set. At each level of the recursion we use probabilistic methods including Håstad's switching lemma [15].

## Related research

Apart from the sorted table, previous linear space solutions to the static dictionary problem implementable on  $AC^0$  RAMs were constructed by Tarjan and Yao [22], achieving constant query time if  $w = c \log n$  for some constant  $c$ , and by Willard [25] and Karlsson [16], achieving query time  $O(\sqrt{w})$ .

Previous lower bounds for the static dictionary problem have been obtained by Fich and Miltersen [12, 20]. The former paper gives lower bounds for the case where the computational instructions allowed are addition, subtraction and multiplication (the Classical RAM), the latter for the case where the instructions allowed are addition, subtraction, bitwise Boolean operations and arbitrary shifts (the Practical RAM). It is interesting to compare the previous bounds with the new ones. If constant query time is desired, the following space bounds are necessary and sufficient (for sets of size  $w \ll n \ll 2^w$ ): On the Classical RAM  $\epsilon 2^w$ , on the Practical RAM  $2^{\epsilon w}$ , and on the  $AC^0$  RAM  $2^{w^\epsilon}$ . If linear space is desired, logarithmic query time is optimal for the Classical RAM, but not for the  $AC^0$  RAM. The best current lower bound for the Practical RAM is the  $\Omega(\sqrt{\log n}/\log \log n)$  one implied by this paper, improving the  $\Omega(\log \log n)$  one of [20].

From circuit complexity there is a result of Mansour, Nisan, and Tiwari [19] (building on [18]) that no  $AC^0$  circuit implements a family of universal hash functions  $\{H_k\}$ , where "implements" means that the circuit on input  $(k, x)$  computes  $H_k(x)$ . This result is an immediate corollary of our lower bounds as the following argument shows: Suppose, to the contrary, that an  $AC^0$  circuit computing a universal family exists. This family could replace the family of hash functions used by Fredman, Komlós and Szemerédi, and thus yielding an  $O(n)$  space,  $O(1)$  query time  $AC^0$  RAM solution to the static dictionary problem, contradicting Theorem A, B and C. However, our result is more general: In the context of static dictionaries, their result shows that we cannot find a single  $AC^0$  instruction to replace multiplication in universal hashing schemes, such as the FKS-technique, but it does not tell us anything about what can be achieved by programs which are not based on universal hashing. As our upper bound shows, there are alternatives to universal hashing for solving the static dictionary problem, and our lower bounds cover all possible programs.

## Notation

In the rest of the paper, the symbol  $n$  will always denote the size of the set we want to store in a dictionary. The symbol  $w$  is the length of the keys to be stored and the word size of the machine. In our recursive constructions, we might want to store sets smaller than  $n$  containing

keys shorter than  $w$ . Usually, the size of such a set will be denoted  $m$  and the length of the keys  $b$ .

We shall assume that various fractions and logarithms are integers to avoid looking at tedious special cases.

We let  $\oplus$  denote exclusive-or. Letters  $x, y, z, z_i$  etc normally represents bit vectors. The  $j$ th bit of vector  $x$  is denoted  $x[j]$ . Often an  $l$ -bit vector  $x$  is identified with the set  $\{j|x[j] = 1\}$ . If a boolean operation is applied to two bit-vectors of the same length, the application is understood to be bit-wise. For example, given two  $l$ -bit vectors  $x, y$ , we have  $x \vee y = x \cup y$  and  $x \wedge y = x \cap y$ . The symbol  $\setminus$  denotes set difference (of sets or words). By  $|x|$  we mean the Hamming weight of  $x$ , i.e. the number of 1-bits in  $x$ . Note that  $|x \oplus y|$  is thus the Hamming distance between  $x$  and  $y$ . The Hamming ball with center  $c$  and radius  $r$  is the set of bit vectors  $\{x \in \{0, 1\}^w : |x \oplus c| \leq r\}$ . The Hamming ball with center  $0^w$  and radius  $r$  is denoted  $H_r$ . In some cases, we use  $H_r^b$  to mark that the keys are of length  $b$ .

Let  $K \subseteq \{1, 2, \dots, w\}$ . The projection function  $\pi_K : \{0, 1\}^w \rightarrow \{0, 1\}^{|K|}$  selects the bits marked by  $K$  from its input.

## 2 Simple upper bounds

**Lemma 1** Let  $A \subseteq \{0, 1\}^w$  be a set of at most  $n$  elements, and let  $r < w/20$ . Let  $K$  be a random subset of  $\{1, \dots, w\}$  of size  $10(\log n)w/r$ . With probability at least  $1 - 1/n^3$ , for all  $x, y \in A$ ,  $|x \oplus y| > r$  implies  $\pi_K(x) \neq \pi_K(y)$ .

**Proof:** Let  $x, y$  be a particular pair in  $A$  with  $|x \oplus y| > r$ . Let  $i$  be a random index in  $\{1, \dots, w\}$ . The probability that  $x[i] = y[i]$  is at most  $1 - r/w$ . Thus, if  $K$  is a randomly chosen set of size  $10(\log n)w/r$ , the probability that  $\pi_K(x) = \pi_K(y)$  is at most  $(1 - r/w)^{10(\log n)w/r}$ . The probability that  $\pi_K(x) = \pi_K(y)$  for any such pair  $x, y$  is therefore at most  $\binom{n}{2}(1 - r/w)^{10(\log n)w/r} < 1/n^3$ . ■

We use  $\pi_K$  as a clustering function. For the cluster buster we need:

**Lemma 2** Let  $A \subseteq H_r$  with  $|A| = m$ . Let  $I_1, I_2, \dots, I_w$  be randomly chosen subsets of  $\{1, \dots, 10r \log m\}$ , each of size  $5 \log m$ . Let  $\mu : \{0, 1\}^w \rightarrow \{0, 1\}^{10r \log m}$  be the map defined by  $\mu(x) = \cup_{i:|x|=1} I_i$ . Then  $\mu$  is 1-1 on  $A$  with probability  $> 1 - 1/m^3$ .

**Proof:** Consider any pair  $x, y$  in  $A$  with  $x \neq y$ . There is an index  $i$ , so that  $x[i] \neq y[i]$ . Assume without loss of generality that  $x[i] = 1$  and  $y[i] = 0$ . We see that  $\mu(x) = \mu(y)$  implies  $I_i \subseteq \mu(y)$  but  $|\mu(y)| \leq 5r \log m$ , so  $\Pr(I_i \subseteq \mu(y)) \leq 2^{-5 \log m} = \frac{1}{m^5}$ . The number of pairs in  $A$  is  $\leq m^2$ , so  $\mu$  is 1-1 on  $A$  with probability  $1 - 1/m^3$ . ■

**Lemma 3** There is an  $AC^0$  function  $h_r : \{0, 1\}^w \rightarrow \{0, 1\}^{10r^2 \log w}$  which is 1-1 on  $H_r$ .

**Proof:** Apply Lemma 2 with  $A = H_r$ . ■

We now show how to use Lemmas 1 and 3 for constructing  $AC^0$  RAM solutions to the static dictionary problem. The basic idea for storing a set  $A$  is this. First,  $A$  is hashed using the function  $\pi_K$  with  $K$  chosen as in Lemma 1. This partitions (clusters) the set into a number of buckets, each being a subset of some Hamming ball of radius  $r$ . Each such subset  $A'$  can be stored by storing the center  $c$  of the Hamming ball in a single word, and then using the injective function of Lemma 3 to store the set  $\{x \oplus c | x \in A'\} \subseteq H_r$  in a collision free hash table using space  $2^{10r^2 \log w}$ . Each bucket is indexed by a signature of  $10(\log n)w/r$  bits (i.e.

the value of  $\pi_K$  for the elements in the bucket), so if we store pointers to the buckets naively using an array, indexed by the possible values of  $\pi_K$ , we would use space  $2^{10(\log n)w/r}$  which is too much for our purpose. In order to avoid this, we recursively store the signatures using our scheme, with pointers to the buckets as associated information. We call the resulting data structure a *clustering structure*.

A query program for the clustering structure needs some non standard instructions: We need an instruction  $\text{Select}(K, x)$  which computes the function  $\pi_K$  on input  $x$ . Here the word  $K$  is interpreted as the concatenation of a list of numbers between 1 and  $w$ . Let these numbers be  $k_1, k_2, \dots, k_w/\log w$ . The output is  $x[k_1|x[k_2|\dots|x[k_w/\log w]$ . This function can be seen to be in  $AC^0$ . Note that in order to store the index  $K$  of the clustering function in constant space, we must have  $K \leq w/\log w$  or equivalently,  $r > 10(\log n)(\log w)$ .

We also need an instruction  $\text{BustHammingBall}(r, x)$ , where  $r$  is a number between 1 and  $w$ . This instruction applies the function  $h_r$  of Lemma 3 to  $x$ , and is easily seen to be in  $AC^0$ , given this lemma.

In more detail: We represent a set of  $m$   $b$ -bit keys in a data structure  $T$  and the function  $\text{Find}(T, x, b)$  returns a reference to  $x$ ; this reference is simply a number in the interval  $[0, m-1]$ . Let  $K$  and  $r$  be as in Lemma 1. Let  $Y = \{\pi_K(x) | x \in X\}$ . The set  $Y$  is represented recursively in the data structure  $T.Y$ .  $T$  also contains an array  $T.C : [0, m-1] \rightarrow \{0, 1\}^b$  such that if  $i = \text{Find}(T.Y, \pi_K(x), b)$  for some  $x$ , then there is such an  $x$  for which  $T.C[i] = x$ . Thus, according to Lemma 1, for any  $x \in X$ ,  $|x \oplus T.C[\pi_K(x)]| \leq r$ .

As the recursion proceeds, the key length  $b$  will decrease. When  $b \leq b_0$ , for some specified value  $b_0$  we switch to some other data structure. In the recursive implementation below, we mark this switch by the function  $\text{Find}'(T, x)$ . Our implementation of  $\text{Find}$  is of the following recursive form:

**Algorithm A:**  $\text{Find}(T, x, b)$

A.1. if  $b \leq b_0$ , return  $\text{Find}'(T, x)$ .

A.2.  $i \leftarrow \text{Find}(T.Y, \text{Select}(K, x), |K|)$ .

A.3. return  $T.B[i, \text{BustHammingBall}(r, x \oplus T.C[i])]$ .

Above,  $T.B$  is a 2-dimensional array with entries from  $[0, m-1] \times \{0, 1\}^{10r^2 \log w}$ . Since all elements from  $X$  are mapped to different entries in  $B$ , we just need to fill these entries with different values in order to make our function  $\text{Find}$  1-1 on  $X$ . The space of the data structure  $T$  is the space of the recursive data structure  $T.Y$  for the at most  $m$   $(10(\log n)w/r)$ -bit keys in  $Y$  plus  $1 + m + m \cdot 2^{10r^2 \log w} = O(m \cdot 2^{10r^2 \log w})$  words for storing  $K$  and the arrays  $T.C$  and  $T.B$ .

We get the following upper bounds, the first of which gives the upper bound of Theorem B of the introduction.

**Corollary 4** For any  $\epsilon > 0$ , there is an  $AC^0$  RAM solution to the static dictionary problem for sets of size  $n < 2^{w^{1/4}}$  with query time  $O(1)$  using space  $2^{w^\epsilon}$ .

**Proof:** Let  $r = 10(\log n)w^\epsilon/100$ . Apply the construction above. After one iteration, we reduce the size of the keys to be stored to  $10(\log n)w/r = w^{1-\epsilon/100}$ . By iterating  $100/\epsilon$  times we thus have reduced the domain of the keys to be stored to constant size, i.e. a trivial problem. At each recursive level we use  $O(n \cdot 2^{10r^2 \log w})$  space. There are  $100/\epsilon$  recursive levels, so the total space used is  $(100/\epsilon)n2^{10r^2 \log w} < 2^{w^\epsilon}$ . ■

**Corollary 5** There is an  $AC^0$  RAM solution to the static dictionary problems for sets of size  $n$  with query time  $O(\log w / \log \log w)$  using space  $2^{\text{poly} \log n}$ .



**Proof:** Let  $r = 10(\log n)(\log w)^2$ . After one iteration of the general construction the size of the keys is reduced by a factor  $\Theta(\log w)$ , so after  $O(\log w / \log \log w)$  iterations, the problem is trivial. The space used is  $O((\log w / \log \log w)n2^{10r^2(\log w)})$ . This is  $2^{\text{poly} \log n}$ , if  $\log n \geq \log w / \log \log w$ . But if  $\log n < \log w / \log \log w$ , the sorted table/binary search solution yields the desired bound. ■

**Lemma 6** *A set of  $n$  keys, each containing  $k$  bits, can be stored using linear space and with query time  $O\left(\frac{\log n}{\log \frac{n}{k}}\right)$ .*

**Proof:** We store the set as a *Packed B-tree* [3], i.e. a B-tree of degree  $w/k$ , where each B-tree node is represented inside a word. In order to search the packed B-tree, we need an instruction  $\text{Rank}(P, X, k)$  that views the words  $P$  and  $X$  as divided in  $k$ -bit fields. If the fields in  $P$  are left-right sorted, the instruction returns the rank of the leftmost field in  $X$  among the fields in  $P$ . This instruction is clearly in  $AC^0$ . ■

**Corollary 7** *There is an  $AC^0$  RAM solution to the static dictionary problems for sets of size  $n$  with query time  $O(\sqrt{\log n / \log \log n})$  and space  $2^{(\text{poly} \log n)(\text{poly} \log w)}$ .*

**Proof:** Let  $r = 10(\log n)^2(\log w)$ . Each iteration of the general construction reduces the number of bits in the keys by at least a factor  $\log n$ , so after  $\sqrt{\log n / \log \log n}$  iterations, the number of bits in the keys is reduced by at least a factor  $2^{\sqrt{\log n \log \log n}}$ . Thus, we can apply lemma 6 with  $k = w/2^{\sqrt{\log n \log \log n}}$  and search in the set of signatures using time  $O(\sqrt{\log n / \log \log n})$ . The space used is  $O((\sqrt{\log n / \log \log n})n2^{10r^2(\log w)}) = 2^{(\text{poly} \log n)(\text{poly} \log w)}$ . ■

### 3 Improved upper bounds

In this section we improve the space to linear in  $n$  and independent of  $w$ .

**Definition 8** *Let  $H(b, r)$  be the maximum cost searching in a data structure storing at most  $n$  keys in linear space, where each key belongs to  $H_r^b$ . Also, let  $T(b) = H(b, b)$ .*

Note that we do not include the size of the stored (sub-)set in this notation. This turns out to be convenient, for in most of our calculations we only need the fact that the total number of keys is  $n$ . The total search cost in our global data structure is  $T(w)$  ( $= H(w, w)$ ).

**Lemma 9** *Assume that for each set  $A \subseteq H_r^b$ , of at most  $n$  keys, there is a function  $f$  which, when applied on  $A$ , produces values in  $H_r^b$  such that for any  $x_i, x_j \in A$ ,  $|x_i \oplus x_j| > r^n$  implies  $f(x_i) \neq f(x_j)$ . Then*

$$H(b, r) = O(1) + H(b', r') + H(b, r'')$$

**Proof:** (very brief sketch) We use an improved version of the clustering structure. The general idea is to use a cluster function to split the search problem into two simpler problems, defined recursively. First, we search for a data structure associated with the value  $f(x)$  and then we search for  $x$  in this data structure, utilizing the fact that  $x$  is contained in a Hamming ball with smaller radius. ■

### Base case of recursion

After a number of recursive steps, the search problems will be simple enough to be handled directly in linear space.

First, we note that  $H(b, 0) = O(1)$  since a Hamming ball of radius 0 only contains one key. Next, we note that Lemma 6 gives  $T(b) = O(\log n / \log(w/b))$ . Finally, we get one base case by simulating traditional hash coding by a multiplication table.

**Lemma 10**  $T(c \log n) = O(c)$ .

**Proof:** (Sketch) We store the keys in a hash-coded path-compressed trie. We use  $\log n$  bits for branching, hence the height of the trie is  $c$ . At each node, the outgoing edges are stored in a hash table. We use perfect hashing and simulate multiplication by a precomputed multiplication table. ■

### Clustering the input into Hamming balls

We improve over Lemma 1 by replacing the selected bits by the parity of  $\text{poly} \log w$  bits. The following lemma is stated without proof.

**Lemma 11** *Let  $I$  be a random  $k \times l$  ( $k$  odd) matrix of independently chosen bit-positions in  $\{1, 2, \dots, w\}$ . Define the function  $C : \{0, 1\}^w \rightarrow \{0, 1\}^l$  by  $C(x)[j] = \bigoplus_{i=1}^k x[I[i, j]]$ . Then, for any  $w$ -bit strings  $x, y$  with  $|x \oplus y| = \tau$ ,  $\Pr(C(x) = C(y)) \leq \max(e^{-\tau k / (2w)}, e^{-1/4})$ .*

**Lemma 12** *Let  $A \subseteq \{0, 1\}^b$  be a set of at most  $n$  elements, and let  $r < b$ . Let  $k = (\log b)^d$  and let  $l = 100b \log n / (r \log^d b)$ . Let  $C$  be the random function defined in Lemma 11. Then, with probability at least  $1 - 1/n^3$ , for any pair  $x, y$  in  $A$  with  $|x \oplus y| = \tau$ ,  $C(x) \neq C(y)$ .*

**Proof:** There are  $n^2$  pairs  $x, y \in A$  so the bound in Lemma 11 is sufficient to prove the lemma. ■

We can use the function of lemma 12 as a clustering function. For this, in addition to the *Select* instruction we need an instruction  $\text{BlockParity}(x, k)$  that views the word  $x$  as divided in  $k$ -bit fields, computes the parity of each field, and concatenates the results. If we restrict the parameter  $k$  to be less than  $\text{poly} \log w$ , the instruction is in  $AC^0$ . Note that we need to select  $k \times l$  bits where  $k = (\log b)^d$  and  $l = 100b \log n / (r \log^d b)$ . To do this using the *Select* instruction, we must have  $r \geq 100b \log n \log w / w$ .

**Corollary 13** *If  $r \geq 100b \log n \log w / w$  then*

$$T(b) = 1 + T\left(\frac{b \log n}{r \log^d b}\right) + H(b, r).$$

### Cluster Busters

We improve over Lemmas 2 and 3 by an alternative technique, reducing the radius of the Hamming ball in a number of iterations.

**Lemma 14** *For any  $a$ , there is an  $AC^0$  circuit  $C : \{0, 1\}^w \rightarrow \{0, 1\}^{8 \log n}$  such that for any  $x, y \in H_{\log n / (a \log w)}$ ,  $|x \oplus y| > \log n / (a^2 \log w)$  implies  $C(x) \neq C(y)$ .*

**Proof:** For  $i = 1, \dots, w$ , choose randomly  $I_i \subseteq \{0, 1, \dots, w-1\}$ ,  $|I_i| = 4\alpha \log w$ , and define  $C(x) = \bigcup_{i \in x} I_i$ .

Consider  $x, y \in H_{\log n / (\alpha \log w)}$  with  $|x| \geq |y|$ . Set  $d = |x \oplus y|$ . Then  $|x \setminus y| \geq d/2$ . Also  $|C(y)| \leq 4\alpha \log w \log n / (\alpha \log w) = 4 \log n$ , so

$$\Pr(C(x) = C(y)) \leq \Pr(C(x \setminus y) \subseteq C(y)) \leq 2^{-d/2 - 4\alpha \log w}.$$

Note that the number of pairs in  $A$  is dominated by  $|H_{\log n / (\alpha \log w)}|^2 < w^2 \log n / \alpha \log w = n^2 / \alpha$ . Hence the probability  $C(x) = C(y)$  for some  $x, y \in H_{\log n / (\alpha \log w)}$  with  $|x \oplus y| \geq \log n / \alpha^2 \log w$  is  $< n^2 / \alpha^2 - d/2 - 4\alpha \log w$ , which is  $< 1$  iff  $2/\alpha \cdot \log n < d/2 - 4\alpha \log w$  iff  $d > \log n / \alpha^2 \log w$ . Hence there exists a circuit separating all the pairs. ■

The circuit in Lemma 14 can be implemented by a single instruction  $\text{ReduceHammingBall}(\alpha, n, x)$  with parameters  $\alpha$  and  $n$ . Hence, we have

$$\text{Corollary 15 } H((b, \log n / (\alpha \log w))) = T(8 \log n) + H(b, \log n / (\alpha^2 \log w)) + O(1).$$

$$\text{Lemma 16 } H(w, \log n / (\log w)^2) = O(\log \log \log n - \log \log \log w).$$

**Proof:** Combining Corollary 15 and Lemma 10 gives  $H((b, \log n / (\alpha \log w))) = H(b, \lfloor \log n / (\alpha^2 \log w) \rfloor) + O(1)$ . Applying recursively  $t$  times gives  $H(b, \log n / (\log w \log w)) = H(b, \lfloor \log n / (\log^t w \log w) \rfloor) + O(t)$ . Since  $H(w, 0) = O(1)$ , we are done when  $\log n / (\log w)^{t+1} \leq 1$  i.e. when  $\log \log n \leq (2^t + 1) \log \log w$ . This holds when  $\log \log \log n \leq t + \log \log \log w$  i.e. when  $\log \log \log n - \log \log \log w \leq t$ . ■

### Clustering with range reduction

In order to use Corollary 13 effectively, we need the following lemma, which is proven in this subsection.

**Lemma 17** *There is a clustering function for the reduction*

$$T(w) = O\left(T(w / \log^3 w) + (\log \log m)^{8/9}\right).$$

*The function requires  $O(n)$  additional space.*

**Lemma 18** *There is a parameterized  $AC^0$  function  $f(a) : \{0, 1\}^w \rightarrow \{0, 1\}^{w / \log^3 w}$ , such that for any set  $A \subseteq \{0, 1\}^w$  with at most  $w$  elements, if  $a$  is picked at random,  $f(a)$  is 1-1 on  $A$  with probability at least  $1 - 1/w^2$ .*

**Proof:** We start by proving two facts.

*Fact 1:* Assume that there is an  $AC^0$  circuit  $C(a) : \{0, 1\}^w \rightarrow \{0, 1\}^{w/u}$ ,  $u > 1$ , which, for some parameter of  $a$ , is 1-1 on a set  $Y$  of  $\sqrt{w}$ -bit keys. Then, there is an  $AC^0$  circuit  $C(a) : \{0, 1\}^w \rightarrow \{0, 1\}^{w/u}$  which, for the same parameter  $a$ , is 1-1 for any set of  $w$ -bit keys formed by concatenating keys from  $Y$ .

*Proof:* Duplicate the circuit  $\sqrt{w}$  times.

*Fact 2:* There is a parameterized  $AC^0$  function  $C(a) : \{0, 1\}^w \rightarrow \{0, 1\}^{O(\sqrt{w} / \log^3 w)}$  such that for any set  $A \subseteq \{0, 1\}^w$  of at most  $w^2$  keys, if  $a$  is picked at random,  $C(a)$  is 1-1 on  $A$  with probability at least  $1 - 1/w^2$ .

*Proof:* Let  $b = \sqrt{w}$ . We just alter the proof of Lemma 12 slightly. We use  $b$  instead of  $w$  (at proper places),  $n = w^2$ ,  $r = 1$ , and  $d = 4$ . This gives  $C(a) : \{0, 1\}^b \rightarrow \{0, 1\}^{O(b \log w / (r \log^d b))} = \{0, 1\}^{O(\sqrt{w} / \log^3 w)}$ . In order to use a fixed  $AC^0$  instruction, the parameter  $a$  should represent a  $k \times l$  matrix inside a constant number of words, where  $k = (\log b)^d$  and  $l = O(b \log w / (r \log^d b))$ . Such a representation requires (at most)  $kl \log w$  bits. With our parameters, this number of bits is  $O(w)$ , and the proof of Fact 2 is completed.

Now, perceive each word in  $A$  as divided into  $\sqrt{w}$ -bit pieces. This gives us a set  $Y$  of  $w/\sqrt{w}$  keys. Hence, Fact 2 gives us a circuit  $C(a) : \{0, 1\}^{\sqrt{w}} \rightarrow \{0, 1\}^{O(\sqrt{w} / \log^3 w)}$  which is 1-1 on  $Y$ . Since  $A$  can be constructed by concatenating keys in  $Y$ , we can apply Fact 1 with  $u = \Theta(\log^3 w)$ , which gives us the desired function. ■

$$\text{Lemma 19 } T(b) \leq 2H(b, \sqrt{b \log n}) + O(1).$$

**Proof:** If we apply Lemma 1 with  $r = \sqrt{b \log n}$ , it follows that we can pick a  $b$ -bit word with  $r$  1s such that for any  $x, y \in X$ , if  $|x \oplus y| \geq r$ , then  $x \wedge a \neq y \wedge a$ . Hence, if we use  $x \wedge a$  as a clustering function, the set of signatures is contained in  $H_{\sqrt{b \log n}}$  and each cluster have Hamming radius  $\sqrt{b \log n}$ . ■

**Lemma 20** *There is a clustering function performing the following reduction:*

$$H(b, r) = O\left(H\left(\sqrt{br}, r\right) + T\left(\frac{w}{\log^3 w}\right) + \frac{\log w}{\sqrt{b/r}}\right).$$

*When applied on a set of size  $m$  the function requires  $O(m)$  extra space.*

**Proof:** Let  $X$  be the set to be stored and let  $q = \sqrt{b/r}$ . Perceive each word  $x$  as divided into  $q$ -bit pieces. Let  $x(i) = x[(i-1)q + 1..iq]$  denote the  $i$ th piece. Now define a clustering function  $f_1^q : \{0, 1\}^w \rightarrow \{0, 1\}^{w/q}$  so that  $f_1^q(x)[i] = 1$  iff there is a 1 in  $x(i)$ . Clearly all  $f_1^q$  is implementable by one  $AC^0$  circuit with a log  $w$  input parameter  $q$ . When using  $f_1^q$  as a clustering function, the signatures are  $(b/q)$ -bit keys containing at most  $r$  1s; they can be stored with query time  $H(b/q, r)$ .

Next, we address the problem of storing the clusters. Consider a cluster of size  $m$  and let  $y$  be the signature corresponding to this cluster. Each bit in  $y$  corresponds to a  $q$ -bit field. The cluster have one important property: For each bit which is 0 in  $y$  the corresponding field in all keys in the cluster contains only 0s. The main idea is to shorten the keys by removing these "empty" fields.

**Case 1:**  $m \leq w$ . Applying Lemma 18 gives query cost  $T(w / \log^3 w)$ .

**Case 2:**  $m > w$ . Associated with  $X(y)$ , we store  $F(y) = \alpha_1 \cdots \alpha_{b/q}$  where  $\alpha_i$  is a  $(\log w)$ -bit integer telling the position of the  $i$ th 1 in  $y$ . The maximum number of 1s in  $y$  is  $b/q$ ; if there are less than  $b/q$  1s, then the last  $\alpha_i$ s are set to 0.  $F(y)$  contains  $b/q$   $(\log w)$ -bit integers. Hence,  $F(y)$  occupies  $\alpha(m)$  space. (Furthermore, each  $\alpha_i$  can be generated in constant time, and hence  $F(y)$  can be constructed in  $O(b/q) = \alpha(m)$  time.) We can easily build an  $AC^0$  circuit  $C_2^q : \{0, 1\}^w \rightarrow \{0, 1\}^{w/\log w}$  so that

$$C_2^q(x, \alpha_1 \cdots \alpha_{b/q}) = x(\alpha_1) \cdots x(\alpha_{b/q})$$

Hence, the function

$$f^q(x, \alpha_1 \cdots \alpha_{b/q}) = x(\alpha_1) \cdots x(\alpha_{b/q})$$

can be computed by applying  $C_2^q$   $1 + b \log w / (qw)$  times  $< 1 + \log w / q$  times.

Now given  $x_1, x_2 \in X(y)$ , if  $x_1 \neq x_2$ , then  $f^q(x_1, F(y)) \neq f^q(x_2, F(y))$ . Since each key contains at most  $r$  1s, the keys produced by  $f^q$  are of length  $qr$ . They can be stored at a cost of  $H(qr, r) + O(1 + \log w/q)$ .

In order to get an upper bound on the total query time, we can add the costs of the two cases. ■

$$\begin{aligned} H(b, r) &\leq H(b/n, r) + H(qr, r) + O(1 + \log w/q) + T(w/\log^3 w) \\ &\leq 2H(\sqrt{r}, r) + T(w/\log^3 w) + O\left(1 + \log w/\sqrt{b/r}\right) \end{aligned}$$

**Proof:** (of Lemma 17) Two cases.

Case 1:  $b \leq \log n(\log \log n)^{4/9}$ . Applying Lemma 10 with  $c = b/\log n$  gives  $T(b) \leq O((\log \log n)^{4/9})$ .

Case 2:  $\log n(\log \log n)^{4/9} < b < n$ . Lemma 19 gives

$$T(b) \leq 2H(b, \sqrt{b \log n}) + O(1)$$

Apply Lemma 20 with  $r = \sqrt{b \log n}$ :

$$\begin{aligned} T(b) &\leq 2H(b^{3/4} \log^{1/4} n, \sqrt{b \log n}) \\ &\quad + T(w/\log^3 w) + O\left(1 + \log b/(b/\log n)^{1/4}\right). \end{aligned}$$

The fact that  $\log n(\log \log n)^{4/9} < b$  implies that  $\log n = O(b/\log^{4/9} b)$  and  $\log b/(b/\log n)^{1/4} \leq (\log \log n)^{8/9}$ . This, in turn, gives

$$\begin{aligned} T(b) &\leq 2H(b/\log^{1/9} b, \sqrt{b \log n}) \\ &\quad + T(w/\log^3 w) + O((\log \log n)^{8/9}) \\ &\leq 2T(b/\log^{1/9} b) \\ &\quad + T(w/\log^3 w) + O((\log \log n)^{8/9}) \end{aligned}$$

Now, we get an upper bound on the cost by just taking the maximum of the costs in the two cases. This gives

$$T(b) \leq 2T(b/\log^{1/9} b) + T(w/\log^3 w) + O((\log \log n)^{8/9})$$

The proof is completed by applying this inequality 18 times. Each time, we get  $O(n)$  additive space. Hence, the total space overhead is  $O(n)$ . ■

**Final upper bound**

**Theorem 21** There is an  $AC^0$  RAM solution to the static dictionary problem with query time

$$\begin{aligned} O\left(\min\left(\frac{\log w(\log \log n - \log \log \log w)}{\log \log w}, \sqrt{\frac{\log n}{\log \log n}}\right)\right) \\ = O\left(\min\left(\log w, \sqrt{\log n/\log \log n}\right)\right) \end{aligned}$$

and space  $O(n)$ .

**Proof:** Starting with  $n$   $w$ -bit keys, we use Lemma 17:

$$T(w) = O\left(T(w/\log^3 w) + (\log \log n)^{8/9}\right) \quad (1)$$

We now study the problem of storing keys of length  $b = w/\log^3 w$ . We use Corollary 13 with  $d = 3$  and  $r = \log n/(\log w)^2$ . Since  $b \leq w/\log^3 w$ , we have that  $r \geq b \log n \log w/w$  and we can use a uniform instruction set. This gives us signatures of length  $b/\log b$ , each signature represents a Hamming ball of diameter  $\log n/(\log w)^2$ .

$$T(b) = T(b/\log b) + H(b, \log n/(\log b)^2)$$

Applying Lemma 16 gives

$$T(b) = T(b/\log b) + (\log \log \log n - \log \log \log b). \quad (2)$$

We now distinguish two cases. First, if  $\log b(\log \log \log n - \log \log \log b)/\log \log b \leq \sqrt{\frac{\log n}{\log \log n}}$ , we just apply Equation 2 recursively  $O(\log b/\log \log b)$  times. Since  $b < w$  and since the additional cost of  $O((\log \log n)^{8/9})$  in Equation 1 is negligible, the proof follows for this case.

In the second case  $\log b(\log \log \log n - \log \log \log b)/\log \log b > \sqrt{\frac{\log n}{\log \log n}}$ . In this case,  $\log b = \Omega(\text{polylog } n)$  which implies that  $\log \log \log n - \log \log \log b = O(1)$ . Thus, Equation 2 gives

$$\begin{aligned} T(b) &= T(n, b/\log b) + O(1) = T(n, b/\text{polylog } n) + O(1) \\ \Rightarrow T(b) &= T(n, k) + O\left(\frac{\log \frac{1}{k}}{\log \log n}\right) \end{aligned}$$

Lemma 6 gives

$$T(b) = O\left(\frac{\log n}{\log \frac{1}{k}} + \frac{\log \frac{1}{k}}{\log \log n}\right)$$

We now set  $\log \frac{1}{k} = \sqrt{\log n \log \log n}$ . Again, the proof follows since  $b < w$  and since the additional cost of  $O((\log \log n)^{8/9})$  in Equation 1 is negligible. ■

## 4 Dynamization

We can dynamize the dictionary using essentially the same rebuilding technique and analysis that was used to dynamize perfect hashing [11].

**Theorem 22** On an  $AC^0$  RAM with randomization, we can maintain a dynamic clustering structure supporting insertion, deletion, and search. The space is  $O(n)$ . The worst case query time and the expected amortized insertion and deletion times are

$$\begin{aligned} O\left(\min\left(\frac{\log w(\log \log \log n - \log \log \log w)}{\log \log w}, \sqrt{\frac{\log n}{\log \log n}}\right)\right) \\ = O\left(\min\left(\log w, \sqrt{\log n/\log \log n}\right)\right) \end{aligned}$$

**Proof:** We only need to consider insertions. During deletion, we do not remove keys, we just mark them as deleted. After  $\Theta(n)$  deletions, we reconstruct the entire data structure. (A reconstruction is done incrementally, by inserting the present keys one by one.)

As long as the clustering function works as desired, our functions Find and Insert will work smoothly. But, what happens when an insertion fails? We recall that the clustering structure relies on the fact that each function  $f$  has the desired property: only keys within a



small Hamming radius get the same signature. If, however,  $f$  fails in this aspect, keys with large Hamming distance will be clustered together. This, in turn, will imply that at the bottom of the recursion, when we expected to have a Hamming ball of radius 0, we will end up having more than one key. When this occurs, we can conclude that some clustering function on the way down the recursion has failed. There is no efficient method to check which function failed. Instead, we have the following facts about our clustering functions:

Lemma 1: The probability of a failure is less than  $1/n^3$ .

Lemma 6: The used data structure is deterministic.

Lemma 10: The used data structure relies on perfect hashing. If we replace perfect hashing by the dynamic version [11], the employed trie structure will allow dynamization as desired.

Lemma 12 and Corollary 13: The probability of a failure is less than  $1/n^3$ .

Lemma 14 and Corollary 15: the described circuit is used deterministically.

Lemma 16: Relies on Lemma 10 and Corollary 15.

Lemma 17: Relies on Lemmas 10, 19, and 20.

Lemma 18: The probability of a failure is  $O(1/w^3)$ .

Lemma 19: Relies on Lemma 1 only.

Lemma 20: During the first  $w$  insertions, the probability of failure is the same as in Lemma 18. At the  $w + 1$ st insertion, we change from Case 1 to Case 2 and the entire structure must be reconstructed as the data structure is changed. Fortunately, this reconstruction occurs only once so the amortized cost can be neglected. After that, the construction is deterministic.

Hence, for all clustering functions, except the one used in Lemma 18, the probability of a failure is less than  $1/n^3$ . The clustering function of Lemma 18 is assumed to be 1-1 on a small set of  $m \leq w$  keys; the probability of a failure is  $O(1/w^3)$ . This property can easily be verified in  $O(m^2)$  time by examining all  $m$  keys and their signatures.

Based on these facts, we use the following insertion algorithm:

Whenever an insertion fails, we traverse the clustering structure bottom-up. If there is any clustering function based on Lemma 18, we check if it is 1-1; if needed, we choose a new function until it is 1-1. If this did not help, the failing clustering function is somewhere else. Then, we rebuild the entire global clustering structure from scratch. In both cases, the probability of failure is small enough to guarantee a low expected insertion cost.

When the value of  $n$  has changed dramatically, after  $\Omega(n)$  updates, we throw away our global data structure and construct a new one, the (expected) amortized cost of doing this is also low.

We can also implement ordered dictionaries on an  $AC^0$  RAM in sublogarithmic time.

Corollary 23 On an  $AC^0$  RAM with randomization we can maintain a structure supporting insertion, deletion, and predecessor queries. The space is  $O(n)$ . The worst case query time and the expected amortized insertion and deletion times are

$$O\left(\min\left(\log^2 w, \frac{\log n}{\log w} + \log \log n\right)\right) = O\left(\log^{2/3} n\right).$$

**Proof: (sketch)** A van Emde Boas tree supports the operations in  $O(\log w)$  time. It require much space, which can be reduced to linear by hashing. Replacing hashing with the result of Theorem 22 we get a multiplicative overhead of  $O(\log w)$ . This gives the  $O(\log^2 w)$  bound.

Next, we note that fusion trees can be implemented with  $AC^0$  instructions only [5]. This, combined with exponential search trees [4], immediately gives a bound of  $O(\log n / \log w + \log \log n)$ .

Previous sub-log  $n$  solutions to the dynamic predecessor problem have used either multiplication or super-linear space (see [3] for a discussion).

It is possible to derandomize the dynamic ordered dictionary structure above, paying only a slight increase in time. We shall only sketch this. The key observation is that standard derandomization techniques (conditional probabilities) makes it possible to find the appropriate clustering functions for the static dictionary in deterministic time  $(nw)^{O(1)}$ . This implies that an (unordered) static dictionary with query time  $O(\log^2 w)$  can be constructed in deterministic time  $n^{O(1)}$ . To see this we study two cases. (i)  $n > w$ . Then, the construction cost is  $(nw)^{O(1)}$  and the query cost follows from Theorem 21. (ii)  $n \leq w$ . In this case, using the methods from [4] and [5], we can construct a static dictionary (consisting of a fusion tree with constant height); the construction cost is  $n^{O(1)}$  and the query cost is  $O(1)$ .

Now, we can use an exponential search tree [4] where each node either contains (i) a trie where each trie-node is stored with  $O(\log w)$  access cost; this a query cost of  $O(\log^2 w)$  within the node of the exponential search tree; or (ii) a static fusion tree; this gives a cost of  $O(\log d / \log w)$  in a node of degree  $d$  in the exponential search tree. Following the proof technique in [4], the cost of searching in the exponential search tree becomes  $T(n)$  where

$$T(n) = O\left(\min\left(\log^2 w, \frac{\log n}{\log w} + 1\right)\right) + T(n^c)$$

for some constant  $c < 1$ . This gives (roughly) the bound

$$O\left(\min\left\{\begin{array}{l} \log^{2/3} n \\ \frac{\log n}{\log w} + \log \log n \\ \log^2 w \log \log n \end{array}\right.\right)$$

time per operation. The cost for searching is worst-case while the cost for updates is amortized.

### Grouping and sorting

We address the problem of sorting  $n$  words on an  $AC^0$  RAM using  $O(n)$  space and  $o(n \log n)$  time. Previous sub- $n \log n$  solutions either used multiplication [14] or super-linear space [2]. As a subroutine in our solution, we consider the problem of grouping duplicates, i.e. producing a permutation of the sequence where identical words are consecutive. Note that a sorting algorithm is also a grouping algorithm, but that the converse does not necessarily hold. From [2], we do have:

Lemma 24 On an  $AC^0$  RAM, if we can identify duplicates among  $n$  keys in time  $D(n)$  and space  $S(n)$ , we can sort  $n$  keys in time  $O(D(n) \log \log n)$  and space  $S(n)$ .

The simplest way of grouping duplicates is to insert the keys one by one using a dynamic dictionary, using Theorem 22. Below, we give a faster method.

As pointed out in [2], the results from [1] imply

Lemma 25 On an  $AC^0$  RAM, we can sort  $n$  keys of length  $w / (\log n \log \log n)$  in time  $O(n)$  and space  $O(n)$ .



**Theorem 26** We can find and group duplicates in a set of size  $n$  in  $O(n \log \log n)$  expected time by a randomized  $AC^0$  RAM algorithm using linear space.

**Proof:** If  $n \geq 2^{9\sqrt{w}}$ , we can immediately apply Theorem 22. Thus, we may assume that  $n < 2^{9\sqrt{w}}$ .

We wish to reduce our grouping problem into two simpler grouping problems. First we apply Lemma 1 with a random  $K$ , mapping our keys into signatures of length  $10(\log n)w/\tau$ , where  $\tau > 10(\log n)(\log w)$ . Our first grouping problem is that of grouping the signatures. Thereby the original keys get grouped into Hamming balls of radius  $\tau$ . To each such Hamming ball, we apply Lemma 3, mapping the keys into keys of size  $10\tau^2(\log w)$ . Grouping these keys finalizes the grouping of the original keys. Trivially, the grouping problem from each Hamming ball can be reduced to one grouping problem of  $n$  keys. Thus, the above construction shows that the grouping problem of  $n$   $w$ -bit keys can be reduced one grouping problem of  $n(10(\log n)w/\tau)$ -bit keys, and one grouping problem of  $n(4\tau^2 \log w)$ -bit keys.

Setting  $\tau = \sqrt[3]{w(\log n)/(2 \log w)}$ , we get two grouping problems for keys of the same length. Note that  $\tau > 10(\log n)(\log w)$  since we have assumed  $n < 2^{9\sqrt{w}}$ .

The key length of our two grouping problems is  $\Theta(w^{2/3}(\log n)^{2/3}(\log w)^{1/3})$ . Since  $n < 2^{9\sqrt{w}}$ ,  $(\log n)^{2/3}(\log n \log \log n) = o(w^{1/3}/(\log w)^{1/3})$ , so

$$\Theta(w^{2/3}(\log n)^{2/3}(\log w)^{1/3}) = O(w/(\log n \log \log n)).$$

Hence each can be solved in linear time by Lemma 25. That is, for  $n < 2^{9\sqrt{w}}$ , the duplicate grouping problem of  $n$   $w$ -bit keys, is solved in linear time and linear space. ■

Combining with Lemma 24, we get

**Corollary 27** We can sort in linear space and expected time  $O(n(\log \log n)^2)$  on an  $AC^0$  RAM with randomization.

A monotone priority queue is a priority queue with non-decreasing minimum, as is the case for applications within greedy algorithms. Modifying a reduction from [23], we derive

**Corollary 28** There is a monotone  $AC^0$  priority queue supporting find-min in constant time and insert and delete in expected amortized time  $O((\log \log n)^2)$ .

Recently, Thorup has improved the above bounds to  $O(n \log \log n)$  and  $O(\log \log n)$ , respectively [24]. His algorithm is much more complicated, but has the advantage of only using standard  $AC^0$  operations; namely, addition, shift, and bit-wise boolean operations.

## 5 Lower bounds

In order to present our lower bounds, we introduce a new model of computation, the circuit computation tree. This model does not have a notion of a random access memory - the static data structure is built into the tree.

A circuit computation tree  $\mathcal{T}$  is a rooted tree, taking as input a word  $x$  of length  $w$ . Each internal node  $v$  contains exactly  $B$  gates,  $B \geq w$  being a parameter called the breadth of the tree. Each gate  $v$  is a  $\wedge$ - or  $\vee$ -gate taking as input an arbitrary subset of the gates in the parent node, the input variables  $x[i]$ , and their negations  $\neg x[i]$ . A given subset of the gates in  $v$  of size  $s$  are called selection gates. Here  $s \leq B$  is another parameter, we call  $s$  the selection length of  $\mathcal{T}$ . All internal nodes have exactly  $2^s$  children corresponding to each of the possible instantiations of the selection gates. Thus any instantiation  $x$  of the input word defines a unique path from the root to some leaf, called the leaf of  $x$ . Each leaf is labelled either 0 or 1. The label on the leaf of  $x$  is the value  $\mathcal{T}(x)$  of the computation by  $\mathcal{T}$  on  $x$ .

**Proposition 29** Let  $\phi$  be a data structure of size  $s$  and  $P$  be a circuit RAM program  $P$  with worst case running time  $t$ , returning 0 or 1 on each input  $x \in \{0, 1\}^w$ . There is a circuit computation tree  $\mathcal{T}$  of breadth  $w^{O(1)}$ , depth  $O(t)$  and selection length  $\log s$  so that  $\mathcal{T}(x) = P(x, \phi)$  for all  $x \in \{0, 1\}^w$ .

**Proof:** (sketch) Straightforward simulation, we simply "fold out" all possible computations of the program. The branching is used to simulate conditional jumps and indirect addressing. An interesting point is that the simulation holds even if the query program is allowed to use any amount of space additional to the data structure. In that case, we simulate indirect read in a constant number of steps, first checking (with an  $AC^0$  circuit) if we read a memory location where we already wrote something, and if not, then simulating reading in the data structure. ■

Proposition 29 shows that in order to show the deterministic version of the desired lower bounds for the Circuit RAM, it is sufficient to construct a set which cannot be decided by any circuit computation tree with certain parameters. In order to get the randomized lower bound, we shall do something stronger. We shall construct a set consisting of pairs, such that no circuit computation tree with certain parameters can separate more than a small fraction of the pairs. By a simple averaging argument, this will give the desired randomized lower bounds for the problem of storing the set consisting of first components of the pairs.

Our lower bound proof uses Hästad's switching Lemma [15]. Rather than the original version, we use the following slight variation, which appears as Lemma 1 in [6]. This version has the advantages of leaving a predetermined number of variables unset and yielding a decision tree rather than a DNF formula. Recall that a restriction on  $m$  variables is a map  $\rho: \{1, \dots, m\} \rightarrow \{0, 1, *\}$ . If  $\rho(i) = *$ , we say that the  $i$ 'th variable is unset. We can apply the restriction to a function  $f$  with domain  $\{0, 1\}^m$ . This yields a function  $f^\rho$  on a smaller domain, namely the subdomain of  $\{0, 1\}^m$  matching the restriction. Two restrictions are said to be disjoint if they set some bit differently, implying that the restricted domains they specify are disjoint. We call a function semi- $k$ -simple if it is expressible as a DNF or a CNF with term size  $k$ . A function is said to be  $k$ -simple if it is decidable by a decision tree of depth  $k$ . Note that if  $f$  is a conjunction or disjunction of  $k$ -simple functions, then  $f$  is semi- $k$ -simple. The other direction does not hold but the switching lemma gives us something almost as good:

**Lemma 30 (Hästad's Switching Lemma)** Let  $f: \{0, 1\}^m \rightarrow \{0, 1\}$  be semi- $k$ -simple. For  $s \geq 0$ ,  $p \leq 1/7$ , we have that with probability at least  $1 - (7pk)^s$ , for a random restriction  $\rho$  leaving exactly  $pm$  variables unset,  $f^\rho$  is  $s$ -simple.

**Lemma 31** Let  $f_1, \dots, f_s: \{0, 1\}^m \rightarrow \{0, 1\}$  be  $k$ -simple. Let  $\rho$  be a random restriction which leaves  $r$  free variables. Then,

$$\Pr[\exists i: f_i^\rho \text{ is not constant}] \leq rks/m$$

**Proof:** First, randomly fix all the  $w$  input bits, thereby fixing all the  $f_i$ . Since each  $f_i$  is  $k$ -simple, there is a set  $X_i$  of the input bits of size  $\leq k$ , whose current instantiation fixes  $f_i$ . As a result, all  $f_i$  are determined by the current instantiation of the input bits in  $X = \bigcup_i X_i$ , where  $|X| \leq sk$ .

Now, randomly unfix a set  $Y$  of  $r$  variables. If  $Y$  does not intersect  $X$ , the output is still determined by the instantiation of the bits in  $X$ . Since  $Y$  is random the probability that  $Y$  intersects  $X$  is bounded by  $rks/w$ . ■

In the following discussion we fix  $B \geq w \geq s \geq 10$  and let  $k = \log B$ .

**Theorem 35** For values  $w, B, d, s$  with  $B \geq w \geq 100$ ,  $w \geq (120sk^4)^d$ , and  $d < \kappa/2$ , there is a set  $P(w, B, d, s) \subseteq \{0, 1\}^w \times \{0, 1\}^w$  of size  $n = B^{4d}$  with the following property. No circuit computation tree  $\mathcal{T}$  of breadth  $B$  and degree  $2^S$  separates more than a fraction  $2d/\kappa^2$  of the pairs in  $P(w, B, d, s)$ .

**Proof:** The set  $P(w, B, d, s)$  is defined recursively in  $d$ :

- $P(w, B, 0, s) = \{(0^w, 1^w)\}$ .
- $P(w, B, d, s)$  for  $d > 0$  is defined as follows. Pick the sequence of  $B^4$  restrictions as in Lemma 34. For each restriction  $\rho_i$ , there are  $w/(120sk^2)$  unset variables. Choose fixings for these variables by going through all the elements of  $P(w/(120sk^4), B, d-1, s)$  (which by induction is already defined). This yields a set  $P_i$ . Let  $P(w, B, d, s)$  be the union of all the  $P_i$ 's.

We now show, by induction in  $d$ , that  $P(w, B, d, s)$  has the right property. To make the induction roll, we show the statement for a slightly larger class of objects than circuit computation trees, namely augmented circuit computation trees. An augmented circuit computation tree of breadth  $B$ , height  $d$ , and degree  $2^S$  is defined as a circuit computation tree, except that each gate in the root may get inputs from at most  $B$  precomputed  $\kappa$ -simple values ( $\kappa = \log B$ ) in addition to input variables and their negations.

For  $d = 0$ , the theorem clearly holds. A circuit computation tree of depth 0 (a leaf) separates no pairs.

Now assume  $d > 0$ , and assume the theorem holds for smaller  $d$ . Let  $\mathcal{T}$  be a circuit computation tree of depth  $d$ . By Lemma 34, at least a  $1 - 2/\kappa^2$  fraction of the  $\rho_i$  restrictions make the selection gates in the root constant and all other gates in the root  $\kappa$ -simple. Take one of those restrictions,  $\rho_i$ . Apply it to  $\mathcal{T}$ . The selection gates of the root now defines a constant vector, say  $c$ . Follow the  $c$ -branch out of the root and consider the subtree found there. It is an augmented circuit computation tree of breadth  $B$ , depth  $d-1$ , and selection length  $s$ , so it separates at most a  $2(d-1)/\kappa^2$  fraction of the pairs of  $P_i = \rho_i(P(w, B, d, s))$ . But if the restricted subtree does not separate the restriction of a particular pair, the original subtree does not separate the original pair. Thus, for each of the sets  $P_i$  for which  $\rho_i$  has the desired property,  $\mathcal{T}$  separates at most a  $(d-1)/\kappa^2$  fraction of  $P_i$ . Thus, the total fraction of pairs in  $P(w, B, d, s)$  separated by  $\mathcal{T}$  is at most  $2/\kappa^2 + 2(d-1)/\kappa^2 = 2d/\kappa^2$ . ■

Theorem 35 and Proposition 29 now gives us all the lower bounds we want. For instance, to show a lower bound on deterministic worst case time, we consider storing the set  $P_i$  of first components of  $P = P(w, B, d, s)$  for appropriately chosen values of the parameters. If a tree does not separate the first and second components, it cannot decide  $P_i$ , since the first and second components should give different outputs. If we want to show a lower bound on the average query time, we again consider storing the set  $P_i$ . Assume that the average query time (over  $P_i$ ) is  $t$  and chop off the computation after  $2t$  steps. At least half the inputs in  $P_i$  must have produced an output by then. But this means that the tree has separated half the corresponding pairs, if it is correct. This gives a circuit computation tree separating half the pairs in  $P$ , a contradiction. If we want to show a lower bound on the query time of Monte Carlo schemes, we again consider storing the set  $P_i$  itself. In a solution, freeze the random choices made, so that we get a deterministic solution, giving the correct answer on  $2/3$  of the pairs of  $P$ . Such a solution must separate at least  $1/3$  of the pairs of  $P$ , a contradiction.

To get the lower bounds in the Introduction, we only have to set the parameters right. To get Theorem B, we fix some  $\epsilon > 0$ ,  $c > 1$  and let  $n = w^{4c/\epsilon}$ ,  $B = w^c$ ,  $d = 1/2\epsilon$ , and  $s = w^\epsilon$ . The set  $P_i$  of first components of  $P(w, B, d, s)$  has size  $n = w^{4c/\epsilon}$ , and Theorem 35 and Proposition 29 now gives Theorem B.

To get the lower bound parts of Theorem A and C, let  $B = w^c$ ,  $s = (\log w)^c$ , and  $d = \frac{\log w}{4\epsilon \log \log w}$ . The set  $P_i$  of first components of  $P(w, B, d, s)$  has size  $n = w^{\log w / \log \log w}$ , and Theorem 35 and Proposition 29 now gives us the lower bound parts of Theorem A and C.

**Lemma 32** Let each of  $f_1, \dots, f_B : \{0, 1\}^w \rightarrow \{0, 1\}$  be expressed as an  $\wedge$ -gate or an  $\vee$ -gate taking a number of  $\kappa$ -simple functions as input. Let  $\rho$  be a random restriction leaving  $w/(120sk^4)$  input bits free. Then with probability  $\geq 1 - 1/\kappa^2$ ,  $f_1^{\rho}, \dots, f_B^{\rho}$  are constant and  $f_{S+1}^{\rho}, \dots, f_B^{\rho}$  are  $\kappa$ -simple.

**Proof:** First, we apply a random restriction  $\rho'$  from Lemma 30 with  $s = k = \kappa$  and  $p = 1/(30\kappa)$ . This leaves  $w/(30\kappa)$  input variables free. For any specific  $j$ ,  $f_j^{\rho'}$  is  $\kappa$ -simple with probability at least  $1 - 1/B$ . Thus with probability at least  $1 - 1/B \geq 1 - 1/2\kappa^2$ , all the  $f_j^{\rho'}$ 's are  $\kappa$ -simple. Now, apply Lemma 31 with  $m = w/(30\kappa)$  and  $r = w/(120sk^4)$  to get a random restriction on the remaining free variables. With probability at least  $1 - \tau ks/m = 1 - 1/4\kappa^2$ , the composition  $\rho$  of the two restrictions has  $f_1^{\rho}, \dots, f_S^{\rho}$  constant. ■

At first sight, knowing that a random restriction simplifies a node in a computation tree does not seem particularly helpful. In order to get the lower bounds we want, we have to consider computation trees deciding quite a small set, and a random restriction is likely to wipe out such a set completely. We now show how we can use the probabilistic method to replace the random restriction by one from a small, fixed, family, and thereby circumvent this problem.

We shall use a Chernoff bound argument. The following version of the Chernoff bound can be found in [21].

**Lemma 33** Let  $X_1, X_2, \dots, X_m$  be independent Bernoulli trials with  $\Pr\{X_i = 1\} = p$ . Let  $X = \sum X_i$  and let  $\mu = E[X] = mp$ . Then  $\Pr\{X > 2\mu\} < e^{-\mu/3}$ .

**Lemma 34** There is a fixed family of pairwise disjoint restrictions  $\rho_1, \dots, \rho_B$  on  $w$  variables leaving free  $w/(120sk^4)$  bits such that for any choice of functions  $f_1, \dots, f_B$ , each being expressed as an  $\wedge$ -gate or an  $\vee$ -gate with at most  $2B$  inputs, each input being some  $\kappa$ -simple function, there is at least a fraction  $1 - 2/\kappa^2$  of the possible values of  $i \in \{1, \dots, B\}$  such that  $f_1^{\rho_i}, \dots, f_B^{\rho_i}$  are constant and  $f_{S+1}^{\rho_i}, \dots, f_B^{\rho_i}$  are  $\kappa$ -simple.

**Proof:** The restrictions  $\rho_1, \dots, \rho_B$  will be chosen randomly and independently, and then we will show that they have the desired properties with non-zero probability.

By Lemma 32, for a fixed choice of the  $f_i$ 's each  $\rho_i$  does not work with probability at most  $1/\kappa^2$ . Assume without loss of generality (a smaller probability is clearly in our favor) that the probability is exactly  $1/\kappa^2$ . The expected number of  $\rho_i$  that do not work is thus  $B/\kappa^2$ . By Lemma 33 the probability that more than a  $2B/\kappa^2 = 2\mu$  do not work is less than  $e^{-\mu/3} = e^{-B^4/(3 \log^2 B)}$ . However, there are at most  $wB \leq 2B \log B$  different choices for each input to each  $f$  (each choice corresponds to a decision tree of depth  $\kappa = \log B$ ), so there are at most  $2^{B \log B} (2B)^B = 2^{2B^3 \log B} = o(e^{-B^4/(3 \log^2 B)})$  different possible families of  $f_i$ 's. Hence, with probability  $1 - o(1)$ ,  $\rho_1, \dots, \rho_B$  have the desired property, that for every family there is a sufficient number of the  $\rho_i$ 's that work.

Next, note that for two random restrictions leaving less than a fraction  $1/120$  of the variables free, at least a fraction  $118/120$  of the variables are fixed on both of them. On each such variable, they disagree with probability  $1/2$ . Thus, two restrictions are not disjoint with probability at most  $2^{-\frac{118}{120} \cdot 120w}$ . Therefore, all the restrictions fail to be pairwise disjoint with probability at most  $(B^4)^2 \cdot 2^{-\frac{118}{120} \cdot 120w}$  but since we can assume  $w \geq 120s(\log B)^2$  (otherwise the statement of the lemma is vacuous), the latter value is negligible. We conclude that the random family of restrictions has the right properties with positive probability, and that a good family of restrictions therefore exists. ■

We say that a tree  $\mathcal{T}$  separates  $(x, y) \in \{0, 1\}^w \times \{0, 1\}^w$  if  $x$  and  $y$  do not have the same leaf in  $\mathcal{T}$ .

## References

- [1] 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.
- [2] 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.
- [3] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. FOCS, 1995*, pages 135-141.
- [4] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. FOCS, 1996*, instructions only, Manuscript, 1996.
- [5] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented using  $AC^0$  instructions only, Manuscript, 1994.
- [6] P. Beaune. A switching lemma primer. Manuscript, 1994.
- [7] P.W. Beane, S.A. Cook, H.J. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994-1003, 1986.
- [8] J.L. Carter and M.N. Wegman. Universal classes of hash functions, *J. Comp. Syst. Sci.* 18 (1979), 143-154.
- [9] A.K. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13:423-439, 1984.
- [10] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Technical Report 513, Fachbereich Informatik, Universität Dortmund, 1993.
- [11] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, and F. Meyer auf der Heide. Dynamic perfect hashing: Upper and lower bounds. In *Proc. 29th IEEE Symp. Foundations of Computer Science*, pages 524-531, 1988.
- [12] F. Fich and P.B. Miltersen. Tables should be sorted (on random access machines). In *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, pages 482-493, 1995.
- [13] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. Ass. Comp. Mach.*, 31:538-544, 1984.
- [14] 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.
- [15] J. Hästad. *Computational Limitations of Small-Depth Circuits*. ACM doctoral dissertation award. MIT Press, 1987.
- [16] R. Karlsson. *Algorithms in a Restricted Universe*. Ph.D. Thesis, University of Waterloo, Canada, 1984.
- [17] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching* Addison-Wesley, 1973.
- [18] N. Liniial, Y. Mansour, and N. Nisan. Constant depth circuits, Fourier transform, and learnability. *Journal of the ACM*, 40(3):607-620, July 1993.
- [19] Y. Mansour, N. Nisan, and P. Tiwari. The computational complexity of universal hashing. *Theoretical Computer Science*, 107:121-133, 1993.
- [20] P.B. Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. ICALP '96, to appear.
- [21] C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
- [22] R.E. Tarjan and A.C. Yao. Storing a sparse table. *Communications of the Ass. Comp. Mach.*, 22(11):606-611, November 1979.
- [23] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59-67, 1996.
- [24] 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, New Orleans, 1997.
- [25] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and Systems Sciences*, 28:379-394, 1984.
- [26] A.C. Yao. Should tables be sorted? *J. Ass. Comp. Mach.*, 28:615-628, 1981.



## Recent BRICS Report Series Publications

- RS-97-14 Arne Andersson, Peter Bro Miltersen, Søren Riss, and Mikkel Thorup. *Dictionaries on AC<sup>0</sup> RAMs: Query Time  $\Theta(\sqrt{\log n / \log \log n})$  is Necessary and Sufficient*. June 1997. 18 pp. Appears in *37th Annual Symposium on Foundations of Computer Science, FOCS '96 Proceedings*, pages 441–450.
- RS-97-13 Jørgen H. Andersen and Kim G. Larsen. *Compositional Safety Logics*. June 1997. 16 pp.
- RS-97-12 Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro. *Trans-Dichotomous Algorithms without Multiplication — some Upper and Lower Bounds*. May 1997. 19 pp. Appears in Dehne, Rau-Chaulin, Sack and Tamassio, editors, *Algorithms and Data Structures: 5th International Workshop, WADS '97 Proceedings*, LNCS 1272, 1997, pages 426–439.
- RS-97-11 Kārlis Čerāns, Jens Chr. Godskesen, and Kim G. Larsen. *Timed Modal Specification — Theory and Tools*. April 1997. 32 pp.
- RS-97-10 Thomas Troels Hildebrandt and Vladimiro Sassone. *Transition Systems with Independence and Multi-Ares*. April 1997. 20 pp. Appears in Peled, Pratt and Holzmann, editors, *DIMACS Workshop on Partial Order Methods in Verification*, POMIV '96, pages 273–288.
- RS-97-9 Jesper G. Henriksen and P. S. Thiagarajan. *A Product Version of Dynamic Linear Time Temporal Logic*. April 1997. 18 pp. Appears in Mazurkiewicz and Winkowski, editors, *Concurrency Theory: 8th International Conference, CONCUR '97 Proceedings*, LNCS 1243, 1997, pages 45–58.
- RS-97-8 Jesper G. Henriksen and P. S. Thiagarajan. *Dynamic Linear Time Temporal Logic*. April 1997. 33 pp.
- RS-97-7 John Hatcliff and Olivier Danvy. *Thunks and the  $\lambda$ -Calculus (Extended Version)*. March 1997. 55 pp. Extended version of article to appear in the *Journal of Functional Programming*.