

# Constructing Efficient Dictionaries in Close to Sorting Time

Milan Ružić  
ITU Copenhagen  
milan@itu.dk

## Abstract

The dictionary problem is among the oldest problems in computer science. Yet our understanding of the complexity of the dictionary problem in realistic models of computation has been far from complete. Designing highly efficient dictionaries without resorting to use of randomness appeared to be a particularly challenging task. We present solutions to the static dictionary problem that significantly improve the previously known upper bounds and bring them close to obvious lower bounds. Our dictionaries have a constant lookup cost and use linear space, which was known to be possible, but the worst-case cost of construction of the structures is proportional to only  $\log \log n$  times the cost of sorting the input. Our claimed performance bounds are obtained in the word RAM model and in the external memory models; only the involved sorting procedures in the algorithms need to be changed between the models.

## 1 Introduction

*Dictionaries* are among the most fundamental data structures. A dictionary stores a set  $S$  which may be any subset of *universe*  $U$ , and it answers membership queries of type “Is  $x$  in  $S$ ?”, for any  $x \in U$ . The elements of  $S$  may be accompanied by *satellite data* which can be retrieved in case  $x \in S$ . The size of the set  $S$  is standardly denoted by  $n$ .

We consider universes whose elements can be viewed as integers or binary strings. In this chapter we concentrate on *static* dictionaries — a static dictionary is constructed over a given set  $S$  that remains fixed. *Dynamic* dictionaries allow further updates of  $S$  through insertions and deletions of elements. Even static dictionaries are sometimes used as stand-alone structures, but more often they appear as components of other algorithms and data structures, including dynamic dictionaries.

The dictionary problem has been well studied. Many solutions have been given, having different characteristics regarding space usage, time bounds, model of computation, and universe in question. A challenge is to simultaneously achieve good performance on all the terms. We consider only dictionaries with realistic space usage of  $O(n)$  registers of size  $\Theta(\log |U|)$  bits. In the usual case when  $|U|$  is at least polynomially larger than  $n$ , this amount of space is necessary (ignoring constant factors) regardless of presence of satellite data. Algorithms involved in construction of a dictionary may be randomized — they require a source of random bits and their time bounds are either *expectations* or hold with high probability. Randomized dictionaries reached a stage of high development and theoretically there is little left to be improved. On the other hand, the progress on deterministic dictionaries was much slower. While in the dynamic case we have some reason to believe that there is a considerable gap between attainable worst-case performance for deterministic

dictionaries and the attainable expected performance for randomized dictionaries, there is not any evidence of a required gap in the static case.

A theoretical interest in deterministic dictionaries comes from the question of what resources are necessary to implement an efficient dictionary structure, and random bits are a resource. Having guaranteed time bounds, deterministic structures can be used in systems with strict performance demands. A sufficiently simple deterministic dictionary having comparable performance to a randomized dictionary would make the randomized structure obsolete. Unfortunately, the new solutions described here are not simple enough to be competitive in practice, except possibly in some special cases.

In this chapter we focus on dictionaries with constant lookup time. Because of faster construction time, dictionaries with slightly slower lookups may sometimes be of interest. For example, a structure supporting searches in time  $O(\log \log n)$  can be built in linear time on sorted input [Ruž09].

## 1.1 The word RAM model and related work

The word RAM is a common computational model in data structures literature. It has the machine word size of  $w$  bits and a standard instruction set, resembling the primitive instructions of the language C. Execution of any instruction takes one unit of time. A usual assumption for RAM dictionaries is that the elements of  $U$  fit in one machine word.<sup>1</sup> Contents of a word may be interpreted either as an integer from  $\{0, \dots, 2^w - 1\}$  or as a bit string from  $\{0, 1\}^w$ . For more information see, e.g., [Hag98].

We will list some important results for deterministic dictionaries with constant query time. Each of those results required a different idea, and a new insight into properties and possibilities of some family of hash functions. A seminal work by Fredman, Komlós, and Szemerédi [FKS84] showed that in the static case it is possible to construct a linear space dictionary with a constant lookup time for arbitrary word sizes (no assumptions about relative values of  $w$  and  $n$ ). This dictionary implementation is known as the *FKS scheme*. Besides a randomized version with expected  $O(n)$  construction time, they also gave a deterministic construction algorithm with a running time of  $O(n^3w)$ . A bottleneck was choosing of appropriate hash functions. Any *universal* family of hash functions [CW79] contains functions suitable for use in the FKS scheme. Raman [Ram96] devised a deterministic algorithm for finding good functions from a certain universal family running in time  $O(n^2w)$ ; this implies the same time bound for construction of the FKS dictionary. For  $w = n^{\Omega(1)}$ , an efficient static dictionary can be built in time  $O(n)$  on a sorted sequence of keys. This follows from a generalization of the fusion trees of Fredman and Willard [FW93], and it was observed by Hagerup [Hag98]. The previously fastest deterministic dictionary with constant lookup time is a result of Hagerup, Miltersen and Pagh [HMP01]. Their construction method has a running time of  $O(n \log n)$ . There exists an issue with compile-time computation of a special constant that is required for each  $w$ , because the only known computation method is a brute-force search that takes time  $2^{\Omega(w)}w$ .

Allowing randomization, the FKS scheme can be dynamized to support updates in amortized expected constant time [DKM<sup>+</sup>94]. The lower bound result in the same paper states that a deterministic dynamic dictionary, based on *pure* hashing schemes, with worst-case lookup time of  $t(n)$  must have amortized insertion time of  $\Omega(t(n) \cdot n^{1/t(n)})$  (this lower bound does not hold in

---

<sup>1</sup>This assumption simplifies analysis. Some schemes, including ours, scale well when keys are multi-word strings.

general, e.g. see the result of Pagh [Pag00]). A standard dynamization technique [OvL81] applied to the static dictionary from [HMP01] yields a similar type of trade-offs: lookups in time  $O(t(n))$ , insertions in time  $O(n^{1/t(n)})$ , and deletions in time  $O(\log n)$ , where  $t$  is a “reasonable” parameter function. The method in [Ruž08b] was devised as an alternative to the method from [HMP01] that eliminates the problem with the high compile-time demand. The dynamic dictionaries from [Ruž08b] almost match the dynamic result from [HMP01] — the difference is that update bounds are amortized, instead of worst-case. An illustration of complete independence from the word size  $w$  is that the structures from [Ruž08b] can be easily adapted to the *Real RAM* model to work with arbitrary real numbers. This is not a feature of any other known hashing method.

## 1.2 External memory models

Real computers don’t have one plain level of memory but a memory hierarchy. Transfers of data between levels of memory are often a dominant term in execution times. The theoretical I/O-model was introduced to model behavior of algorithms in such a setting [AV88]. The I/O-model was generalized to the cache-oblivious model [FLPR99], where the algorithm does not know the size of the internal memory  $M$  and the block size  $B$ . That is, the analysis of an algorithm should be valid for any values of  $B$  and  $M$ . Comparison-based sorting of  $n$  integers (which occupy one memory cell) takes  $\Theta(\text{Sort}(n))$  I/Os [FLPR99], where  $\text{Sort}(n) = \frac{n}{B} \log_{M/B} \frac{n}{B}$ .

From the structures mentioned before for the word RAM model, it can be observed that the methods of Raman [Ram96] and Ružić [Ruž08b] can easily be adapted to the external memory models and attain analogous bounds, respectively  $O(\frac{n^2}{B} \log |U|)$  and  $O(\frac{n^{1+\epsilon}}{B})$  I/Os. We take the block size parameter  $B$  to represent the number of  $\log |U|$ -bit items that can fit in a memory block. For the dictionary from [HMP01], no better bound than  $O(n \log n)$  I/Os can be stated. The main problem for I/O performance was the dictionary for universes of size polynomial in  $n$ , which is a component of the construction from [HMP01].

## 1.3 Background of our techniques

Our contribution consists of two parts. One part is a very efficient dictionary for universes of size  $n^{O(1)}$ . Beside its use in composition with methods that perform *universe reduction*, this case has a significance of its own. The most prominent example of stand-alone use of dictionaries for “small” universes is representation of a graph. In the problem of storing and (random) accessing edges of a graph, the universe is of quadratic size. The problem is also of interest to some situations in practice, since in reality integer keys are not often very large relative to  $n$ . The main part of our structure uses the same kind of hash functions that were used in [HMP01] for this case. Interestingly, those functions are very similar to the functions from [TY79] where construction time was  $\Theta(n^2)$ . The construction algorithm from [HMP01] runs in time  $\Theta(n \log n)$ . We devised a different and more efficient construction algorithm.

The other part of the contribution is a follow-up on our technique of making deterministic signatures from [Ruž09]. That paper introduced a new type of hash functions and associated algorithms for injectively mapping a given set of keys to a set of signatures of  $O(\log n)$  bits. The methods are computationally efficient in various models of computation, especially for keys of medium to large lengths. More precisely, when given keys have a length of at least  $\log^{3+\epsilon} n$  bits, the algorithms for selecting perfect hash functions have a linear running cost on sorted input. Those functions have rather succinct descriptions, and they might have an application outside of

dictionary structures. In our quest for a faster construction in the case  $w = \log^{O(1)} n$  we will give up the requirement of complete injectiveness, and replace it with considerably weaker and rather specific properties. These weaker functions will be meaningful only within our dictionary construction.

## 1.4 Our results

The result for the case of universes of polynomial size is summarized in the following theorem.

**Theorem 1.** *Let  $S$  be any given set of  $n$  integers from the universe  $\{0, 1, 2, \dots, n^{O(1)}\}$ . In the word RAM model, in time  $O(n \log \log n)$  it is possible to deterministically construct a static dictionary over  $S$  that performs lookups in constant time and occupies linear space. In the cache-oblivious model, and hence in the I/O model as well, a similar structure can be built using  $O(\text{Sort}(n) \log \log n)$  I/Os.*

The method is discussed in Section 2. The given structure complements additional results from [Ruž09] in the external memory setting, such as a static predecessor structure for variable and unbounded length binary strings.

In the second part of the paper (Section 3) we describe the structures and associated procedures that are efficient when  $w = \log^{O(1)} n$ . In conjunction with the earlier results, this implies the claimed results for the general case, which are formally expressed in the following theorems. In the performance bounds we plugged in the currently known upper bounds on sorting (which may be optimal).

**Theorem 2.** *In the cache-oblivious model, a static linear space dictionary on a set of  $n$  keys can be deterministically constructed using  $O(\text{Sort}(n) \log \log n)$  I/Os, so that lookups to the dictionary take  $O(1)$  I/Os.*

**Theorem 3.** *In the word RAM model, a static linear space dictionary on a set of  $n$  keys can be deterministically constructed in time  $O(n(\log \log n)^2)$ , so that lookups to the dictionary take time  $O(1)$ .*

We could have also listed results for strings, etc. The stated general bounds do not match the actual times in every case. We make remarks on some meaningful special cases, when performance is better.

**Remark 1.** *Suppose that  $\log |U| = \Omega(\log n \log \log n)$ . The construction cost of the dictionary referred to in Theorem 2 is  $O(\text{Sort}(n))$  I/Os.*

**Remark 2.** *Supposing that  $w = O(\log n \log \log n)$ , the time taken to build the dictionary from Theorem 3 is  $O(n \log \log n)$ .*

**Remark 3.** *If  $w > \log^{3+\epsilon} n$  and the input set of keys is sorted, the time taken to build the dictionary from Theorem 3 is  $O(n)$ .*

At the moment, our fast static dictionaries do not yield an improvement for dynamic deterministic dictionaries. It is one of major challenges in data structures research to either significantly improve performance of dynamic dictionaries, or to prove general lower bounds that would definitely establish a gap between deterministic and randomized dictionaries. How far deterministic dictionaries can go remains unknown, even in the static case.

## 2 Universes of Polynomial Size

### 2.1 Notation and comments

We use the symbol  $\oplus$  to denote bitwise exclusive or operation. The number of *collisions* of a function  $h$  on a subset  $A$  of its domain represents the value  $|\{\{x, y\} : h(x) = h(y) \wedge x, y \in A \wedge x \neq y\}|$ . For multisets  $A$  and  $B$ , the value  $|\{\{x, y\} \in A \times B : x = y\}|$ , which may be thought of as the number of collisions between the multisets, is denoted by  $\text{coll}(A, B)$ . For a multiset  $A$ ,  $A \oplus y$  stands for the multiset  $\{x \oplus y\}_{x \in A}$ . We use notation  $[x]$  to represent the set  $\{0, 1, \dots, x - 1\}$ . Also,  $\log x$  means  $\log_2 x$ .

Throughout the presentation, statements of performance bounds for both the word RAM model and the external memory models will appear at several places. The discussion was not separated for different models because we end up with essentially one construction algorithm for all the models, with the only difference being the sorting procedure that gets called (although on a RAM sorting can be avoided at some places, and we may have slightly simpler algorithms). By changing the procedure for sorting, we get methods efficient in the word RAM model, the I/O model, or the cache-oblivious model. In the external memory models we take the block size parameter  $B$  to represent the number of  $(\log |U|)$ -bit items that can fit in a memory block. For this problem,  $\log |U| = O(\log n)$ .

### 2.2 About universe size

Suppose that the universe has size  $2^{\lceil 2 \log N - 2 \log \log N \rceil}$ , for some  $N$  which is a power of two. We provide a dictionary structure that on a given set of  $n \leq N$  keys uses memory space of size  $O(N)$ , can be constructed in time  $O(n \log \log N + N)$ , and performs lookups in  $O(1)$  time. Hence, for universes of size  $O(n^2 / \log^2 n)$  such a structure immediately satisfies the desired performance. In case  $\Omega(n^2 / \log^2 n) \leq |U| \leq n^{O(1)}$  we may use a sequence of dictionary structures of the same type. The first dictionary is built over the projection of  $S$  on the first  $2 \log n - 2 \log \log n$  bits. If the size of the projected set is  $n_1$ , then each projected value can be assigned a unique identifier in the set  $[n_1]$ . During lookups these identifiers can be retrieved using the dictionary. The second dictionary is built over elements that are formed by concatenating the associated identifier and the projection of original key on the next  $2 \log n - \log n_1 - 2 \log \log n$  bits. This process is continued until all bits are exhausted. Since  $|U| = n^{O(1)}$  there is a constant number of dictionaries in the sequence.

A possible practical optimization is to handle small subsets directly. Namely, if an identifier value corresponds to a “small” number of elements of  $S$ , a specialized structure can store those elements, and they skip the rest of the general procedure. There are structures that can very efficiently handle sets of size  $\log^{O(1)} N$  (see Section 2.8).

### 2.3 Central part

Here we give a high-level description of the construction. Explanations of subprocedures and second-level structures follow in later subsections. Let  $\Psi = \log N$  and  $\Phi = \lceil \log N - 2 \log \log N \rceil$ . Suppose that  $\phi : U \rightarrow \{0, 1\}^\Phi$  and  $\psi : U \rightarrow \{0, 1\}^\Psi$  are functions such that the combined function  $(\phi, \psi)$  is 1-1 on  $U$ . An easy choice is to take  $\phi$  to be the projection on the  $\Phi$  highest order bits, and  $\psi$  to be the projection on the  $\Psi$  lowest order bits of binary representations of keys. We have that  $\Phi + \log \Phi + \log \Psi \leq \Psi + 1$  and  $\log n \leq \Psi$ .

The main hash function is of type

$$h(x) = \psi(x) \oplus a_{\phi(x)} ,$$

where  $(a_i)$  is an array of  $\Psi$ -bit elements, with  $i \in \{0, 1\}^\Phi$ . Our aim is to set the values of the array elements in a way that makes the function  $h$  have no more than  $3\Phi^2 n$  collisions on a given set  $S \subset U$ . It will become clear that this is always possible. After the function  $h$  is fixed, buckets of elements colliding under  $h$  need to be resolved. This is much easier than the original problem, since the average size of buckets is small. If the size of a bucket is less than  $\Phi^3 \Psi$  then a structure specialized for small sets will handle it. The total number of elements in the remaining (“large”) buckets is  $O(\frac{n}{\Phi^3 \Psi})$ . This can be seen by analyzing function  $\sum_i b_i$  under constraint  $\sum_i \binom{b_i}{2} \leq 3\Phi^2 n$  and with variable domains  $[\Phi^3 \Psi, \infty)$ . Let  $S'$  be the subset of  $S$  comprising the elements that fall in the “large” buckets. Constructing an efficient dictionary over  $S'$  will be an easier task, because we can afford to spend  $O(|S'| \Phi \Psi + 2^\Psi)$  construction time on it; we cover this in Section 2.9. No additional new techniques are required to design these second-level structures.

We will now give an overview of the algorithm for selecting values for the elements of the array  $a$ . The array  $a$  is initially set to all-zeros. Values of array elements will be decided in stages, with each stage being responsible for a separate set of bit positions. In our numbering of bit positions, position 0 refers to the most-significant bit position. Let  $i_* = \lfloor \log \Psi - \log \log \Phi - 1 \rfloor$ . There will be a total of  $2i_* + 2 = O(\log \Psi)$  stages. In the stages numbered  $1, 2, \dots, i_*$  the sizes of *active* sets of bit positions decrease roughly geometrically, while in the remaining  $i_* + 2$  stages they have the same (small) size. Let  $p_0 = 0$ ,  $p_i = \lfloor (1 - 2^{-i}) \Psi \rfloor - i \cdot \lfloor \log \Phi \rfloor$  for  $0 < i \leq i_*$ , and  $p_i = p_{i-1} + \lfloor \log \Phi \rfloor$  for  $i_* < i \leq 2i_* + 2$ . In the  $i$ th stage bits at positions between  $p_{i-1}$  and  $p_i - 1$  (inclusive) are decided on all elements of  $a$ .

The last  $i_* + 2$  stages could be replaced with different and shorter sequences. Yet, in this presentation of the algorithm we keep the chosen setting because it is relatively simple and incurs a relatively small increase in the overall constant factor. Operation in all the stages is done by the same procedure, parameterized by values  $p_{i-1}$  and  $p_i$ . We introduce symbols  $\eta_i$  denoting  $2^{p_i - p_{i-1} + \lfloor \log \Phi \rfloor}$ .

After the  $i$ th stage of the algorithm, the projection of  $h(x)$  on the high-order  $p_i$  bits is known. In other words, for any  $x \in U$  the value of  $h(x) \text{ div } 2^{\Psi - p_i}$  is fixed after the  $i$ th stage. To describe operation of the algorithm in stage  $i$ , we will define sets  $T(v, j, k)$ ,  $v \in \{0, 1\}^{p_{i-1}}$ ,  $0 \leq j \leq \Phi$ ,  $0 \leq k < 2^{\Phi - j}$  (whenever we talk about sets  $T(v, j, k)$  the stage number  $i$  is assumed to be fixed). Sets  $T(v, j, k)$  are defined recursively as follows:

- For any  $v \in \{0, 1\}^{p_{i-1}}$ ,  $T(v, \Phi, 0) = \{x \in S \mid h(x) \text{ div } 2^{\Psi - p_{i-1}} = v\}$ .
- For  $j < \Phi$ , if  $|T(v, j + 1, k \text{ div } 2)| < \eta_i$  then  $T(v, j, k) = \emptyset$ .
- For  $j < \Phi$ , if  $|T(v, j + 1, k \text{ div } 2)| \geq \eta_i$  then

$$T(v, j, k) = \{x \in T(v, \Phi, 0) \mid k2^j \leq \phi(x) < (k + 1)2^j\} .$$

Only non-empty sets  $T(v, j, k)$  are of interest to us. For any fixed  $v$ , subset relation on the family of non-empty sets  $T(v, j, k)$  can be described by a binary tree, with nodes labeled by pairs  $(j, k)$ . Sets  $T(v, j, k)$  that correspond to leaves of that tree are those that satisfy  $j = 0$  or  $T(v, j - 1, 2k) \cup T(v, j - 1, 2k + 1) = \emptyset$ . Let  $\{S_{vl}\}_{v,l}$  be the collection of all such “leaf” sets, over  $v \in \{0, 1\}^{p_{i-1}}$ . The collection  $\{S_{vl}\}$  is a partition of the set  $S$ .

No matter how the elements of the array  $a$  are modified in current and later stages, that is on bit positions from  $p_{i-1}$  to  $\Psi - 1$ , the number of collisions that  $h$  may create is bounded by  $\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}|$  plus a bound on the total number of collisions within the sets  $S_{vl}$ . If a set  $S_{vl}$  has size greater than  $\eta_i$  then it has to be one of the sets  $T(v, 0, k)$ . However, the set  $\{x \in S \mid \phi(x) = k\} \supset T(v, 0, k)$  is always mapped injectively by  $h$ . This follows from the definition of the function  $h$ , the fact that  $(\phi, \psi)$  is 1-1 on  $U$ , and the properties of xor operation. Therefore collisions may happen only within the sets  $S_{vl}$  such that  $|S_{vl}| < \eta_i$ . An upper bound on the total number of collisions that may happen within the sets  $S_{vl}$  is  $\frac{1}{2}n\eta_i$ , which can easily be seen by analyzing function  $\frac{1}{2} \sum_{j=1}^n b_j^2$  under constraint  $\sum_j b_j = n$  and over domain  $[0, \eta_i]^n$ . The goal of processing in stage  $i$  is to modify the values in the array  $a$  so that the number of collisions of  $h$  on  $S$  does not exceed

$$\eta_i \frac{n}{2} + \frac{1}{2^{p_i - p_{i-1}}} \sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}| , \quad (1)$$

when the stage ends. By solving appropriate recurrences, the following technical lemma can be shown. The proof is in Section 2.4.

**Lemma 4.** *If modifications to the array  $a$  by the selection algorithm make the number of collisions of  $h$  on  $S$  not exceed (1) at the end of stage  $i$ , for each  $i$ , then the final number of collisions will be less than  $3\Phi^2 n$ .*

The term  $\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}|$  from (1) can be re-expressed in an algorithmically more useful form. Each set  $T(v, j, k)$  is the union of some sets  $S_{vl}$ . Thus, we may write  $|T(v, j, k)| = \sum |S_{vl}|$ , where the sum is over all  $l$  such that  $S_{vl} \subset T(v, j, k)$ . The product  $|S_{vl_1}| \cdot |S_{vl_2}|$ , for some  $l_1, l_2$ , will be a term in the expanded expression for a product of type  $|T(v, j, 2k)| \cdot |T(v, j, 2k+1)|$ . Actually it will appear as a component of exactly one such product; in the mentioned binary tree, the node with label  $(j+1, k)$  has to be the lowest common ancestor of the nodes that correspond to the sets  $S_{vl_1}$  and  $S_{vl_2}$ . As a result, it holds that

$$\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}| = \sum_v \sum_{j=0}^{\Phi-1} \sum_{k=0}^{2^{\Phi-j-1}-1} |T(v, j, 2k)| \cdot |T(v, j, 2k+1)| .$$

After we specified the goal of processing in every stage, we proceed to giving a high-level description of the sequence of operations done at each stage. We introduce multiset variables  $X(v, j, k)$ , and we implicitly initialize all of them to  $\emptyset$ . In the outermost loop of the procedure,  $j$  takes values from 0 to  $\Phi - 1$ . We describe principal operations performed for a fixed  $j$ . First, for all leaf sets  $T(v, j, k)$ , i.e. those that equal one of the sets  $S_{vl}$ , we make the assignment

$$X(v, j, k) = \{(h(x) \operatorname{div} 2^{\Psi - p_i}) \bmod 2^{p_i - p_{i-1}} \mid x \in T(v, j, k)\} ,$$

where values  $h(x)$  are taken to be determined by the current state of the array  $a$ . We effectively calculated the projections of the current values  $h(x)$ ,  $x \in T(v, j, k)$ , on the bits at positions  $p_{i-1}$  through  $p_i - 1$ . The multisets can be stored as sets of element-multiplicity pairs. For each  $k$ ,  $0 \leq k \leq 2^{\Phi-j-1} - 1$ , the algorithm will find a value  $\delta \in \{0, 1\}^{p_i - p_{i-1}}$  such that

$$\sum_v \operatorname{coll}(X(v, j, 2k), X(v, j, 2k+1) \oplus \delta) \leq \frac{1}{2^{p_i - p_{i-1}}} \sum_v |T(v, j, 2k)| \cdot |T(v, j, 2k+1)|$$

and then make assignments  $X(v, j+1, k) = X(v, j, 2k) \cup (X(v, j, 2k+1) \oplus \delta)$ , where the union is in the multiset sense. The subprocedure that finds a suitable value  $\delta$  is described in Section 2.6. The elements of the array  $a$  are modified so that  $a_l = a_l \oplus 0^{p_i-1} \delta 0^{\Psi-p_i}$ , for  $(2k+1)2^j \leq l < (2k+2)2^j$ . At the end of the current iteration of the loop over  $j$ , the equality

$$X(v, j+1, k) = \{(h(x) \operatorname{div} 2^{\Psi-p_i}) \bmod 2^{p_i-p_i-1} \mid x \in T(v, j+1, k)\}$$

holds for every non-leaf set  $T(v, j+1, k)$ .

It is not hard to formally verify that a procedure conforming with this high-level description meets the specified goal of reducing the number of collisions of the function  $h$  on the set  $S$ . Performance analysis is completed in Section 2.7. We mention here that the following fact is used.

**Lemma 5.** *There can be at most  $4n \frac{\Phi+1}{\eta_i}$  non-empty multisets  $X(v, j, k)$ .*

## 2.4 The proof of Lemma 4

An upper bound on the number of possible collisions after completion of stage  $i$  is given by the following recurrence for  $i \leq i_*$ :

$$c_0 = \frac{n^2}{2}, \quad c_i = 2^{p_i-p_{i-1}} 2^{\lfloor \log \Phi \rfloor} \frac{n}{2} + \frac{1}{2^{p_i-p_{i-1}}} c_{i-1} .$$

The solution to the recurrence is

$$c_i = 2^{\lfloor \log \Phi \rfloor} \frac{n}{2} \sum_{j=1}^i \frac{2^{p_j-p_{j-1}}}{2^{p_i-p_j}} + \frac{1}{2^{p_i}} \frac{n^2}{2} .$$

According to the definition of the sequence  $(p_i)$ , the difference  $p_i - p_j$ , for  $j < i \leq i_*$ , satisfies the following inequality.

$$|(p_i - p_j) - (2^{-j} - 2^{-i})\Psi + (i-j)\lfloor \log \Phi \rfloor| \leq 1 . \quad (2)$$

Using (2) we may bound  $\frac{2^{p_j-p_{j-1}}}{2^{p_i-p_j}}$  by  $4 \cdot 2^{2^{-i}\Psi + (i-j-1)\lfloor \log \Phi \rfloor}$ . As a result,

$$c_i \leq 4 \cdot 2^{2^{-i}\Psi} \frac{n}{2} \sum_{j=0}^{i-1} 2^{j\lfloor \log \Phi \rfloor} + \frac{2^{i\lfloor \log \Phi \rfloor}}{2^{(1-2^{-i})\Psi}} n^2 .$$

A simple and relatively tight upper bound on  $c_{i_*}$  is  $2^{2^{-i_*}\Psi} 2^{i_*\lfloor \log \Phi \rfloor} (n + \frac{n^2}{2^\Psi})$ . We use this value as the starting point in the recurrence that corresponds to the remaining stages:

$$\bar{c}_0 = 2^{2^{-i_*}\Psi} 2^{i_*\lfloor \log \Phi \rfloor} \left( n + \frac{n^2}{2^\Psi} \right), \quad \bar{c}_i = 2^{2\lfloor \log \Phi \rfloor} \frac{n}{2} + \frac{1}{2^{\lfloor \log \Phi \rfloor}} \bar{c}_{i-1} .$$

The solution to this recurrence is

$$\bar{c}_i = 2^{2\lfloor \log \Phi \rfloor} \frac{n}{2} \sum_{j=0}^{i-1} \frac{1}{2^{j\lfloor \log \Phi \rfloor}} + \frac{1}{2^{i\lfloor \log \Phi \rfloor}} \bar{c}_0 .$$

After the whole procedure of selecting values for the elements of  $a$  is finished, the number of collisions of  $h$  on  $S$  is no more than  $\bar{c}_{i_*+2} < n\Phi^2(2 + 2^{-\Psi}n)$ . From  $\Psi \geq \log n$  it follows that  $\bar{c}_{i_*+2} < 3\Phi^2n$ , as required at the beginning.

## 2.5 Determining and arranging sets $S_{vl}$

We need a procedure that efficiently determines which of the sets  $T(v, j, k)$  are leaf sets, that is, those that equal one of the sets  $S_{vl}$ . This procedure needs to be executed at the beginning of every stage of the algorithm. The output should be a list of triples of form  $(v, j, k)$ . Naturally, the first step of the procedure is to sort the elements of  $S$  according to values of the function  $(h(x) \operatorname{div} 2^{\Psi-p_{i-1}}, \phi(x))$ . One way of determining leaf sets  $T(v, j, k)$  is to build *path-compressed* tries. A trie would be built for each value of  $h(x) \operatorname{div} 2^{\Psi-p_{i-1}}$  separately. By computing and storing weights of subtrees at each internal node, it is easy to determine the leaf sets recursively by the definition. This method would be efficient in the word RAM model, but we would have problems making it worst-case efficient in the external memory models.

We constructed a different and somewhat simpler algorithm for this problem, which is efficient both in terms of RAM time and I/O cost. The pseudo-code of the procedure for determining leaf sets  $T(v, j, k)$  is listed in Algorithm 1. Apart from the initial sorting, it runs in  $O(n)$  time making  $O(n/B)$  I/Os. In terms of external memory computation, effectively two scans of the array are made; it is possible to do only a single scan with a small complication. The algorithm is not completely trivial, but it is not a problem to prove its correctness. We leave the tedious details out. Performance in the external memory setting is obvious. The only non-obvious points in time analysis for the RAM model are the two while loops that increase/decrease values of  $j$  and  $k$ . We charge each iteration of these loops to the latest output leaf set that contained at least  $\frac{1}{2}\eta_i$  elements. The crucial observation is that a total of  $O(\Phi)$  iterations may be charged to a set, possibly split between different executions of the loops. Since  $\eta_i > \Phi$ , for any  $i$ , we are done.

So far we have the labels of the leaf sets  $T(v, j, k)$  in form of triples  $(v, j, k)$ , and the set  $S$  is stored so that the elements of each set  $T(v, j, k)$  appear at consecutive cells. As a preparation for subsequent computations, the sets are separated into  $\Phi$  groups according to the value of the index  $j$ . Consider the group  $\bar{j}$ , for some  $0 \leq \bar{j} < \Phi$ . The sets in group  $\bar{j}$  do not participate in computations during the first  $\bar{j}$  iterations of the outer loop in stage  $i$ , i.e. for  $0 \leq j < \bar{j}$ . When  $j$  reaches value  $\bar{j}$  it is time to include those sets in processing, which requires the corresponding multisets  $X(v, \bar{j}, k)$  to be computed (remember that we are dealing only with leaf sets at this point). Remark that the description of the function  $h$  has changed since the start of stage  $i$ , and therefore this computation cannot be performed at an earlier time, before the iteration number  $\bar{j}$ . In the external memory setting the computation of the multisets  $X(v, \bar{j}, k)$  involves sorting of all elements in  $\cup_{v,k} T(v, \bar{j}, k)$ , where the union only includes the leaf sets, according to values of the function  $\phi$ , in order to compute the values of the function  $h$  on those elements. The function values are then computed by making linear scans over the obtained sorted sequence and the array  $a$ . Another sorting operation is used to group elements of each set  $X(v, \bar{j}, k)$  together, with the sets laid out sorted according to the values of indices  $k$  and  $v$ , respectively (having  $\phi(x)$  as the first attribute in a sorting key is not the same as having  $k$ -value as the first attribute when the key is composite).

## 2.6 Finding suitable $\delta$ values and merging the multisets

We do not store the multisets  $X(v, j, k)$  in the simplest form. To facilitate bit-by-bit selection of  $\delta$  values, we store weighted full binary tree representations of the multisets  $X(v, j, k)$ . Namely, for each multiset  $X(v, j, k)$  we store an array of length  $s_i = 2^{p_i-p_{i-1}+1} - 1$ . Let  $(w_q)_{q=1}^{s_i}$  denote such an array. Then for  $2^{p_i-p_{i-1}} \leq q \leq s_i$ ,  $w_q$  has the value of the multiplicity of element  $q - 2^{p_i-p_{i-1}}$  in the multiset. The multiplicity of an element that does not belong to the multiset is zero. For

```

 $(x_q) \leftarrow$  sequence of elements of  $S$  sorted according to values of the function
 $(h(x) \operatorname{div} 2^{\Psi-p_{i-1}}, \phi(x))$ ;

 $j \leftarrow \Phi$ ;
 $k \leftarrow 0$ ;
 $v \leftarrow h(x_1) \operatorname{div} 2^{\Psi-p_{i-1}}$ ;
 $count \leftarrow 1$ ;
 $q_{start} \leftarrow 1$ ;
for  $q \leftarrow 2$  to  $n$  do
  if  $v \neq h(x_q) \operatorname{div} 2^{\Psi-p_{i-1}}$  then
    if  $count > 0$  then
      Append triple  $(v, j, k)$  to the output list;
       $j \leftarrow \Phi$ ;
       $k \leftarrow 0$ ;
       $v \leftarrow h(x_q) \operatorname{div} 2^{\Psi-p_{i-1}}$ ;
       $count \leftarrow 1$ ;
       $q_{start} \leftarrow q$ ;
    else if  $\phi(x_q) \geq (k+1)2^j$  then
      if  $count > 0$  then
        Append triple  $(v, j, k)$  to the output list;
      while  $(k+2)2^j < \phi(x_q)$  do
         $j \leftarrow j+1$ ;
         $k \leftarrow k \operatorname{div} 2$ ;
      end
       $k \leftarrow k+1$ ;
       $count \leftarrow 1$ ;
       $q_{start} \leftarrow q$ ;
    else
       $count \leftarrow count+1$ ;
      if  $count \geq \eta_i$  then
        while  $j > 0 \wedge (k+1)2^j > \phi(x_q)$  do
           $j \leftarrow j-1$ ;
           $k \leftarrow 2k$ ;
        end
        if  $(k+1)2^j \leq \phi(x_q)$  then
          if  $\phi(q_{start}) < (k+1)2^j$  then
            Append triple  $(v, j, k)$  to the output list;
            while  $\phi(q_{start}) < (k+1)2^j$  do
               $q_{start} \leftarrow q_{start}+1$ ;
               $count \leftarrow count-1$ ;
            end
           $k \leftarrow k+1$ ;
      end
  end
end

```

**Algorithm 1:** Finding leaf sets  $T(v, j, k)$

$1 \leq q < 2^{p_i - p_{i-1}}$ ,  $w_q$  has the value of  $w_{2q} + w_{2q+1}$ .

Applying the operator  $\oplus$  with argument  $\delta$  on  $X(v, j, k)$  has the following effect on its weighted binary tree representation: if the bit at position  $l$  of  $\delta$  has value 1 then for every node of level  $l$  its left and right subtree are interchanged (the root is at level 0). Each level in the weighted tree for the set  $X(v, j, k) \oplus \delta$  is a permutation of the same level in the tree for  $X(v, j, k)$ . For level  $l$ , the first  $l$  bits of  $\delta$  determine the permutation.

For leaf sets  $T(v, j, k)$  the weighted tree representations of the corresponding multisets  $X(v, j, k)$  are constructed after computation and grouping of the sets  $X(v, j, k)$ , mentioned in Section 2.5. It will be convenient to have the arrays of weights split into segments corresponding to levels of the trees, and to group together segments for each level. There are  $p_i - p_{i-1}$  groups, and within each group segments are arranged sorted according to the values of indices  $k$  and  $v$  of their sets  $X(v, j, k)$ . All this preprocessing of data structures is done for the multisets  $X(v, \bar{j}, k)$  that correspond to the leaf sets  $T(v, \bar{j}, k)$  in the  $(\bar{j} + 1)$ -st iteration of the outer loop in stage  $i$ . A result of the first  $\bar{j}$  iterations of the loop are the same kind of data structures for the multisets  $X(v, \bar{j}, k)$  that correspond to non-leaf sets  $T(v, \bar{j}, k)$ , also laid out in the same way. As the last preparation step in the  $(\bar{j} + 1)$ -st iteration, before doing the main operations, those two sets of data structures are merged into one. Since in both parts all pieces are already arranged sorted according to tree-level number, and then indices  $k$  and  $v$ , no sorting operation is involved in this step.

Recall of our principal task at this place — for fixed  $j$  and  $k$  we need to select  $\delta$  such that the following relation is satisfied.

$$\sum_v \text{coll}(X(v, j, 2k), X(v, j, 2k + 1) \oplus \delta) \leq \frac{1}{2^{p_i - p_{i-1}}} \sum_v |T(v, j, 2k)| \cdot |T(v, j, 2k + 1)| \quad (3)$$

Initially  $\delta$  is set to 0, and in increasing order of bit positions it is decided whether to set the bit to 1 or not. Since in each problem the values of  $j$  and  $k$  are fixed, we may use  $(w_{vq})$  to denote the weight arrays of the multisets  $X(v, j, 2k)$ , and  $(w_{vq}^\delta)$  to denote the weight arrays of the multisets  $X(v, j, 2k + 1) \oplus \delta$ . Define the function

$$\mu(l, \delta) = \sum_v \sum_{q=2^l}^{2^{l+1}-1} w_{vq} \cdot w_{vq}^\delta .$$

For any  $\delta$ , the value of  $\mu(p_i - p_{i-1}, \delta)$  is an upper bound on

$$\sum_v \text{coll}(X(v, j, 2k), X(v, j, 2k + 1) \oplus \delta) .$$

Therefore it is sufficient to set the bits of  $\delta$  so that  $\mu(l + 1, \delta) \leq \frac{1}{2}\mu(l, \delta)$ , for every  $l$  (only the first  $l$  bits of  $\delta$  influence  $\mu(l, \delta)$ ). Let  $\delta_l$  be the value obtained by taking the first  $l$  bits of  $\delta$  and setting the remaining bits to zeros. By expanding  $w_{vq}$  and  $w_{vq}^\delta$  into  $w_{v2q} + w_{v2q+1}$  and  $w_{v2q}^{\delta_l} + w_{v2q+1}^{\delta_l}$ , we

get that

$$\begin{aligned} \mu(l, \delta) &= \sum_v \sum_{q=2^{l+1}}^{2^{l+2}-1} (w_{vq} \cdot w_{vq}^{\delta_l} + w_{vq+1} \cdot w_{vq+1}^{\delta_l}) + \\ &+ \sum_v \sum_{q=2^{l+1}}^{2^{l+2}-1} (w_{vq} \cdot w_{vq+1}^{\delta_l} + w_{vq+1} \cdot w_{vq}^{\delta_l}) . \end{aligned}$$

We expressed  $\mu(l, \delta)$  as the sum of two terms such that: if the bit at position  $l$  of  $\delta$  is set to 0 then  $\mu(l+1, \delta)$  becomes equal to the first term, and otherwise if it is set to 1 then  $\mu(l+1, \delta)$  becomes equal to the second term. The choice is made based on which value is smaller.

When deciding on the value of bit at position  $l$ , the procedure needs to permute the sequence  $(w_{vq}^0)_{q=2^{l+1}}^{2^{l+2}-1}$  according to  $\delta_l$  to get  $(w_{vq}^{\delta_l})_{q=2^{l+1}}^{2^{l+2}-1}$ , for each  $v$ . In the word RAM model this is easy to do in  $O(2^{l+1})$  time, while in external memory sorting is required. Joining the weighted binary tree representations of multisets  $X(v, j, 2k)$  and  $X(v, j, 2k+1) \oplus \delta$  into the same type of representation for the multiset  $X(v, j+1, k)$  is straightforward. It can even be done on-the-fly during the selection of bits of  $\delta$ . If the data structures for  $X(v, j+1, k)$  are stored at the same places were the structures for  $X(v, j, 2k)$  resided, then the memory that stored the structures for  $X(v, j, 2k+1)$  becomes available. The space should be compacted to contain no unused holes. The compaction can be done either at the end of processing in the iteration number  $j$ , or on-the fly during the other computations.

Total time spent in stage  $i$  on all operations covered in this subsection can be expressed as  $O(2^{p_i-p_{i-1}}) = O(\frac{\eta_i}{\Phi})$  per every non-empty multiset  $X(v, j, k)$ . When counting the number of I/Os in the external memory version, note that sorting operations in this part of the algorithm are conducted as procedures of multisorting of groups of elements of equal sizes. All other operations involve only sequential scans. The total number of I/Os made in stage  $i$  for the operations covered in this subsection can be expressed as  $O(\text{Sort}(2^{p_i-p_{i-1}})) = O(\frac{1}{\Phi} \text{Sort}(\eta_i))$  per every non-empty set  $X(v, j, k)$ .

## 2.7 Completing the analysis of the main algorithm

We analyze the total cost of one stage of the algorithm for selecting values for the elements of the array  $a$ . Aggregate complexity of the procedures specified in Section 2.5 is proportional to the complexity of sorting  $n$  integers of size  $O(\log N)$  bits. In the external memory setting the limited size of numbers does not help us (at least it is not known to), and the required complexity is simply  $O(\text{Sort}(n))$  I/Os. In the version for the word RAM model we choose to use radix sort when  $n \geq N/\log^2 N$ . Then, using a space of size  $O(n)$  sorting takes time  $O(n)$ . When  $n < N/\log^2 N$  we use a comparison-based sorting procedure. In any case, a time bound for this part is  $O(n + N/\log N)$ .

In Section 2.6 it was said that the total time spent in stage  $i$  on all the operations from that part can be expressed as  $O(\frac{\eta_i}{\Phi})$  per every non-empty set  $X(v, j, k)$ . Lemma 5 implies that the operations described in Section 2.6 run in total time  $O(n)$  in any stage. In the cache-oblivious model the cost is  $O(\frac{n}{B} \log_{M/B} \eta_i) = O(\text{Sort}(n))$  I/Os.

**Proof of Lemma 5.** Consider any fixed  $v$  and the set  $T(v, \Phi, 0)$  with its associated binary tree that captures the subset relation on non-empty sets  $T(v, j, k)$ . In the degenerate case  $|T(v, \Phi, 0)| <$

$\eta_i$  the tree is just the root node, and  $X(v, j, k) = \emptyset$  for any  $j, k$  ( $X(v, \Phi, 0)$  is also empty since no assignment involving that variable is made in the course of the algorithm; the outermost loop goes until  $j = \Phi - 1$ ). Suppose that  $|T(v, \Phi, 0)| \geq \eta_i$ . Since the collection of leaf sets  $T(v, j, k)$  is a partition of  $T(v, \Phi, 0)$ , among the leaf sets there can be at most  $2|T(v, \Phi, 0)|/\eta_i$  sets having size at least  $\eta_i/2$ . Any non-leaf set has a descendant leaf set of size at least  $\eta_i/2$ , which is straightforwardly derived from the definition of sets  $T(v, j, k)$ . We “assign” each non-leaf node to one of its descendant leaf node sets of size at least  $\eta_i/2$  (multiple nodes can be assigned to a leaf node). Further, we assign each leaf node  $(j, k)$  such that  $|T(v, j, k)| < \eta_i/2$  to one leaf descendant  $(j_1, k_1)$  of its parent for which  $|T(v, j_1, k_1)| \geq \eta_i/2$ . Each leaf node  $(j, k)$  such that  $|T(v, j, k)| \geq \eta_i/2$  can have at most  $2(\Phi - j)$  nodes assigned to it — at most two per every level on the path to the root. Consequently, there can be at most  $(2\Phi + 1)\frac{2}{\eta_i}|T(v, \Phi, 0)|$  non-empty sets  $X(v, j, k)$ , for the chosen fixed  $v$ . Summing over all  $v$  finishes the proof.  $\square$

The only remaining thing is to take into account the operations of modifying the elements of  $a$ , that is, the operations of type

$$a_l = a_l \oplus 0^{p_i-1} \delta 0^{\Psi-p_i} .$$

Every element gets modified  $\Phi$  times in every stage of the algorithm; therefore  $O(2^\Phi \Phi)$  time is spent on this process in one stage. Because of the inequality  $\Phi + \log \Phi + \log \Psi \leq \Psi + 1$ , a time bound is  $O(2^\Psi/\Psi) = O(N/\log N)$ . The elements of  $(a_l)$  are modified in order of increasing index  $l$ . Hence, I/O cost is  $O(\frac{N}{B \log N})$ .

Even if the sequence  $(p_i)$  was changed, possibly resulting in a higher number of stages, there could be at most  $\Psi = \log N$  stages. Hence, the bounds which were expressed only in terms of  $N$  yield a combined bound of  $O(N)$  over the entire algorithm. With the chosen parameters there are  $O(\log \Psi) = O(\log \log N)$  stages, implying a complete time bound of  $O(n \log \log N + N)$ . In the cache-oblivious model the algorithm makes  $O(\text{Sort}(n) \log \log N + N/B)$  I/Os.

## 2.8 Subsets of size $\log^{O(1)} N$

In the word RAM model, a generalization of *fusion trees* of Fredman and Willard [FW93] yields a linear-space static dictionary that on a set of  $m$  keys has a lookup time of  $O(1 + \frac{\log m}{\log w})$  and it can be constructed in  $O(m)$  time on sorted input (this was explicitly stated by Hagerup [Hag98]). Plugging in  $m = \log^{O(1)} N$  and  $w = \Omega(\log N)$  shows that lookup time would be  $O(1)$  in our case. Sorted sequence can be obtained through radix-sorting elements of type (bucket-id,  $x$ ) from all the small buckets.

Alternatively, we may use a combination of our deterministic signatures method [Ruž09] and packed B-trees (e.g. see [And95]). This combination is a bit simpler than fusion trees, and it can also be used in external memory. A signature function that maps keys to  $O(\log \log N)$  bit values can be found in linear time on sorted input. The signatures of size  $O(\log \log N)$  bits are stored in a packed B-tree. Since the word size is  $\Omega(\log N)$  bits, the tree will have a constant depth.

## 2.9 The secondary structure for large buckets

On a RAM we could resort to using the structure for polynomial-size universes from [HMP01]. Because there are  $o(n/\log n)$  elements at this part, the added construction cost is  $o(n)$ . However, in external memory this structure is inefficient to construct.

We will use a variation of our algorithm to cut the number of collisions on these elements down to  $o(n/\log n)$ . Then within the buckets created by this second-level structure we use hash functions with quadratic range; quadratic space can be afforded in each bucket because the sum of collisions is small. This approach is the same as in the FKS scheme [FKS84]. To find injective hash functions deterministically we may use the algorithm from [Ram96] or the algorithm from [Ruž08b]. The former one would run in time  $O(m^2 \log n)$  on a bucket of size  $m$ , while the latter one would take time  $O(m^2 \log m)$ . In any case the combined time is  $o(n)$ . These algorithms can easily be made efficient in the cache-oblivious model.

It remains to describe changes to some parameters from Section 2, which will allow a higher drop in the number of collisions. The parameters that correspond to sequences  $(p_i)$  and  $(\eta_i)$  will be set differently. Now there will be  $\Psi$  stages. We simply set  $\hat{p}_i = i$  and  $\hat{\eta}_i = 1$ , for  $0 \leq i \leq \Psi$ . The algorithm is obtained by substituting  $\hat{p}_i$  and  $\hat{\eta}_i$  for  $p_i$  and  $\eta_i$ , respectively. Except from some parts of the analysis, everything else stays the same. The recurrence for the number of possible collisions after completion of stage  $i$  is now simply:  $\hat{c}_0 = O((\frac{n}{\Phi\Psi})^2)$ ,  $\hat{c}_i = \frac{1}{2}\hat{c}_{i-1}$ . The final number of collisions is  $O(n/(\Phi\Psi)^2) = O(n/\log^2 n)$ . Performance bounds for procedures described in Section 2.6 grew by a factor of  $O(\Phi\Psi)$ , but this was amortized by having the input set of smaller size.

Remark that only a small fraction of the elements goes through the additional complications from this part.

## 3 Larger Universes

### 3.1 Background on signature functions

The basic type of functions used in [Ruž09] is  $f(x, s, a) = x \operatorname{div} 2^s + a \cdot (x \bmod 2^s)$ , where  $a$  is a parameter chosen from  $\{1, 2, \dots, n^c - 1\}$ ,  $c \geq 2$ . The parameter  $s$  has a value dependent only on the domain of  $x$ , for example  $s = \lfloor \frac{1}{2} \log |U| \rfloor$ . The integer division and modulo functions were chosen as they are perhaps the simplest of all pairs of functions  $(\phi, \psi)$  such that  $(\phi, \psi)$  is 1-1 on  $U$ , and so that both functions map to a (significantly) smaller universe. In a more general form, we write  $f(x, a) = \phi(x) + a \cdot \psi(x)$ . Suppose that  $K$  is the number of keys that can be packed in a machine word. With  $c = 3.42$ , on any given set of  $n$  keys, a value for the parameter  $a$  that makes the function  $f$  injective on the set can be found in time  $O(n(\log n)^2 \frac{\log K}{K} + (\log n)^3)$ . The basic function can be combined in different ways to achieve a larger reduction of universe. The ultimate goal is to have a function that maps original keys to signatures of size  $O(\log n)$  bits. Many concrete variants of this approach can be imagined, yet we need only two.

Let  $x[0]x[1] \dots x[q-1]$  be a string representation of key  $x$  over some alphabet. One possibility is to apply  $f$  to all characters and concatenate the resulting values, viewed as binary strings. We may use the same multiplier parameter for all characters, and thus the length-reduced value for key  $x$  after one level of reduction has a form of  $f(x[0], a)f(x[1], a) \dots f(x[q-1], a)$ . The process is repeated with different multipliers and possibly different alphabets at subsequent levels of reduction. We refer to this way of combining function  $f$  as the *parallel reduction*. In the second version, which we call *suffix reduction*, only the last characters get reduced at a single reduction level. Although the structure of reduction sequences is different for those two variants, as well as the processes of parameter selection, the final functions for those two compositions can have similar *dot product* forms. A precondition for this is to suitably set intermediate alphabet sizes in the parallel reduction. For example, after two levels of parallel reduction we want the function to have a form

of  $x \mapsto f(f(x[0], a)f(x[1], a), a') \dots f(f(x[q-2], a)f(x[q-1], a), a')$ . Having the final functions in dot product form means that they can be evaluated rather efficiently on a word RAM. For more information about these methods see [Ruž09].

A final injective function is composed of functions generated by the method of parallel reduction and the method of suffix reduction. It can be evaluated in constant time on a word RAM. When  $\log^{3+\epsilon} n < w < 2^{\frac{n}{\log n}}$  the construction algorithm runs in linear time on sorted input. We may use fusion trees to cover the extreme case  $w \geq 2^{\frac{n}{\log n}}$  efficiently. In this chapter we show an improved complexity of dictionary construction in the case that  $w \leq \log^{O(1)} n$ .

### 3.2 Speed-up of the suffix reduction

The briefly outlined method of making deterministic signatures produces perfect hash functions with ranges of polynomial size. The functions have rather succinct descriptions, and they might have an application outside of dictionary structures. Here we will give up the injectiveness requirement, and replace it with considerably weaker properties. These weaker functions will be useful only when combined with additional data structures, foremost a dictionary structure for universes of polynomial size. The variant of the suffix reduction method that we introduce is particularly efficient in the external memory models. Yet it also produces some useful results in the word RAM model.

Let  $x[i]_\sigma$  denote the  $i$ th character of key  $x$  viewed as a string over the alphabet  $[2^\sigma]$ . It is assumed that  $\sigma = \Omega(\log n)$ . In the original version of suffix reduction we want to find multipliers  $a_0, a_1, \dots, a_{q-2}$  such that the function

$$a_0 \cdot x[0]_\sigma + (\dots + (a_{q-3} \cdot x[q-3]_\sigma + (a_{q-2} \cdot x[q-2]_\sigma + x[q-1]_\sigma)) \dots) \quad (4)$$

is injective on  $S$ . The multiplier selection algorithm is applied  $q-1$  times, as suggested by the expression in (4). In the new version, the multiplier selection procedure is again called  $q-1$  times, but each time with an input set of size  $O(n/(\log n)^2)$ . There will be no limit on the number of collisions that the final function may cause. Yet the function will have some properties that will allow the initial searching problem to be reduced either to a problem over a universe of size  $O(\sigma)$  bits, or to a problem over a set of size  $O(\log^2 n)$ . The high level idea is to look for clusters of elements that already piled up and will hash to equal values by the final function, and to prevent further collisions between already formed clusters. Sorting operations (over shorter keys) will dominate the running times. Suppose that values for the parameters  $a_l, a_{l+1}, \dots, a_{q-2}$  were selected. If two keys  $x$  and  $y$  share the prefix of length  $l$  and the function values on their suffixes of length  $q-l$  collide, i.e.  $a_l \cdot x[l]_\sigma + a_{l+1} \cdot x[l+1]_\sigma + \dots + x[q-1]_\sigma = a_l \cdot y[l]_\sigma + a_{l+1} \cdot y[l+1]_\sigma + \dots + y[q-1]_\sigma$ , then  $x$  and  $y$  will certainly be mapped to the same value by the final function. On the other hand, if the length  $l$  prefixes of  $x$  and  $y$  differ, it does not matter what are the values of the partial function on their suffixes of length  $q-l$ , since the separation of their hash values will be decided at a later time. The construction algorithm will keep track of sufficiently large clusters of elements that are certain to collide given the already selected multipliers. Different clusters will be ensured to map to different values. However keys not yet belonging to any cluster are able to join existing clusters or form new ones. The time of joining a cluster for a given key, specified by a prefix length, is possible to determine quickly during lookups. Some pieces of information related to this joining point will enable us to substantially reduce the search space. To be precise, the reduced search space will consist of keys of length  $O(\sigma)$  bits. If we set  $\sigma = \Theta(\log n)$  then the method can be composed with

the structure from Section 2. In some uses of the method, as we will see in Section 3.3, we need to set a higher value of  $\sigma$ ; there the “pipelined” dictionary for keys of length  $O(\sigma)$  bits is more complex.

The computationally dominant process in the construction algorithm will usually be sorting. The procedure performs  $O(q)$  sorting operations over sets of  $O(n)$  keys of length  $O(\sigma)$  bits. In the external memory models this amounts to  $O(\text{Sort}(n))$  I/Os (with the block size  $B$  expressed in terms of  $(\log |U|)$ -bit items, where  $U$  is the universe of keys that are input to the method). Combining this with the result from Section 2 produces the result stated in Theorem 2. In the word RAM model, the total sorting time is in general  $O(nq \log \log n)$ , based on [Han04]. When  $\sigma = \Theta(\log n)$  we may use radix sort and get a time bound of  $O(nq)$ , which explains Remark 2. When  $q$  is sufficiently large it makes sense to use a serial version of the parallel sorting algorithm from [AH92]; however, in our applications, suffix reduction is used with relatively small values of  $q$ .

We will mainly describe the process of constructing the functions, and through it provide understanding of the properties that the functions possess. The lookup procedure will become immediate once the construction is understood.

Suppose w.l.o.g that the sequence of the input keys is ordered as  $x_1 < x_2 < \dots < x_n$ . As a preparation step, the procedure will assign some identifiers to prefixes of the keys. The identifiers need to be relatively small. Consider the set  $\{x[0]_\sigma x[1]_\sigma \dots x[l-1]_\sigma \mid x \in S\}$  of prefixes of length  $l$  of the keys. To each value in this set we want to assign a unique identifier from the set  $[n]$ . No relation is imposed between identifiers of prefixes of different lengths. For each element  $x_k \in S$  we further want to have identifiers of its prefixes stored in a word-packed form, in the natural order. Assignment of prefix identifiers and storing of identifier sequences can be done simultaneously, doing an iteration through the sorted sequence of keys. For the first element we write the sequence  $(0, 0, \dots, 0)$ . Suppose that the sequence of prefix identifiers for  $x_k$  is  $(p_0, p_1, \dots, p_{q-1})$ . Let  $l$  be the length of the longest common prefix of  $x_k$  and  $x_{k+1}$ . Then for element  $x_{k+1}$  we write the sequence

$$(p_0, p_1, \dots, p_l, p_{l+1} + 1, \dots, p_{q-2} + 1, p_{q-1} + 1) .$$

The entire operation involving elements  $x_k$  and  $x_{k+1}$  can be executed in constant time using some standard techniques of computation using word-level parallelism. Let  $p(k, l)$  denote the identifier value of prefix of  $x_k$  of length  $l$ .

We introduce a collection variable  $C$  and initialize it to  $\emptyset$ . During the construction procedure, every element of  $S$  will be assigned at most one element from  $C$ . Let  $(c_i)_{i=1}^n$  be an array such that  $c_i$  holds a reference to the item from  $C$  assigned to  $x_i$ , if one exists; otherwise  $c_i$  has value  $-1$ .

For the first parameter in the sequence, which is  $a_{q-2}$ , we do not even need to run the selection algorithm; we simply put some value, e.g.  $a_{q-2} = 1$ . Then, the algorithm computes the set of triples

$$\{(p(k, q-2), a_{q-2} \cdot x_k[q-2]_\sigma + x_k[q-1]_\sigma, k)\}_{k=1}^n .$$

This set gets sorted in lexicographic order. The algorithm inspects subsets containing elements that match on the first two fields of the triples. Subsets of size less than  $\log^2 n$  are ignored. Consider a subset of size  $m \geq \log^2 n$ . Let the value of the second field be  $y$  and the set of values at the third field be  $\{k_1, k_2, \dots, k_m\}$ . The tuple  $(k_1, y)$  is added to the collection  $C$ , and references to it are set on  $c_{k_i}$ ,  $1 \leq i \leq m$ . No matter how the remaining parameters are selected, the elements  $x_{k_1}, \dots, x_{k_m}$  will be mapped to the same value by the final function.

Suppose that values for the parameters  $a_{l+1}, a_{l+2}, \dots, a_{q-2}$  were selected. We feed as input to the procedure that should select a value for  $a_l$  the set  $\{(y, x_k[l]_\sigma) \mid (k, y) \in C\}$  (the input was

represented as a set of pairs of values of functions  $\phi$  and  $\psi$ ). After the selection of  $a_l$ , the set of triples  $\{(p(k, l), a_l \cdot x_k[l]_\sigma + a_{l+1} \cdot x_k[l+1]_\sigma + \dots + x_k[q-1]_\sigma, k)\}_{k=1}^n$ , is computed and then sorted in lexicographic order. The algorithm inspects the subsets containing elements that match on the first two fields of the triples. Subsets of size less than  $\log^2 n$  are ignored. Consider a subset of size  $m \geq \log^2 n$ . Let the value of the second field be  $y$  and the set of values at the third field be  $\{k_1, k_2, \dots, k_m\}$ . If  $c_{k_i} = -1$  for all  $i \in \{1, \dots, m\}$ , then the tuple  $(k_1, y)$  is added to  $C$ , and references to it are set on  $c_{k_i}$ ,  $1 \leq i \leq m$ . Otherwise, suppose that  $c_{k_j} \neq -1$ . According to our construction, if  $c_{k_i} \neq -1$  for any  $i \in \{1, \dots, m\}$ , then it must be  $c_{k_i} = c_{k_j}$ . The assignments  $c_{k_i} = c_{k_j}$  are made for all  $i \neq j$ , and the second field of the associated tuple in  $C$  is changed to  $y$ .

Denote the final function by  $g$ . If  $q = n^{O(1)}$  the range of  $g$  has a size of  $O(\sigma)$  bits. The set  $\{g(x) \mid x \in S\}$  is plugged in as the input to a dictionary for universe of size  $O(\sigma)$  bits. For any  $i$ ,  $1 \leq i \leq n$ , it holds that either  $|\{x \in S \mid g(x) = g(x_i)\}| < \log^2 n$  or  $c_i \neq -1$ . The former case can be directly handled by a specialized structure. In the latter case, let  $(k, y)$  be the element of  $C$  referenced by  $c_i$ . The index  $k$  becomes the associated attribute of the hash value  $g(x_i) = g(x_k)$ .

For keys  $x \in U$  whose hash value  $g(x)$  falls in one of the buckets of size no less than  $\log^2 n$ , define function  $\bar{g}$  by

$$\bar{g}(x) = (l, p(k, l), x[l], a_{l+1} \cdot x[l+1]_\sigma + \dots + x[q-1]_\sigma),$$

where  $k$  is the index associated with value  $g(x)$ , and  $l$  is the length of the longest common prefix between  $x$  and  $x_k$ . It is not hard to prove that  $|\{x \in S \mid \bar{g}(x) = y\}| < \log^2 n$ , for any  $y$ . The elements  $\bar{g}(x)$  are stored in another dictionary for universes of size  $O(\sigma)$  bits, for keys  $x \in S$  whose hash value  $g(x)$  falls in one of the ‘‘large’’ buckets. With each stored value  $\bar{g}(x)$  we associate a reference to a dictionary specialized for sets of size  $\log^{O(1)} n$ .

Examining the performance complexity of the presented method is easy. Supposing that the pipelined dictionary for keys of length  $O(\sigma)$  bits has a lookup cost of  $O(1)$ , the entire cost of lookup is (a larger) constant. Excluding the construction complexity of the secondary dictionaries, sorting processes usually dominate the cost of the construction procedure (only when  $q$  is extremely large will the cost of the basic procedure that selects  $a_0, \dots, a_{q-2}$  outweigh the cost of sorting in the main procedure, and we never use this method for large  $q$ ). The main procedure performs  $O(q)$  sorting operations over sets of  $O(n)$  keys of length  $O(\sigma)$  bits.

It is interesting to observe that recursively applying the above reduction method a constant number of times leads to a dictionary with a construction time of  $O(n \log^\epsilon n)$  on keys of length  $\log^{O(1)} n$  bits, for any fixed  $\epsilon > 0$ ; the lookup time is constant for a fixed  $\epsilon$ , but grows quickly as  $\epsilon$  decreases towards 0. A theoretically superior approach is outlined in Section 3.3.

### 3.3 Speed-up of the parallel reduction

This section covers the remaining case that  $\omega(\log n \log \log n) < w < \log^{3+\epsilon} n$  to complete Theorem 3. The approach is conceptually very similar to the approach that led to the speed-up of the method of suffix reduction, but some details are different and the computation is more involved. Because of the similarities we give only an outline; it should be enough for understanding, provided that Section 3.2 is understood (as well as some of the prior work on our signature functions, on which everything is based).

Consider partially reduced keys, after some number of levels of the parallel reduction. We call a prefix value heavy if it is shared by at least  $(\log n)^2$  partially reduced keys. We ensure that

the current level of reduction avoids any collisions between heavy prefixes. For this purpose, it is convenient to maintain the trie of the partially reduced set (see [Ruž07]). It is again possible to substantially reduce the search space for a given key by using information related to the key's point of joining a cluster of piled up elements.

In order to determine the level at which a key joined a cluster in constant time, we need to evaluate partially reduced values of the key for all levels in constant time. This is possible if  $\Omega(K^2)$  copies of a key can fit in a single machine word, where  $K$  is the number of reduction levels. We explain this now. For simplicity, assume that  $2K^2$  copies of a key can fit into one word; the assumption will easily generalize to  $\Omega(K^2)$ . The value of each partial evaluation is a concatenation of dot products (in a complete evaluation there is just one dot product). The constant operand is the same for all products (at one level), while the second operand is a segment of a key; different segment for different products. Because the constant operand is the same, all dot products can still be computed in constant time, using two multiplications and a few bitwise operations. The resulting value is not compacted — there are gaps with zero bits — but this is not a problem, they may stay this way. However, partial evaluations that go until different levels use different kinds of dot products, and the key needs to be prepared (split using bitwise AND) differently due to different alphabet sizes. Therefore, we write  $K$  equidistant copies of the key in one word. The other word contains constant operands of dot products for different levels, spaced at distance equal to two key sizes. There needs to be a lot of zero space between copies of the key in the first word to hold all  $K$  calculation results. Actually, for each copy there is one meaningful result which is kept; the others are by-products of the computation, and they are thrown away.

For  $w = \log^{O(1)} n$  we have that  $K = O(\log \log n)$ . To provide a situation where  $\Omega(K^2)$  copies of a key can fit into one word, we use two levels of the searching problem reduction from Section 3.2, using the setting  $q = O(\log \log n)$ . Hence the incurred construction cost from these two reduction steps is  $O(n(\log \log n)^2)$ .

The construction of this version of the parallel reduction function has a time cost proportional to  $K$  times the sorting time. For  $w = \log^{O(1)} n$  we again get a bound of  $O(n(\log \log n)^2)$ . Since through the method of parallel reduction we map the keys to a range of size  $O(\log n \log \frac{w}{\log n})$  bits, at the bottom end we again employ the suffix reduction, but this time paired with the dictionary for polynomial-size universes.

## 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 (SODA)*, pages 463–472, 1992.
- [AN96] Noga Alon and Moni Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.
- [And95] Arne Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th Symposium on Foundations of Computer Science (FOCS)*, pages 655–663, 1995.
- [AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

- [BF03] Gerth Stlting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, 2003.
- [CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. of Comput. Syst. Sci.*, 18(2):143–154, 1979.
- [DKM<sup>+</sup>94] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.
- [FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. ACM*, 31(3):538–544, 1984.
- [FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999.
- [FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.
- [Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *15th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1373 of *LNCS*, pages 366–398. Springer, 1998.
- [Han04] Yijie Han. Deterministic sorting in  $O(n \log \log n)$  time and linear space. *J. Algor.*, 50(1):96–105, 2004.
- [HMP01] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algor.*, 41(1):69–85, 2001.
- [OvL81] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Proc. Lett.*, 12(4):168–173, 1981.
- [Pag00] Rasmus Pagh. A trade-off for worst-case efficient dictionaries. *Nordic J. Comput.*, 7(3):151–163, 2000.
- [Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proc. 4th European Symposium on Algorithms (ESA)*, volume 1136 of *LNCS*, pages 121–137. Springer-Verlag, 1996.
- [Ruž07] Milan Ružić. Making deterministic signatures quickly. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 900–909. ACM and SIAM, 2007.
- [Ruž08a] Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 84–95. Springer, 2008.
- [Ruž08b] Milan Ružić. Uniform deterministic dictionaries. *ACM Transactions on Algorithms*, 4(1):Article 1, 2008.

- [Ruž09] Milan Ružić. Making deterministic signatures quickly. *ACM Transactions on Algorithms*, 5(3):Article 26, 2009.
- [TY79] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.