Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation*

Yuriy Arbitman, Moni Naor†, and Gil Segev‡
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel
Email: yuriy.arbitman@gmail.com, {moni.naor, gil.segev}@weizmann.ac.il

***Abstract*—The performance of a dynamic dictionary is measured mainly by its update time, lookup time, and space consumption. In terms of update time and lookup time there are known constructions that guarantee constant-time operations in the worst case with high probability, and in terms of space consumption there are known constructions that use essentially optimal space. However, although the first analysis of a dynamic dictionary dates back more than 45 years ago (when Knuth analyzed linear probing in 1963), the trade-off between these aspects of performance is still not completely understood. In this paper we settle two fundamental open problems:**

- **We construct the first dynamic dictionary that enjoys the best of both worlds: it stores n elements using $(1+\epsilon)n$ memory words, and guarantees constant-time operations in the worst case with high probability. Specifically, for any $\epsilon = \Omega((\log\log n/\log n)^{1/2})$ and for any sequence of polynomially many operations, with high probability over the randomness of the initialization phase, all operations are performed in constant time which is independent of ϵ.**
  The construction is a two-level variant of cuckoo hashing, augmented with a "backyard" that handles a large fraction of the elements, together with a de-amortized perfect hashing scheme for eliminating the dependency on ϵ.
- **We present a variant of the above construction that uses only $(1+o(1))\mathcal{B}$ bits, where $\mathcal{B}$ is the information-theoretic lower bound for representing a set of size n taken from a universe of size u, and guarantees constant-time operations in the worst case with high probability, as before. This problem was open even in the *amortized* setting. One of the main ingredients of our construction is a permutation-based variant of cuckoo hashing, which significantly improves the space consumption of cuckoo hashing when dealing with a rather small universe.**

I. Introduction

A dynamic dictionary is a data structure used for maintaining a set of elements under insertions, deletions, and lookup queries. The first analysis of a dynamic dictionary dates back more than 45 years ago, when Knuth analyzed linear probing in 1963 [31] (see also [32]). Over the years dynamic dictionaries have played a fundamental role in computer science, and a significant amount of research has been devoted for their construction and analysis.

The performance of a dynamic dictionary is measured mainly by its update time, lookup time, and space consumption. Although each of these performance aspects alone can be made essentially optimal rather easily, it seems to be a highly challenging task to construct dynamic dictionaries that enjoy good performance in all three aspects. Specifically, in terms of update time and lookup time there are known constructions that guarantee constant-time operations in the worst case with high probability[1] (e.g., [1], [9], [11], [15]), and in terms of space consumption there are known constructions that provide almost full memory utilization (e.g., [18], [21], [45]) – even with constant-time lookups, but without constant-time updates.

In this paper we address the task of constructing a dynamic dictionary that enjoys optimal guarantees in all of the above aspects. This problem is motivated not only by the natural theoretical insight that its solution may shed on the feasibility and efficiency of dynamic dictionaries, but also by practical considerations. First, the space consumption of dictionary is clearly a crucial measure for its applicability in the real world. Second, whereas amortized performance guarantees are suitable for a very wide range of applications, for other applications it is highly desirable that all operations are performed in constant time in the worst case. For example, in the setting of hardware routers and IP lookups, routers must keep up with line speeds and memory accesses are at a premium (see [4], [29]). An additional motivation for the construction of dictionaries with worst case guarantees is combatting "timing attacks", first suggested by Lipton and Naughton [35]. They showed that timing information may reveal sensitive information on the randomness used by the data structure, and this can enable an adversary to identify elements whose insertion results in poor running time. The concern regarding timing information is even more acute in a cryptographic environment with an active adversary who might use timing information to compromise the security of the system (see, for example, [33], [50]).

* We refer the reader to a longer version available as [2].

† Incumbent of the Judith Kleeman Professorial Chair. Research supported in part by a grant from the Israel Science Foundation. Part of this work was done while visiting the Center for Computational Intractability at Princeton University.

‡ Research supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

[1] More specifically, for any sequence of operations, with high probability over the randomness of the initialization phase of the data structure, each operation is performed in constant time.

A. Our Contributions

In this paper we settle two fundamental open problems in the design and analysis of dynamic dictionaries. We consider the unit cost RAM model in which the elements are taken from a universe of size u, and each element can be stored in a single word of length $w = \lceil \log u \rceil$ bits. Any operation in the standard instruction set can be executed in constant time on w-bit operands. This includes addition, subtraction, bitwise Boolean operations, left and right bit shifts by an arbitrary number of positions, and multiplication[2]. Our contributions are as follows:

Achieving the best of both worlds. We construct a two-level variant of cuckoo hashing [44] that uses $(1+\epsilon)n$ memory words, where n is the maximal number of elements stored at any point in time, and guarantees constant-time operations in the worst case with high probability. Specifically, for any $0 < \epsilon < 1$ and for any sequence of polynomially many operations, with overwhelming probability over the randomness of the initialization phase, all insertions are performed in time $O(\log(1/\epsilon)/\epsilon^2)$ in the worst case. Deletions and lookups are always performed in time $O(\log(1/\epsilon)/\epsilon^2)$ in the worst case.

We then show that this construction can be augmented with a de-amortized perfect hashing scheme, resulting in a dynamic dictionary in which all operations are performed in constant time which is independent of ϵ, for any $\epsilon = \Omega((\log\log n/\log n)^{1/2})$. The augmentation is based on a rather general de-amortization technique that can rely on any perfect hashing scheme with two natural properties.

Succinct representation. The above construction stores n elements using $(1+o(1))n$ memory words, which are $(1+o(1))n\log u$ bits. This may be rather far from the information-theoretic bound of $\mathcal{B}(u,n) = \lceil \log \binom{u}{n} \rceil$ bits for representing a set of size n taken from a universe of size u. We present a variant of our construction that uses only $(1+o(1))\mathcal{B}$ bits[3], and guarantees constant-time operations in the worst case with high probability as before. Our approach is based on hashing elements using permutations instead of functions. We first present a scheme assuming the availability of truly random permutations, and then show that this assumption can be eliminated by using k-wise δ-dependent permutations.

Permutation-based cuckoo hashing. One of the main ingredients of our construction is a permutation-based variant of cuckoo hashing. This variant improves the space consumption of cuckoo hashing by storing n elements using $(2+\epsilon)n\log(u/n)$ bits instead of $(2+\epsilon)n\log u$ bits. When dealing with a rather small universe, this improvement to the space consumption of cuckoo hashing might be much more significant than that guaranteed by other variants of cuckoo hashing that store n elements using $(1+\epsilon)n\log u$ bits [18], [21], [45]. Analyzing our permutation-based variant is more challenging than analyzing the standard cuckoo hashing, as permutations induce inherent dependencies among the outputs of different inputs (these dependencies are especially significant when dealing with a rather small universe). Our analysis relies on subtle coupling argument between a random function and a random permutation, that is enabled by a specific monotonicity property of the bipartite graphs underlying the structure of cuckoo hashing.

Application of small universes: A nearly-optimal Bloom filter alternative. The difference between using $(1+o(1))\log\binom{u}{n}$ bits and using $(1+o(1))n\log u$ bits is significant when dealing with a small universe. An example for an application where the universe size is small and in which our construction yields a significant improvement arises when applying dictionaries to solve the *approximate set membership problem*: representing a set of size n in order to support lookup queries, allowing a false positive rate of at most $0 < \delta < 1$, and no false negatives. In particular, we are interested in the *dynamic* setting where the elements of the set are specified one by one via a sequence of insertions. This setting corresponds to applications such as graph exploration where the inserted elements correspond to nodes that have already been visited (e.g. [8]), global deduplication-based compression systems where the inserted elements correspond to data segments that have already been compressed (e.g. [53]), and more. In these applications δ has to be roughly $1/n$ so as not to make any error in the whole process.

The information-theoretic lower bound for the space required by any solution to this problem is $n\log(1/\delta)$ bits, and this holds even in the static setting where the set is given in advance [7]. The problem was first solved using a Bloom filter [3], whose space consumption is $n\log(1/\delta)\log e$ bits (i.e., this is a compact representation).

Using our succinctly-represented dictionary we present the first solution to this problem whose space consumption is only $(1+o(1))n\log(1/\delta)+O(n+\log u)$ bits, and guarantees constant-time lookups and insertions in the worst case with high probability (previously such guarantees were only known in the amortized sense). In particular, the lookup time and insertion time are independent of δ. For any sub-constant δ (the case in the above applications), and under the reasonable assumption that $u \leq 2^{O(n)}$, the space consumption of our solution is $(1+o(1))n\log(1/\delta)$, which is optimal up to an additive lower order term (i.e., this is a succinct representation)[4].

B. Related Work

A significant amount of work was devoted to constructing dynamic dictionaries over the years, and here we focus only on the results that are most relevant to our setting.

Dynamic dictionaries with constant-time operations in the worst case. Dietzfelbinger and Meyer auf der Heide [15] constructed the first dynamic dictionary with constant-time operations in the worst case with high probability, using $O(n)$

[2]The unit cost RAM model has been the subject of much research, and is considered the standard model for analyzing the efficiency of data structures (see, for example, [16], [26], [27], [39], [41], [47] and the references therein).

[3]Demaine [10] classifies data structures into "implicit" (redundancy $O(1)$), "succinct" (redundancy $o(\mathcal{B})$) and "compact" (redundancy $O(\mathcal{B})$).

[4]For constant δ there is a recent lower bound of Lovett and Porat [36] showing we cannot get to $(1+o(1))n\log(1/\delta)$ bits in the dynamic setting.

memory words to store n elements (the construction is based on the dynamic dictionary of Dietzfelbinger et al. [14]). While this construction is a significant theoretical contribution, it may be unsuitable for highly demanding applications. Most notably, it suffers from large multiplicative constant factors in its memory utilization and running time, and from an inherently hierarchal structure. Recently, Arbitman et al. [1] presented a de-amortization of cuckoo hashing that guarantees constant-time operations in the worst case with high probability, and achieves memory utilization of about 50%. Their experimental results indicate that the scheme is efficient, and provides a practical alternative to the construction of Dietzfelbinger and Meyer auf der Heide.

Dynamic dictionaries with full memory utilization. Linear probing is the most classical hashing scheme that offers full memory utilization. When storing n elements using $(1+\epsilon)n$ memory words, its expected insertion time is polynomial in $1/\epsilon$. However, for memory utilization close to 100% it is rather inefficient, and the average search time becomes linear in the number of elements stored (for more details see Theorem K and the subsequent discussion in [32, Chapter 6.4]).

Cuckoo hashing [44] achieves memory utilization of slightly less than 50%, and its generalizations [18], [21], [45] were shown to achieve full memory utilization. These generalizations follow two lines: using multiple hash functions, and storing more than one element in each bin. To store n elements using $(1+\epsilon)n$ memory words, the expected insertion time when using multiple hash functions was shown to be $(1/\epsilon)^{O(\log\log(1/\epsilon))}$, and when using bins with more than one element it was shown to be $\log(1/\epsilon)^{O(\log\log(1/\epsilon))}$. For further and improved analysis see also [6], [12], [13], [20], [22], [24], [25], [34].

Fotakis et al. [21] suggested a general approach for improving the memory utilization of a given scheme by employing a multi-level construction: their dictionary comprises of several levels of decreasing sizes, and elements that cannot be accommodated in any of these levels are placed in an auxiliary dictionary. Their scheme, however, does not efficiently support deletions, and the number of levels (and thus also the insertion time and lookup time) depends on the overall loss in memory utilization.

Dictionaries approaching the information-theoretic space bound. A number of dictionaries with space consumption that approaches the information-theoretic space bound are known. Raman and Rao [47] constructed a dynamic dictionary that uses $(1+o(1))\mathcal{B}$ bits, but provides only amortized guarantees and does not support deletions efficiently. The above mentioned construction of Dietzfelbinger and Meyer auf der Heide [15] was extended by Demaine et al. [11] to a dynamic dictionary that uses $O(\mathcal{B})$ bits[5], where each operation is performed in constant time in the worst case with high probability. Of particular interest to our setting is their construction of quotient hash functions, that are used to hash elements similarly to the way our construction uses permutations (permutations can be viewed as a particular case of quotient hash functions). Our approach using k-wise almost independent permutations can be used to significantly simplify their construction, and in addition it allows a more uniform treatment without separately considering different ranges of the parameters.

In the static dictionary case (with no insertions or deletions) much work was done on succinct data structures. The first to achieve a succinct representation of static dictionary supporting $O(1)$ retrievals were Brodnik and Munro [5]. More efficient schemes were given by [43] and [16]. Most recently, Pătraşcu [46] showed a succinct dictionary where the redundancy can be $O(n/\mathrm{polylog}(n))$.

II. Preliminaries and Tools

***k*-wise independent functions.** A collection $\mathcal{F}$ of functions $f : U \to V$ is k-wise independent if for any distinct $x_1,\ldots,x_k \in U$ and for any $y_1,\ldots,y_k \in V$ it holds that

$$\Pr\left[f(x_1)=y_1 \wedge \cdots \wedge f(x_k)=y_k\right] = 1/|V|^k \ .$$

More generally, a collection $\mathcal{F}$ is k-wise δ-dependent if for any distinct $x_1,\ldots,x_k \in U$ the distribution $(f(x_1),\ldots,f(x_k))$ where f is sampled from $\mathcal{F}$ is δ-close in statistical distance to the distribution $(f^*(x_1),\ldots,f^*(x_k))$ where f^* is a truly random function. A simple example for k-wise independent functions is the collection of all polynomials of degree $k-1$ over a finite field.

In this paper we are interested in functions that have a short representation and can be evaluated in constant time in the unit cost RAM model. Although there are no such constructions of k-wise independent functions, Siegel [48] constructed a pretty good approximation that is sufficient for our applications (see also the recent improvement of Dietzfelbinger and Rink [17] to Siegel's construction). For any two sets U and V of size polynomial in n, and for any constant $c>0$, Siegel presented a randomized algorithm outputting a collection $\mathcal{F}$ of functions $f : U \to V$ with the following guarantees:

1) With probability at least $1-n^{-c}$, the collection $\mathcal{F}$ is n^α-wise independent for some constant $0<\alpha<1$ that depends on $|U|$ and n.
2) Any function $f \in \mathcal{F}$ is represented using n^β bits, for some constant $\alpha<\beta<1$, and evaluated in constant time in the unit cost RAM model.

Several comments are in place regarding the applicability of Siegel's construction in our setting. First, whenever we use n^α-wise independent functions in this paper, we instantiate them with Siegel's construction, and this contributes at most an additive n^{-c} factor to the failure probability of our schemes[6]. Second, the condition that U and V are of polynomial size does not hurt the generality of our results: in our applications $|V| \le |U|$, and U can always be assumed to be of sufficiently large polynomial size by using a pairwise (almost) independent function mapping U to a set of polynomial size without

[5]Using the terminology of Demaine [10], this data structure is "compact".

[6]Note that property 1 above is stronger in general than k-wise δ-dependence.

any collisions with high probability. Finally, each function is represented using n^β bits, for some constant $\beta < 1$, and this enables us in particular to store any constant number of such functions: the additional space consumption is only $O(n^\beta) = o(n \log(u/n))$ bits which is negligible compared to the space consumption of our schemes.

A significantly simpler and more efficient construction, but with a weaker guarantee on the randomness, was provided by Dietzfelbinger and Woelfel [19] following Pagh and Pagh [41] (see also [17]). For any two sets U and V of size polynomial in n, and for any integer $k \leq n$ and constant $c > 0$, they presented a randomized algorithm outputting a collection $\mathcal{F}$ of functions $f : U \to V$ with the following guarantees:

1) For any *specific* set $S \subset U$ of size k, there is an n^{-c} probability of failure (i.e., choosing a "bad" function for this set), but if failure does not occur, then a randomly chosen $f \in \mathcal{F}$ is fully random on S.
2) Any function $f \in \mathcal{F}$ is represented using $O(k \log n)$ bits, and evaluated in constant time in the unit cost RAM model.

Note that such a guarantee is indeed slightly weaker than that provided by Siegel's construction: in general, we cannot identify a bad event whose probability is polynomially small in n, so that if it does not occur then the resulting distribution is k-wise independent. Therefore it is harder to plug in such a distribution instead of an exact k-wise independent distribution (e.g., it is not clear that the k-th moments remain the same). Specifically, this type of guarantee implies that for a set of size n, if one considers all its subsets of size k, then a randomly chosen function from the collection behaves close to a truly random function on each set, but this does not necessary hold *simultaneously* for all subsets of size k, as we would like in many applications. Nevertheless, inspired by the approach of [17], in the longer version of this paper [2] we show that our constructions can in fact rely on such a weaker guarantee, resulting in significantly simpler and more efficient instantiations.

***k*-wise almost independent permutations.** A collection Π of permutations $\pi : U \to U$ is k-wise δ-dependent if for any distinct $x_1, \ldots, x_k \in U$ the distribution $(\pi(x_1), \ldots, \pi(x_k))$ where π is sampled from Π is δ-close in statistical distance to the distribution $(\pi^*(x_1), \ldots, \pi^*(x_k))$ where π^* is a truly random permutation. For $k > 3$ no explicit construction is known for k-wise *exactly independent* permutations (i.e., $\delta = 0$), and therefore it seems rather necessary to currently settle for *almost independence* (see [28] for a more elaborated discussion).

In the longer version of this paper [2] we observe a construction of k-wise δ-dependent permutations with a short description and constant evaluation time. The construction is obtained by combining known results from two independent lines of research: constructions of pseudorandom permutations (see, for example, [37], [40]), and constructions of k-wise independent functions with short descriptions and constant evaluation time as discussed above.

III. The Backyard Construction

Our construction is based on two-level hashing, where the first level consists of a collection of bins of constant size each, and the second level consists of cuckoo hashing. One of the main observations underlying our construction is that the specific structure of cuckoo hashing enables a very efficient interplay between the two levels.

Full memory utilization via two-level hashing. Given an upper bound n on the number of elements stored at any point in time, and a memory utilization parameter $0 < \epsilon < 1$, set $d = \lceil c \log(1/\epsilon)/\epsilon^2 \rceil$ for some constant $c > 1$, $m = \lceil (1+\epsilon/2)n/d \rceil$, and $k = \lceil n^\alpha \rceil$ for some constant $0 < \alpha < 1$. The first level of our dictionary is a table T_0 containing m entries (referred to as bins), each of which contains d memory words. The table is equipped with a hash function $h_0 : \mathcal{U} \to [m]$ that is sampled from a collection of k-wise independent hash functions (see Section II for constructions of such functions with succinct representation and constant evaluation time). Any element $x \in \mathcal{U}$ is stored either in the bin $T_0[h_0(x)]$ or in the second level. The lookup procedure is straightforward: when given an element x, perform a lookup in the bin $T_0[h_0(x)]$ and in the second level. The deletion procedure simply deletes x from its current location. As for inserting an element x, if the bin $T_0[h_0(x)]$ contains less than d elements then we store x there, and otherwise we store x in the second level. We show that the number of elements that cannot be stored in the first level after exactly n insertions is at most $\epsilon n/16$ with high probability. Thus, the second level should be constructed to store only $\epsilon n/16$ elements.

Supporting deletions efficiently: cuckoo hashing. When dealing with long sequences of operations (as opposed to only n insertions as considered in the previous paragraph), we must be able to move elements from the second level back to the first level. Otherwise, when elements are deleted from the first level, and new elements are inserted into the second level, it is no longer true that the second level contains at most $\epsilon n/16$ elements at any point in time. One possible solution to this problem is to equip each first-level bin with a doubly-linked list, pointing to all the "overflowing" elements of the bin (these elements are stored in the second level). Upon every deletion from a bin in the first level we move one of these overflowing elements from the second level to this bin. We prefer, however, to avoid such a solution due to its extensive usage of pointers and the rather inefficient maintenance of the linked lists.

We provide an efficient solution to this problem by using cuckoo hashing as the second level dictionary. Cuckoo hashing uses two tables T_1 and T_2, each consisting of $r = (1+\delta)\ell$ entries for some small constant $\delta > 0$ for storing at most $\ell = \epsilon n/16$ elements, and two hash functions $h_1, h_2 : \mathcal{U} \to \{1, \ldots, r\}$. An element x is stored either in entry $h_1(x)$ of table T_1 or in entry $h_2(x)$ of table T_2, but never in both. The lookup and deletion procedure are naturally defined, and as for insertions, Pagh and Rodler [44] proved that the "cuckoo approach", kicking other elements away until every element has its own "nest", leads to an efficient insertion procedure.

More specifically, in order to insert an element x we store it in entry $T_1[h_1(x)]$. If this entry is not occupied, then we are done, and otherwise we make its previous occupant "nestless". This element is then inserted to T_2 using h_2 in the same manner, and so forth iteratively. We refer the reader to [44] for a more comprehensive description of cuckoo hashing.

A very useful property of cuckoo hashing in our setting is that in its insertion procedure, whenever stored elements are encountered we add a test to check whether they actually "belong" to the main table T_0 (i.e., whether their corresponding bin has an available entry). The key property is that if we ever encounter such an element, the insertion procedure is over (since an available position is found for storing the current nestless element). Therefore, as far as the cuckoo hashing is concerned, it stores at most $\epsilon n/16$ elements at any point in time. This guarantees that any insert operation leads to at most one insert operation in the cuckoo hashing, and one insert operation in the first-level bins.

Constant worst-case operations: de-amortized cuckoo hashing. Instead of using the classical cuckoo hashing we use the recent construction of Arbitman et al. [1] who showed how to de-amortize the insertion time of cuckoo hashing using a queue. The insertion procedure in the second level is now parameterized by a constant L, and is defined as follows. Given a new element x (which cannot be stored in the first level), we place the pair $(x, 1)$ at the *back* of the queue (the additional value indicates to which of the two cuckoo tables the element should be inserted next). Then, we carry out the following procedure as long as no more than L moves are performed in the cuckoo tables: we take the pair (y, b) from the *head* of the queue, and check whether y can be inserted into the first level. If its bin in the first level is not full then we store y there, and otherwise we place y in entry $T_b[h_b(y)]$. If this entry was unoccupied (or if y was successfully moved to the first level of the dictionary), then we are done with the current element y, this is counted as one move and the next element is fetched from the *head* of the queue. However, if the entry $T_b[h_b(y)]$ was occupied, we check whether its previous occupant z can be stored in the first level and otherwise we store z in entry $T_{3-b}[h_{3-b}(z)]$ and so on, as in the above description of the standard cuckoo hashing. After L elements have been moved, we place the current "nestless" element at the *head* of the queue, together with a bit indicating the next table to which it should be inserted, and terminate the insertion procedure (note that it may take less than L moves, if the queue becomes empty). An important ingredient in the construction of Arbitman et al. is the use of a small auxiliary data structure called "stash" that enables to avoid rehashing, as suggested by Kirsch et al. [30].

A schematic diagram of our construction and a formal description of its procedures are provided in the longer version of this paper [2], where we prove the following theorem:

Theorem III.1. *For any n and $0 < \epsilon < 1$ there exists a dynamic dictionary with the following properties:*

1) *The dictionary stores n elements using $(1+\epsilon)n$ memory words.*
2) *For any polynomial $p(n)$ and for any sequence of at most $p(n)$ operations in which at any point in time at most n elements are stored in the dictionary, with probability at least $1 - 1/p(n)$ over the randomness of the initialization phase, all insertions are performed in time $O(\log(1/\epsilon)/\epsilon^2)$ in the worst case. Deletions and lookups are always performed in time $O(\log(1/\epsilon)/\epsilon^2)$ in the worst case.*

IV. De-amortized Perfect Hashing: Eliminating the Dependency on ϵ

The dependency on ϵ in the deletion and lookup times can be eliminated by using a perfect hashing scheme (with a succinct representation) in each of the first-level bins. Upon storing an element in one of the bins, the insertion procedure reconstructs the perfect hash function for this bin. As long as the reconstruction can be done in time linear in the size of a bin, then the insertion procedure still takes time $O(d) = O(\log(1/\epsilon)/\epsilon^2)$ in the worst case, and the deletion and lookup procedures take constant time that is independent of ϵ. Such a solution, however, does not eliminate the dependency on ϵ in the insertion time.

In this section we present an augmentation that completely eliminates the dependency on ϵ. We present a rather general technique for de-amortizing a perfect hashing scheme to be used in each of the first-level bins. Our approach relies on the fact that the same scheme is employed in a rather large number of bins at the same time, and this enables us to use a single queue to guarantee that even insertions are performed in constant time that is independent of ϵ. Using this augmentation we immediately obtain the following refined variant of Theorem III.1 (the restriction $\epsilon = \Theta((\log\log n/\log n)^{1/2})$ is due to the specific scheme that we de-amortize – see more details below):

Theorem IV.1. *For any integer n there exists a dynamic dictionary with the following properties:*

1) *The dictionary stores n elements using $(1+\epsilon)n$ memory words, for $\epsilon = \Theta((\log\log n/\log n)^{1/2})$.*
2) *For any polynomial $p(n)$ and for any sequence of at most $p(n)$ operations in which at any point in time at most n elements are stored in the dictionary, with probability at least $1-1/p(n)$ over the randomness of the initialization phase, all operations are performed in constant time, independent of ϵ, in the worst case.*

This augmentation is rather general and we can use any perfect hashing scheme with two natural properties. We require that for any sequence σ of operations leading to a set S of size at most $d-1$, for any sequence of memory configurations and rehashing times occurring during the execution of σ, and for any element $x \notin S$ that is currently being inserted it holds that:

- **Property 1:** With probability $1-O(1/d)$ the current hash function can be adjusted to support the set $S \cup \{x\}$ in

expected constant time. In addition, the adjustment time in this case is always upper bounded by $O(d)$.

- **Property 2:** With probability $O(1/d)$ rehashing is required, and the rehashing time is dominated by $O(d) \cdot Z$ where Z is a geometric random variable with a constant expectation.

Our augmentation introduces an overhead which imposes a restriction on the range of possible values for ϵ. The restriction comes from two sources: the description length of the perfect hash function in every bin, and the computation time of the hash function and its adjustment on every insertion. We propose a specific scheme that satisfies the above properties, and can handle $\epsilon = \Omega((\log\log n/\log n)^{1/2})$. It is rather likely that various other schemes such as [14], [23] can be slightly modified to satisfy these properties. In particular, the schemes [42], [51] seem especially suitable for this purpose.

To de-amortize any scheme that satisfies these two properties we use an auxiliary queue (the same queue is used for all bins), and the insertion procedure to the bins is now defined as follows: upon insertion, the new element is placed at the *back* of the queue, and we perform a constant number of steps (denoted by L) on the element currently located at the *head* of the queue. If these L steps are not enough to insert this element into its bin, we return it to the *head* of the queue, and continue working on this element upon the next insertion. If we managed to insert this element by using less than L steps, we continue with the next element and so on until we complete L steps[7]. As for deletions, these are also processed using the queue, and when deleting an element we simply locate the element inside its bin and mark it as deleted (i.e., deletions are always performed in constant time). For more details we refer the reader to the longer version of this paper [2].

A. A Specific Scheme for $\epsilon = \Omega((\log\log n/\log n)^{1/2})$

The scheme uses exactly d memory words to store d elements, and 3 additional words to store the description of its hash function. The elements are mapped into the set $[d]$ using two functions. The first is a pairwise independent function h mapping the elements into the set $[d^2]$. This function can be described using 2 memory words and evaluated in constant time. The second is a function g that records for each $r \in [d^2]$ for which there is a stored element x with $h(x) = r$ the location of x in $[d]$. The description of g consists of at most d pairs taken from $[d^2] \times [d]$ and therefore can be represented using $3d\log d$ bits.

The lookup operation of an element x computes $h(x) = r$ and then $g(r)$ to check if x is stored in that location. In general, we cannot assume that the function g can be evaluated in constant time, and therefore we also store a lookup table for its evaluation. This table is shared by all the bins, and it represents the function that takes as input the description of g and a value r, and outputs $g(r)$ or null. The size of this lookup table is $2^{3d\log d+2\log d} \cdot \log d$ bits. The deletion operation performs a lookup for x, and then updates the description of g. Again, for updating the description of g we use another lookup table (shared among all bins) that takes as input the current description of g and a value $r = h(x)$, and outputs a new description for g. The size of this lookup table is $2^{3d\log d+2\log d} \cdot 3d\log d$ bits.

As for the insert operation, in the longer version of this paper [2] we prove that with probability $1 - O(1/d)$ a new element will not introduce a collision for the function h. In this case we store the new element in the next available entry of $[d]$, and update the description of g. For identifying the next available entry we use a global lookup table of size $2^d \log d$ bits (each row in the table corresponds to an array of d bits describing the occupied entries of a bin), and for updating the description of g we use a lookup table of size $2^{3d\log d+2\log d} \cdot 3d\log d$ bits as before. With probability $O(1/d)$ when inserting a new element we need to rehash by sampling a new function h, and executing the insert operation on all the elements. In this case the rehashing time is upper bounded by $O(d) \cdot Z$ where Z is a geometric random variable with a constant expectation. Thus, this scheme satisfies the two properties stated in the beginning of the section.

The total amount of space used by the global lookup tables is $O(2^{3d\log d+2\log d} \cdot d\log d)$ bits. For $\epsilon = \Omega((\log\log n/\log n)^{1/2})$ this is at most n^α bits for some constant $0 < \alpha < 1$, and therefore negligible compared to our space consumption. In addition, the hash function of every bin is described using $2\log u + d\log d$ bits, and therefore summing over all $m = \lceil(1+\epsilon/2)n/d\rceil$ bins this is $O(n/d \cdot \log u + n\log d)$. For $\epsilon = \Omega(\log\log n/\log n)$ this is at most $\epsilon n \log u$ bits, which is again negligible compared to our space consumption. Thus, this forces the restriction $\epsilon = \Omega((\log\log n/\log n)^{1/2})$.

V. Matching the Information-Theoretic Space Bound

In this section we present a variant of our construction that uses only $(1+o(1))\mathcal{B}$ bits, where $\mathcal{B} = \mathcal{B}(u,n)$ is the information-theoretic bound for representing a set of size n taken from a universe of size u, and guarantees constant-time operations in the worst case with high probability. We first present a scheme that is based on truly random permutations, and then present a scheme that is based on k-wise δ-dependent permutations. We prove the following theorem:

Theorem V.1. *For any integers u and $n \leq u$ there exists a dynamic dictionary with the following properties:*

1) *The dictionary stores n elements taken from a universe of size u using $(1+\epsilon)\mathcal{B}$ bits, where $\mathcal{B} = \lceil\log\binom{u}{n}\rceil$ and $\epsilon = \Theta(\log\log n/(\log n)^{1/3})$.*
2) *For any polynomial $p(n)$ and for any sequence of at most $p(n)$ operations in which at any point in time at most n elements are stored in the dictionary, with probability at least $1-1/p(n)$ over the randomness of the initialization*

[7] A comment is in place regarding rehashing. If rehashing is needed, then we copy the content of the rehashed bin to a dedicated memory location, perform the rehash, and then copy back the content of the bin, and all this is done in several phases of L steps. Note that the usage of the queue guarantees that at any point in time we rehash at most one bin.

phase, all operations are performed in constant time, independent of ϵ, in the worst case.

One of the ideas we will utilize is that when we apply a permutation π to an element x we may think of $\pi(x)$ as a new identity for x, provided that we are also able to compute $\pi^{-1}(x)$. The advantage is that we can now store explicitly only part of $\pi(x)$, where the remainder is stored *implicitly* by the location where the value is stored. This is the idea behind quotient hash functions, as suggested previously by Pagh [43] and Demaine et al. [11].

A. A Scheme based on Truly Random Permutations

Recall that our construction consists of two levels: a table in the first level that contains $m \approx n/d$ bins, each of which stores at most d elements, and the de-amortized cuckoo hashing in the second level for dealing with the overflowing elements. The construction described in this section shares the same structure, while refining the memory consumptions in each of the two levels separately. In turn, Theorem V.1 (assuming truly random permutations for now) follows immediately by plugging in the following modifications to our previous schemes.

1) First-Level Hashing Using Permutations: We reduce the space consumption in the first level of our construction by hashing the elements into the first-level table using a "chopped" permutation π over the universe $\mathcal{U}$ as follows. For simplicity we assume here that u and m are powers of 2, and refer the reader to the longer version of this paper [2] for an extension to the more general case. Given a permutation π and an element $x \in \mathcal{U}$, we denote by $\pi_L(x)$ the left-most $\log m$ bits of $\pi(x)$, and by $\pi_R(x)$ the right-most $\log(u/m)$ bits of $\pi(x)$. That is, $\pi(x)$ is the concatenation of the bit-strings $\pi_L(x)$ and $\pi_R(x)$. We use π_L as the function mapping elements into bins, and π_R as the identity of the elements inside the bins: any element x is stored either in the first level in bin $\pi_L(x)$ using the identity $\pi_R(x)$, or in the second level if its first-level bin already contains d other elements. The update and lookup procedures remain exactly the same, and note that the correctness of the lookup procedure is guaranteed by the fact that π is a permutation, and therefore the function π_R is one-to-one inside every bin.

In the following lemma we bound the number of overflowing elements in the first level when using a truly random permutation. Recall that an element is overflowing if it is mapped to a bin with at least d other elements. The lemma guarantees that by setting $d = O(\log(1/\epsilon)/\epsilon^2)$ there are at most $\epsilon n/16$ overflowing elements with an overwhelming probability, exactly as in Section III.

Lemma V.2. *Fix any n, d, ϵ, and a set $S \subseteq \mathcal{U}$ of n elements. With probability $1 - 2^{-\omega(\log n)}$ over the choice of a truly random permutation π, when using the function π_L for mapping the elements of S into $m = \lceil (1+\epsilon)n/d \rceil$ bins of size d, the number of* non-overflowing *elements is at least $(1 - \epsilon/32)(1 - 4e^{-\Omega(\epsilon^2 d)})n$.*

2) The Bins in the First-Level Table: We follow the general approach presented in Section IV to guarantee that the update and lookup operations on the first-level bins are performed in constant time that is independent of the size of the bins (and thus independent of ϵ). Depending on the ratio between the size of the universe u and the number of elements n, we present hashing schemes that satisfy the two properties stated in the beginning of Section IV. Our task here is a bit more subtle than in Section IV, since we must guarantee that the descriptions of the hash functions inside the bins (and any global lookup tables that are used) do not occupy too much space compared to the information-theoretic bound. This puts a restriction on the size of the bins. We consider two cases (these cases are not necessarily mutually exclusive):

Case 1: $u \leq n \cdot 2^{(\log n)^\beta}$ for some $\beta < 1$. In this case we store all elements in a single word using the information-theoretic representation, and use lookup tables to guarantee constant time operations. Specifically, recall that the elements in each bin are now taken from a universe of size u/m, and each bin contains at most d elements. Thus, the content of a bin can be represented using $\lceil \log \binom{u/m}{d} \rceil$ bits. Insertions and deletions are performed using a global lookup table that is shared among all bins. The table represents a function that receives as input a description of a bin, and an additional element, and outputs an updated description for the bin. This lookup table can be represented using $2^{\lceil \log \binom{u/m}{d} \rceil + \lceil \log \frac{u/m}{d} \rceil} \cdot \lceil \log \binom{u/m}{d} \rceil$ bits. Similarly, lookups are performed using a global table that occupies $2^{\lceil \log \binom{u/m}{d} \rceil + \lceil \log \frac{u/m}{d} \rceil}$ bits.

These force two restrictions on d. First, the description of a bin has to fit into one memory word, to enable constant-time evaluation using the lookup tables. Second, the two lookup tables have to fit into at most, say, $(\epsilon/6) \cdot n \log(u/n)$ bits. When assuming that $u \leq n \cdot 2^{(\log n)^\beta}$ for some $\beta < 1$, these two restrictions allow $d = O((\log n)^{1-\beta})$. Recall that $d = O(\log(1/\epsilon)/\epsilon^2)$, and this implies that $\epsilon = \Omega\left(\frac{(\log\log n)^{1/2}}{(\log n)^{(1-\beta)/2}}\right)$.

Case 2: $u > n \cdot 2^{(\log n)^\beta}$ for some $\beta < 1$. In this case we use the scheme described in Section IV-A. In every bin the pairwise independent function f can be represented using $2\lceil \log(u/m) \rceil$ bits (as opposed to $2\lceil \log u \rceil$ bits in Section IV-A), and the function g can be represented using $3d\lceil \log d \rceil$ bits (as in Section IV-A). Summing these over all m bins results in $O(n/d \cdot \log(u/n) + n \log d)$ bits, and therefore the first restriction is that the latter is at most, say, $(\epsilon/12) \cdot n \log(u/n)$ bits. Assuming that $u > n \cdot 2^{(\log n)^\beta}$ for some $\beta < 1$ (and recall that $d = O(\log(1/\epsilon)/\epsilon^2)$) this allows $\epsilon = \Omega\left(\frac{\log\log n}{(\log n)^\beta}\right)$.

In addition, as discussed in Section IV-A, the scheme requires global lookup tables that occupy a total $O(2^{3d \log d + 2\log d} \cdot d \log d)$ bits, and therefore the second restriction is that the latter is again at most $(\epsilon/12) \cdot n \log(u/n)$ bits. This allows $d = O(\log n / \log\log n)$, and therefore $\epsilon = \Omega\left(\left(\frac{\log\log n}{\log n}\right)^{1/2}\right)$. Thus, in this case we can deal with $\epsilon = \Omega\left(\max\left\{\frac{\log\log n}{(\log n)^\beta}, \left(\frac{\log\log n}{\log n}\right)^{1/2}\right\}\right)$.

An essentially optimal trade off (asymptotically) between these two cases occurs for $\beta = 1/3$, which allows $\epsilon = \Omega\left(\frac{(\log\log n)^{1/2}}{(\log n)^{1/3}}\right)$ in the first case, and $\epsilon = \Omega\left(\frac{\log\log n}{(\log n)^{1/3}}\right)$ in the second case. Therefore, regardless of the ratio between u and n, our construction can always allow $\epsilon = \Omega\left(\frac{\log\log n}{(\log n)^{1/3}}\right)$.

3) The Second Level: Permutation-based Cuckoo Hashing: First of all note that if $u > n^{1+\alpha}$ for some constant $\alpha < 1$, then $\log u \leq (1/\alpha + 1)\log(u/n)$, and therefore we can allow ourselves to store $\alpha\epsilon n$ overflowing elements using $\log u$ bits each as before. For the general case, we present a variant of the de-amortized cuckoo hashing scheme that is based on permutations, where each element is stored using roughly $\log(u/n)$ bits instead of $\log u$ bits[8]. Recall that cuckoo hashing uses two tables T_1 and T_2, each consisting of $r = (1+\delta)\ell$ entries for some small constant $\delta > 0$ for storing a set $S \subseteq \mathcal{U}$ of at most ℓ elements, and two hash functions $h_1, h_2 : \mathcal{U} \to [r]$. An element x is stored either in entry $h_1(x)$ of table T_1 or in entry $h_2(x)$ of table T_2. This naturally defines the cuckoo graph, which is the bipartite graph defined on $[r] \times [r]$ with edges $\{(h_1(x), h_2(x))\}$ for every $x \in S$.

We modify cuckoo hashing to use permutations as follows (for simplicity we again assume that u and r are powers of 2, but this is not essential). Given two permutations π_1 and π_2 over $\mathcal{U}$, we define h_1 as the left-most $\log r$ bits of π_1, and h_2 as the left-most $\log r$ bits of π_2. An element x is stored either in entry $h_1(x)$ of table T_1 using the right-most $\log(u/r)$ bits of $\pi_1(x)$ as its new identity, or in entry $h_2(x)$ of table T_2 using the right-most $\log(u/r)$ bits of $\pi_2(x)$ as its new identity. The update and lookup procedures are naturally defined as before. Note that the permutations π_1 and π_2 have to be easily invertible to allow moving elements between the two tables, and this is satisfied by our constructions of k-wise δ-dependent permutations. We now argue that by slightly increasing the size r of each table, the de-amortization of cuckoo hashing (and, in particular, cuckoo hashing itself) still has the same performance guarantees when using permutations instead of functions. The de-amortization of [1] relies on two properties of the cuckoo graph:

1) With high probability the sum of sizes of any $\log \ell$ connected components is $O(\log \ell)$.
2) The probability that there are at least s edges that close a second cycle is $O(r^{-s})$.

These properties are known to be satisfied when h_1 and h_2 are truly random functions, and here we present a coupling argument showing that they are satisfied also when h_1 and h_2 are defined as above using truly random permutations. Our argument relies on the monotonicity of these properties: if they are satisfied by a graph, then they are also satisfied by all its subgraphs. In the longer version of this paper [2] we prove the following claim:

Claim V.3. *Let $\ell = \lceil \epsilon n/16 \rceil$ and $r = \lceil (1+\delta)(1+\epsilon)\ell \rceil$ for some constant $0 < \delta < 1$. There exists a joint distribution $\mathcal{D} = (\mathcal{G}_{f_1,f_2}, \mathcal{G}_{\pi_1,\pi_2})$ such that:*

- *$\mathcal{G}_{f_1,f_2}$ is identical to the distribution of cuckoo graphs over $[r] \times [r]$ with $\lceil (1+\epsilon)\ell \rceil$ edges, defined by h_1 and h_2 that are the left-most $\log r$ bits of two truly random functions $f_1, f_2 : \mathcal{U} \to \mathcal{U}$.*
- *$\mathcal{G}_{\pi_1,\pi_2}$ is identical to the distribution of cuckoo graphs over $[r] \times [r]$ with ℓ edges, defined by h_1 and h_2 that are the left-most $\log r$ bits of two truly random permutations $\pi_1, \pi_2 : \mathcal{U} \to \mathcal{U}$.*
- *With probability $1 - e^{-\Omega(\epsilon^3 n)}$ over the choice of $(G_{f_1,f_2}, G_{\pi_1,\pi_2}) \leftarrow \mathcal{D}$, it holds that G_{π_1,π_2} is a subgraph of G_{f_1,f_2}.*

B. A Scheme based on k-wise δ-dependent Permutations

We eliminate the need for truly random permutations by first reducing the problem of dealing with n elements to several instances of the problem on n^α elements, for some $\alpha < 1$. Then, for each such instance we apply the solution that assumes truly random permutations, but using a k-wise δ-dependent permutation, for $k = n^\alpha$ and $\delta = 1/\mathrm{poly}(n)$, that can be shared among all instances. Although the following discussion can be framed in terms of any small constant $\alpha < 1$, for concreteness we use $\alpha \approx 1/10$.

Specifically, we hash the elements into $m = n^{9/10}$ bins of size at most $d = n^{1/10} + n^{3/40}$ each, using a permutation $\pi : \mathcal{U} \to \mathcal{U}$ sampled from a collection Π of one-round Feistel permutations, and prove that with overwhelming probability there are no overflowing bins. The collection Π is defined as follows. For simplicity we assume that u and m are powers of 2, and refer the reader to the longer version of this paper [2] for an extension to the more general case. Let $\mathcal{F}$ be a collection of k'-wise independent functions $f : \{0,1\}^{\log(u/m)} \to \{0,1\}^{\log m}$, where $k' = O(n^{1/20})$, with a short representation and constant evaluation time (see Section II). Given an input $x \in \{0,1\}^{\log u}$ we denote by x_L its left-most $\log m$ bits, and by x_R its right-most $\log(u/m)$ bits. For every $f \in \mathcal{F}$ we define a permutation $\pi = \pi_f \in \Pi$ by $\pi(x) = (x_L \oplus f(x_R), x_R)$. Any element x is mapped to the bin $\pi_L(x) = x_L \oplus f(x_R)$, and is stored there using the identity $\pi_R(x) = x_R$.

Then, in every bin we apply the scheme from Section V-A that relies on truly random permutations, but using three k-wise δ-dependent permutations that are shared among all bins (recall that the latter scheme requires three permutations: one for its first-level hashing, and two for its permutation-based cuckoo hashing that stores the overflowing elements)[9]. By setting $k = n^{1/10} + n^{3/40}$ it holds that the distribution inside every bin is δ-close in statistical distance to that when using truly random permutations. Therefore, Lemma V.2 and Claim V.3 guarantee that these permutations provide the required performance guarantees for each bin with probability $1 - (2^{-\omega(\log n)} + \delta) = 1 - 1/\mathrm{poly}(n)$. Thus, applying a union bound on all m bins yields the same performance guarantees as

[8]There is also an auxiliary data structure (a queue) that contains roughly $\log n$ elements, each of which can be represented using $\log u$ bits.

[9]When dealing with a universe of size $u \leq n^{1+\gamma}$ for a small constant $\gamma < 1$, we can even store three *truly random* permutations, but this solution does not extend to the more general case where u/m might be rather large.

in Section V-A with probability $1-1/\mathrm{poly}(n)$, for an arbitrary large polynomial. In the longer version of this paper [2] we prove the following claim :

Claim V.4. *Fix u and $n \leq u$, let $m = n^{9/10}$, and let $\mathcal{F}$ be a collection of k'-wise independent functions $f : \{0,1\}^{\log(u/m)} \to \{0,1\}^{\log m}$ for $k' = \lfloor n^{1/20}/e^{1/3} \rfloor$. For any set $S \subset \{0,1\}^{\log u}$ of size n, with probability $1 - 2^{-\omega(\log n)}$ over the choice of $f \in \mathcal{F}$, when using the function $x \mapsto x_L \oplus f(x_R)$ for mapping the elements of S into m bins, no bin contains more than $n^{1/10} + n^{3/40}$ elements.*

We note that a possible (but not essential) refinement is to combine the queues of all m bins. Recall that each bin has two queues: a queue for its de-amortized cuckoo hashing, and a queue for its first-level bins. An analysis almost identical to that of [1] (for the de-amortized cuckoo hashing) and of Section IV (for the first-level bins) shows that we can in fact combine all the queues of the de-amortized cuckoo hashing schemes, and all the queues of the first-level bins.

VI. Concluding Remarks and Open Problems

Implications of our constructions for the amortized setting. We note that our constructions offer various advantages over previous constructions even in the amortized setting, where one is not interested in worst-case guarantees. In particular, instantiating our dictionary with the classical cuckoo hashing [44] (instead of its de-amortized variant [1]) already gives a logarithmic upper bound *with high probability* for the update time, together with a space consumption of $(1+\epsilon)n$ memory words for a *sub-constant* ϵ.

On the practicality of our schemes. In this paper we concentrated on showing that it is possible to obtain a succinct representation with worst-case operations. The natural question is how applicable these methods are. There are a number of approaches that can be applied to reduce the overflow of the first-level bins. First, we can use the two-choice paradigm (or, more generally, d-choice) in the first-level bins instead of the single function we currently employ. Another alternative is to apply the generalized cuckoo hashing [18] inside the first-level bins, limiting the number of moves to a small constant, and storing the overflowing elements in de-amortized cuckoo hashing as in our actual construction. Experiments we performed indicate that these approaches result in (sometimes quite dramatic) improvements. The experiments suggest that for the latter variant, maintaining a small queue of at most logarithmic size, enables us even to get rid of the second-level cuckoo hashing: i.e., an element can reside in one of two possible first-level bins, or in the queue.

Another natural tweak is using a single queue for all the de-amortizations together. Finally, while the use of chopped permutations introduces only a negligible overhead, the use of an intermediate level seems redundant and we conjecture that better analysis would indeed show that.

Clocked adversaries. The worst-case guarantees of our dictionary are important if one wishes to protect against "clocked adversaries", as mentioned in Section I. This in itself can yield a solution in the following sense: have an upper bound α on the time each memory access takes, and then make sure that all requests are answered in time exactly α times the worst-case upper bound on the number of memory probes. Such an approach, however, may be quite wasteful in terms of computing resources, since we are not taking advantage of the fact that some operations may be processed in time that is below the worst-case guarantee. In addition, this approach ignores the memory hierarchy, that can possibly be used to our advantage.

Lower bounds for dynamic dictionaries. The worst-case performance guarantees of our constructions are satisfied with all but an arbitrary small polynomial probability over the randomness of their initialization phase. There are several open problems that arise in this context. One problem is to reduce the failure probability to sub-polynomial. The main bottleneck is the approximation to k-wise functions or permutations. Another bottleneck is the lookup procedure of the queue (if the universe is of polynomial size then we can in fact maintain a small queue deterministically). Another problem is to identify whether randomness is needed at all. That is, whether it is possible to construct a deterministic dictionary with similar guarantees. We conjecture that randomness is necessary. Various non-constant lower bounds on the performance of deterministic dynamic dictionaries are known for several models of computation [14], [38], [49]. Although these models capture a wide range of possible constructions, for the most general cell probe model [52] it is still an open problem whether a non-constant lower bound can be proved.

Extending the scheme to smaller values of ϵ. Recall that in the de-amortized construction of perfect hashing inside the first-level bins (Section IV), we suggested a specific scheme that can handle $\epsilon = \Omega((\log\log n/\log n)^{1/2})$. This restriction on ϵ was dictated by the space consumption of the global lookup tables together with the hash functions inside each bin. The question is how small can ϵ be and how close to the information theoretic bound can we be, that is for what function f can we use $\mathcal{B} + f(n,u)$ bits. A possible approach is to use the two-choice paradigm for reducing the number of overflowing elements from the first level of our construction, as already mentioned.

Constructions of k-wise almost independent permutations. In the longer version of this paper [2] we present a construction of k-wise δ-dependent permutations with a succinct representation and a constant evaluation time. Two natural open problems are to allow larger values of k (the main bottlenecks are the restrictions $k < u^{1/2}$ in [40] and $k \leq n^{\alpha}$ in [48]), and a sub-polynomial δ (the main bottleneck is the failure probability of Siegel's construction [48]).

Supporting dynamic resizing. In this paper we assumed that there is a pre-determined bound on the maximal number of stored elements. It would be interesting to construct a dynamic dictionary with constant worst-case operations and full memory utilization *at any point in time*. That is, at any

point in time if there are ℓ stored elements then the dictionary occupies $(1 + o(1))\ell$ memory words (even more challenging requirement may be to use only $(1 + o(1))\mathcal{B}(u, \ell)$ bits of memory, where $\mathcal{B}(u, \ell)$ is the information-theoretic bound for representing a set of size ℓ taken from a universe of size u). This requires designing a method for dynamic resizing that essentially does not incur any noticeable time or space overhead in the worst case. We note that in our construction it is rather simple to dynamically resize the bins in the first-level table, and this provides some flexibility.

ACKNOWLEDGMENTS

We thank Rasmus Pagh and Udi Wieder for many useful remarks and suggestions.

REFERENCES

[1] Y. Arbitman, M. Naor, and G. Segev, "De-amortized cuckoo hashing: Provable worst-case performance and experimental results," in *36th ICALP*, 2009, pp. 107–118.

[2] ——, "Backyard cuckoo hashing: Constant worst-case operations with a succinct representation," arXiv report 0912.5424, 2010.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *INFOCOM*, 2001, pp. 1454–1463.

[5] A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM J. Comput.*, vol. 28, no. 5, pp. 1627–1640, 1999.

[6] J. A. Cain, P. Sanders, and N. C. Wormald, "The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation," in *18th SODA*, 2007, pp. 469–476.

[7] L. Carter, R. W. Floyd, J. Gill, G. Markowsky, and M. N. Wegman, "Exact and approximate membership testers," in *10th STOC*, 1978, pp. 59–65.

[8] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 275–288, 1992.

[9] K. Dalal, L. Devroye, E. Malalla, and E. McLeis, "Two-way chaining with reassignment," *SIAM J. Comput.*, vol. 35, no. 2, pp. 327–340, 2005.

[10] E. Demaine, "Lecture notes for the course "Advanced data structures"," Available at http://courses.csail.mit.edu/6.851/spring07/scribe/lec21.pdf, 2007.

[11] E. D. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Pătraşcu, "De dictionariis dynamicis pauco spatio utentibus (*lat.* On dynamic dictionaries using little space)," in *7th LATIN*, 2006, pp. 349–361.

[12] L. Devroye and E. Malalla, "On the k-orientability of random graphs," *Discrete Mathematics*, vol. 309, no. 6, pp. 1476–1490, 2009.

[13] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via XORSAT," in *37th ICALP*, 2010, pp. 213–225.

[14] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," *SIAM J. Comput.*, vol. 23, no. 4, pp. 738–761, 1994.

[15] M. Dietzfelbinger and F. Meyer auf der Heide, "A new universal class of hash functions and dynamic hashing in real time," in *17th ICALP*, 1990, pp. 6–19.

[16] M. Dietzfelbinger and R. Pagh, "Succinct data structures for retrieval and approximate membership," in *35th ICALP*, 2008, pp. 385–396.

[17] M. Dietzfelbinger and M. Rink, "Applications of a splitting trick," in *36th ICALP*, 2009, pp. 354–365.

[18] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," *Theor. Comput. Sci.*, vol. 380, no. 1-2, pp. 47–68, 2007.

[19] M. Dietzfelbinger and P. Woelfel, "Almost random graphs with simple hash functions," in *35th STOC*, 2003, pp. 629–638.

[20] D. Fernholz and V. Ramachandran, "The k-orientability thresholds for $G_{n,p}$," in *18th SODA*, 2007, pp. 459–468.

[21] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis, "Space efficient hash tables with worst case constant access time," *Theor. Comput. Sci.*, vol. 38, no. 2, pp. 229–248, 2005.

[22] N. Fountoulakis and K. Panagiotou, "Sharp load thresholds for cuckoo hashing," arXiv report 0910.5147, 2009.

[23] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with $O(1)$ worst case access time," *J. ACM*, vol. 31, no. 3, pp. 538–544, 1984.

[24] A. Frieze and P. Melsted, "Maximum matchings in random bipartite graphs and the space utilization of cuckoo hashtables," arXiv report 0910.5535, 2009.

[25] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *13th APPROX-RANDOM*, 2009, pp. 490–503.

[26] T. Hagerup, "Sorting and searching on the word RAM," in *15th STACS*, 1998, pp. 366–398.

[27] T. Hagerup, P. B. Miltersen, and R. Pagh, "Deterministic dictionaries," *J. Algorithms*, vol. 41, no. 1, pp. 69–85, 2001.

[28] E. Kaplan, M. Naor, and O. Reingold, "Derandomized constructions of k-wise (almost) independent permutations," *Algorithmica*, vol. 55, no. 1, pp. 113–133, 2009.

[29] A. Kirsch and M. Mitzenmacher, "Using a queue to de-amortize cuckoo hashing in hardware," in *45th Allerton Conf.*, 2007, pp. 751–758.

[30] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM J. Comput.*, vol. 39, no. 4, pp. 1543–1561, 2009.

[31] D. E. Knuth, "Notes on "open" addressing," Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4899, 1963.

[32] ——, *The Art of Computer Programming. Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, 1998.

[33] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO '96*, 1996, pp. 104–113.

[34] E. Lehman and R. Panigrahy, "3.5-way cuckoo hashing for the price of 2-and-a-bit," in *17th ESA*, 2009, pp. 671–681.

[35] R. J. Lipton and J. F. Naughton, "Clocked adversaries for hashing," *Algorithmica*, vol. 9, no. 3, pp. 239–252, 1993.

[36] S. Lovett and E. Porat, "A lower bound for dynamic Bloom filters," in *51st FOCS*, 2010, to appear.

[37] M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM J. Comput.*, vol. 17, no. 2, pp. 373–386, 1988.

[38] K. Mehlhorn, S. Näher, and M. Rauch, "On the complexity of a game related to the dictionary problem," *SIAM J. Comput.*, vol. 19, no. 5, pp. 902–906, 1990.

[39] P. B. Miltersen, "Cell probe complexity - a survey," in *19th FSTTCS*, 1999.

[40] M. Naor and O. Reingold, "On the construction of pseudorandom permutations: Luby-Rackoff revisited," *J. Cryptology*, vol. 12, no. 1, pp. 29–66, 1999.

[41] A. Pagh and R. Pagh, "Uniform hashing in constant time and optimal space," *SIAM J. Comput.*, vol. 38, no. 1, pp. 85–96, 2008.

[42] R. Pagh, "Hash and displace: Efficient evaluation of minimal perfect hash functions," in *6th WADS*, 1999, pp. 49–54.

[43] ——, "Low redundancy in static dictionaries with constant query time," *SIAM J. Comput.*, vol. 31, no. 2, pp. 353–363, 2001.

[44] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[45] R. Panigrahy, "Efficient hashing with lookups in two memory accesses," in *16th SODA*, 2005, pp. 830–839.

[46] M. Pătraşcu, "Succincter," in *49th FOCS*, 2008, pp. 305–313.

[47] R. Raman and S. S. Rao, "Succinct dynamic dictionaries and trees," in *30th ICALP*, 2003, pp. 357–368.

[48] A. Siegel, "On universal classes of extremely random constant-time hash functions," *SIAM J. Comput.*, vol. 33, no. 3, pp. 505–543, 2004.

[49] R. Sundar, "A lower bound for the dictionary problem under a hashing model," in *32nd FOCS*, 1991, pp. 612–621.

[50] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[51] P. Woelfel, "Maintaining external memory efficient hash tables," in *10th APPROX-RANDOM*, 2006, pp. 508–519.

[52] A. C.-C. Yao, "Should tables be sorted?" *J. ACM*, vol. 28, no. 3, pp. 615–628, 1981.

[53] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *6th USENIX Conference on File and Storage Technologies*, 2008, pp. 269–282.