KTH - DD2467 Individual Project in Theoretical Computer Science

Backyard Cuckoo Hashing - Theory and Practice

Marc Gröling

Supervisor: Ioana-Oriana Bercea

January 2025

Abstract

Dynamic dictionaries (e.g. hash tables) are data structures that can efficiently support lookup-queries on dynamic sets and are widely used in the area of computer science. Their performance is usually measured by their update time, lookup time, and space consumption. This work examines a theoretical construction of a dynamic dictionary that is the first to achieve optimality in all three aspects. Backyard cuckoo hashing, that was recently proposed by Arbitman, Naor, and Segev [1], is a two-level variant of cuckoo hashing that only uses $(1 + \epsilon)n$ memory words and guarantees constant-time operations in the worst case, with high probability. Furthermore, two augmentations of it are discussed. The first one is a de-amortized perfect hashing scheme, which eliminates the dependency on ϵ for the time complexity of the operations. And the second one reduces the space usage to $(1 + o(1))\mathcal{B}$, where $\mathcal{B}$ denotes the information-theoretic lower bound for representing a set of size n taken from a universe of size u. Additionally, we provide an implementation of the construction in C++ and discuss implementation details and insights. Finally, we present two experiments, which shed light on how to parametrize the construction for good practical performance.

Key Words: data structures, dynamic dictionaries, cuckoo hashing, de-amortization, hash functions

Contents

1 Introduction

Checking whether an element is contained in a set of elements S is a common problem in computer science [2]. This problem can be stated in the static and dynamic case. In the latter case, there may be insertions, deletions, and lookup. One data structure that is especially suitable to solve this problem are hash tables (which are an implementation of dynamic dictionaries). These employ a hash function which maps elements of the input domain, the universe $\mathcal{U}$, to a memory location, where the element is then stored. Usually, the employed hash functions are imperfect (since perfect hash functions are only practical for small sets), which means that two elements of S may hash to the same location. Resolving these collisions can be done in many ways, e.g. linear probing, separate chaining, cuckoo hashing. These variants offer different guarantees on the time and space complexity of the resulting data structure.

The performance of a dynamic dictionary can be measured by its update time, lookup time, and space consumption. Ideally, one would want to use space linear in the number of stored elements and support constant time operations. While obtaining an optimal solution for any one of these criteria is rather simple, obtaining it for all three is more complicated.

In this work, a dynamic dictionary is examined that promises optimality in all three performance aspects. Backyard cuckoo hashing is a two-level variant of cuckoo hashing [3] that only uses $(1 + \epsilon)n$ memory words (where n is the maximum number of elements that may be stored). Furthermore, it guarantees constant-time operations with high probability.

The main contributions of this work are:

- We provide an intuitive and easily readable analysis of the backyard construction. We do this by going all the way from cuckoo hashing and iteratively extending it until we get the backyard construction. We also provide detailed descriptions of the auxiliary data structures, which the original paper does not. For formal proofs, we try to make these more easily understandable and only provide an intuition for the more complicated ones.
- We give an implementation of the backyard construction in C++, which can be found in our GitHub repository[*]. We do not know of another existing implementation.
- We conduct two experiments that helps determine which values might be suitable for the choice of parameters of the backyard construction. The original paper only provides ideas for parametrization using O/Θ notation.

In Section 2, the used model of computation is discussed as well as other preliminaries. This is followed by Section 3, which provides a description of cuckoo hashing and discusses its flaws. This leads to de-amortized cuckoo hashing, which is presented in Section 4 and fixes one of the main flaws of cuckoo hashing of only providing amortized guarantees for insertion time. After that, in Section 5, backyard cuckoo hashing is examined, which fixes the low memory efficiency of de-amortized cuckoo hashing. In the following section, Section 6 implementation details are discussed, which are mainly composed of which hash functions are used in the implementation. Then, in Section 7, two experiments that were conducted are presented, which give more information on how to parametrize the backyard construction. Finally, Section 8 provides a summary of the work and discusses interesting future work regarding this topic.

[*]`https://github.com/mgroling/Backyard-Cuckoo-Hashing`

2 Preliminaries

For the analysis of this work, we use the unit cost RAM model, which is also used in the analysis of [1]. This model assumes that elements come from a universe $\mathcal{U}$ of size u, which can be stored in a single word of length $\omega = \lceil \log u \rceil$ bits. Standard operations on these words (e.g. addition, multiplication, bitwise Boolean operations), are assumed to be executable in constant time, as well as memory accesses.

Furthermore, we use the following notations and definitions:

Notation 2.1. *Let $[m]$ for a number $m \in \mathbb{N}$ denote the set $\{0, 1, \ldots, m-1\}$.*

Definition 2.1. *A dynamic dictionary maintains a set S and supports the following operations for any element $x \in \mathcal{U}$:*

$$
\begin{aligned}
&insert(x) : S \leftarrow S \cup \{x\} \\
&delete(x) : S \leftarrow S \setminus \{x\} \\
&lookup(x) : \textit{return} \begin{cases} yes & \textit{if } x \in S \\ no & otherwise \end{cases}
\end{aligned}
$$

Definition 2.2. *A collection $\mathcal{F}$ of functions $f : U \to V$ is uniform if for every $x \in U$ and $y \in V$, it holds that*

$$Pr_{f \in \mathcal{F}}[f(x) = y] = \frac{1}{|V|}$$

Definition 2.3. *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*

$$Pr_{f \in \mathcal{F}}[f(x_1) = y_1 \wedge \cdots \wedge f(x_k) = y_k] = \frac{1}{|V|^k}$$

3 Cuckoo Hashing

Cuckoo Hashing [3] is a scheme to implement dynamic dictionaries. It uses two tables T_1, T_2 each of size r that are each equipped with a hash function h_1, h_2 chosen from a family of k-wise independent hash functions $\mathcal{H}$ that are of the domain $h_i : \mathcal{U} \to [r]$. One can imagine that these two hash functions give each element a nest (in which it may reside in). As such, an element $x \in \mathcal{U}$ may only be either in $T_1[h_1(x)]$ or $T_2[h_2(x)]$.

In the insertion procedure, an element is first inserted into its nest (determined by h_1) in T_1. If the position was empty before, then the insertion procedure is over. However, if there was an element there, then it is kicked out and is now ”nestless”. This nestless element is now inserted into its position in T_2. Again, if that position was empty before, then the insertion procedure is over. And also as before, if the position was not empty, then the element that resided there, is now ”nestless”. This ”nestless” element is now inserted into its position in T_1. This moving of elements between T_1 and T_2 repeats until an empty position is found or the maximum number of iterations is reached. In the latter case, new hash functions are selected, and all elements are reinserted into the tables. Note, that one insert call may trigger multiple rehashes. An example of an insertion procedure can be seen in Figure 1. Here, the key x is first inserted into T_1, to where y is initially at. y is then moved to the position where z is initially at and z to the empty spot.

In the lookup operation, it is checked whether the element is in one of its two possible positions (it is either in $T_1[h_1(x)]$ or $T_2[h_2(x)]$). For the delete operation, it is checked whether the element is in one of its two positions and if yes, then, it is deleted from there.

The operations of the data structure are also defined in pseudocode in Algorithm 1.

Algorithm 1 Procedures of a Cuckoo Hash Table [3]

```
1: function lookup($x$):
2:     return $T_1[h_1(x)] = x \lor T_2[h_2(x)] = x$;

1: function insert($x$):
2:     if lookup(x) then return
3:     loop $MaxLoop$ times:
4:         $x \leftrightarrow T_1[h_1(x)]$;                ▷ Swap the elements in $x$ and $T_1[h_1(x)]$
5:         if $x = \bot$ then return
6:         $x \leftrightarrow T_2[h_2(x)]$;                ▷ Swap the elements in $x$ and $T_2[h_2(x)]$
7:         if $x = \bot$ then return
8:     rehash(); insert(x);

1: function delete($x$):
2:     if $T_1[h_1(x)] = x$ then $T_1[h_1(x)] = \bot$; return
3:     if $T_2[h_2(x)] = x$ then $T_2[h_2(x)] = \bot$;

1: function rehash():
2:     sample new hash functions $h_1, h_2$ from $\mathcal{H}$
3:     collect all elements in $T_1$ and $T_2$ and save them externally, then remove them from $T_1, T_2$
4:     insert all previously removed elements with the newly sampled hash functions
```

One can see that the *lookup* and *delete* operations only require a constant number of steps in the unit cost RAM model. However, the analysis of the *insert* operation is more complicated, since it is unclear how many iterations are needed to find an empty spot, or even if rehashing is necessary. To understand the time complexity of the operation, it is vital to consider the cuckoo graph.

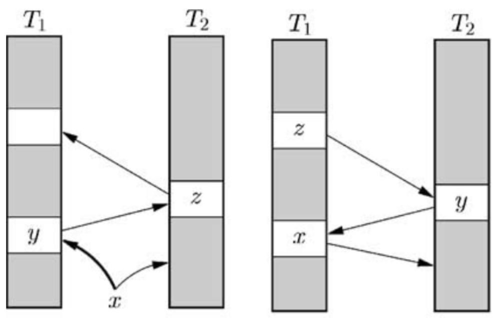


Figure 1: Example of a successful insertion of x into the cuckoo tables, with arrows denoting possible spots where elements could go

Definition 3.1. *Given a set $S \subseteq \mathcal{U}$ and two hash functions, $h_1, h_2 : \mathcal{U} \to [r]$ the cuckoo graph is the bipartite graph $G = (L, R, E)$, where $L = R = [r]$ and $E = \{(h_1(x), h_2(x)) : x \in S\}$. (as in [4])*

Lemma 3.1. *A Set S can be stored in a cuckoo hash table using the hash functions h_1 and h_2 if and only if no connected component of the corresponding cuckoo graph has more edges than nodes.*

Proof Sketch. Suppose no connected component of the cuckoo graph has more edges than nodes, then each connected component has at most one cycle. Also, if S can be stored in a cuckoo hash table, then means that each element is either in its position in T_1 or T_2. One can think of finding a valid way to store S, as assigning a direction to each edge (which represents an element) in the cuckoo graph, such that it is either stored in T_1 or T_2. This is also known as an orientation of an undirected graph, which turns it into a directed graph. If the orientation leads to each node only having ≤ 1 incoming edges, then it is a valid assignment of elements to positions in a cuckoo hash table.

"$\Rightarrow$": Since S can be stored in a hash table, there is a valid direction assignment of each edge in the cuckoo graph. And if a connected component contains more edges than nodes, then there must necessarily be a vertex with more than one incoming edge and as such no valid direction assignment is possible. As such, no connected component of the cuckoo graph contains more edges than nodes.

"$\Leftarrow$": One has to consider 2 cases, either an edge is part of a cycle or it is not part of a cycle. If an edge is part of a cycle, since that component is unicyclic, the edges of that cycle can be directed in such a way that each node only has one incoming edge. If an edge is not part of a cycle, then it must point "away" from any cycle that a path leads to from one of its connected nodes (which is at most one, since otherwise the assumption is hurt). As such, there is a valid direction assignment to each edge, which means that S can be stored in a cuckoo hash table.

Note. Notice that this also defines when the insertion procedure will terminate in a finite amount of steps. That is, if an element $x \in \mathcal{U}$ is inserted and every connected component of the cuckoo graph of $x \cup S$ (with h_1, h_2) has at maximum an equal amount of edges compared to nodes, then the insertion will terminate in a finite number of steps.

As such, the time required for an insertion heavily depends on how sparse the graph is. The load factor of a cuckoo hash table denotes the number of elements in the data structure divided by the number of available positions in the data structure. For a load factor of $1/2$ or lower, it is shown in [3] that the amortized expected time for an insertion is bounded by a constant. Intuitively, this can be explained by the fact that if the graph only has a load factor of $1/2$ or lower, then the average size of a component in the cuckoo graph is small, since the graph is sparse. And the insertion time depends solely on the component of the inserted element, in particular the number of required moves is at most the size of the component (assuming that the component does not have cycles).

There may be larger components for an element and as such a guarantee of constant insertion time cannot be given (in the non-amortized fashion). However, since these larger components occur with only low probability and on average the size of a component is ”small”, the insertion procedure runs in constant amortized time.

4 De-Amortized Cuckoo Hashing

One of the downsides of Cuckoo Hashing (see Section 3) is that, although insertions only take amortized constant time, during the insertion process of n elements, (1) some insertions take $\Omega(\log n)$ time with a noticeable probability. Also, (2) sometimes the data structures has to be re-randomized (rehashed) because an element does not fit into the data structure anymore and all elements need to be inserted again. In both of these cases, the insertion procedure takes longer than constant time. For some applications, such as hardware router implementations, this is unacceptable [5].

One way to deal with both of these issues is to use a queue to insert elements, which deals with these two issues in the following ways:

- As mentioned (1), some elements may need $\Omega(\log n)$ time to be inserted into the data structure. By using a queue as an intermediate buffer, one can first insert the elements into it and then take out elements one at a time and insert them into the cuckoo tables. As such, one can load-balance the operations of more costly insertions among less expensive ones. And since on average, the insertion of an element only requires a constant amount of steps, one only needs to do a constant amount of steps in each insertion call, leading to constant time insert operations.
- As mentioned (2), sometimes the cuckoo hash table may need to be re-randomized (rehashed). This happens, when even after the maximum number of iterations, there still has not been found an empty spot in one of the cuckoo tables. Using the queue, one can simply stash these more difficult elements there and continue trying to insert other elements. Since one uses a queue to ”stash” the elements away, they will be inserted back into the cuckoo tables after some time, which ensures that the queue will not contain too many elements.

Since, the queue is now part of the whole dynamic dictionary, it needs to support the operations $lookup$ and $delete$ in constant time, which a traditional queue does not. A such, a different implementation has to be used that supports all of these operations in constant time. In Section 4.1 a queue with these properties is presented. Furthermore, in Section 4.2 it is explained how problematic elements (elements that cause a rehash) can be detected efficiently. Additionally, in Section 4.3, it is presented how these auxiliary data structures are used to construct de-amortized cuckoo hashing and the operations of it are defined. Finally, in Section 4.4, it is proven that the auxiliary data structures (which are initialized to store $O(\log n)$ elements) do not overflow, which implies that the construction supports all operations in constant time.

4.1 A Queue with Constant-Time Operations

Since the aim is to implement a dynamic dictionary with constant time operations, all operations of auxiliary data structures must be constant time as well. This includes the queue that will be used to insert elements into the hash table. One approach to implement a queue with constant time support for the operations $pushBack$, $pushFront$, $popFront$, $lookup$, and $delete$ is presented in this section (in Section 4.3 it will be explained why exactly these operations are required). However, one should note that any other implementation with the same guarantees will also work. The implementation of the queue here is the same implementation as proposed in [4].

The queue uses a constant number k of arrays $A_1, \dots, A_k$, each of size n^δ, for some $\delta < 1$. Each element, stored in these arrays, has a previous and a next pointer, as well as its data element. Furthermore, two additional pointers are maintained: a head and a tail pointer. Each element x will be inserted into the first available entry among $\{A_1[h_1(x)], \dots, A_k[h_k(x)]\}$, where $h_1, \dots, h_k$ are hash functions of the domain $h_i : \mathcal{U} \to [n^\delta]$ randomly chosen from a pairwise-independent hash family. A conceptual view of the data structure can be seen in Figure 2.

The $lookup(x)$ operation examines all the possible positions $A_1[h_1(x)] \dots, A_k[h_k(x)]$ for the input element x, and if it is present in any, then, it returns $true$, and otherwise $false$. In the $push\_back(x)$ procedure, the first empty position of $A_1[h_1(x)] \dots, A_k[h_k(x)]$ is found for the input element x. Once, an empty position has been found, the element is placed there. Also, the new element is now the $tail$ of the queue and its $previous$ pointer points to the old $tail$ (and the $next$ pointer of the old tail is updated to the new $tail$). The $push\_front$ procedure works the same, except that the new element is the $head$ instead of the $tail$ and its $next$ pointer is adjusted instead of its $previous$ pointer. Furthermore, the $previous$ pointer of the old head is set to the new node. For the $pop\_front()$ operation, the $head$ element is returned if it is present. Furthermore, the $head$ pointer has to be adjusted to $head.next$ and the pointers of the $previous$ pointer of the head. Also, if the queue is now empty, $tail$ has to be reset. Finally, the $delete(x)$ procedure works similar to the $lookup(x)$ operation, in that it searches for the input element x in its possible positions $A_1[h_1(x)] \dots, A_k[h_k(x)]$. If it is found, then it is deleted and its $next$ and $previous$ pointers are adjusted. Also, if it was previously the $head$ or $tail$, then these pointers are also adjusted.

The operations are also defined in pseudocode in Algorithm 2.

Algorithm 2 Procedures of the Custom Queue Data Structure

```
1: function lookup($x$):
2:     for $i \in [k]$:
3:         if $A_i[h_i(x)] = x$ then return true;
4:     return false;

1: function push_back($x$):
2:     for $i \in [k]$:
3:         if $A_i[h_i(x)] = \perp$ then
4:             $A_i[h_i(x)].elem = x$;
5:             update previous pointer of $A_i[h_i(x)]$ and $tail$;
6:     announce failure

1: function push_front($x$):
2:     for $i \in [k]$:
3:         if $A_i[h_i(x)] = \perp$ then
4:             $A_i[h_i(x)].elem = x$;
5:             update next pointer of $A_i[h_i(x)]$ and $head$;
6:     announce failure

1: function pop_front():
2:     if $head = \perp$ then return $\perp$;
3:     $item \leftarrow head.elem$;
4:     update $head$ pointer and $tail$ if the queue is empty now;
5:     return item;

1: function delete($x$):
2:     for $i \in [k]$:
3:         if $A_i[h_i(x)] = x$ then
4:             $A_i[h_i(x)].elem = \perp$;
5:             update $next$ pointer of $A_i[h_i(x)].prev$ and $prev$ pointer of $A_i[h_i(x)].next$;
6:             update $head$ and $tail$ pointer if they pointed to the removed node;
7:             return
```

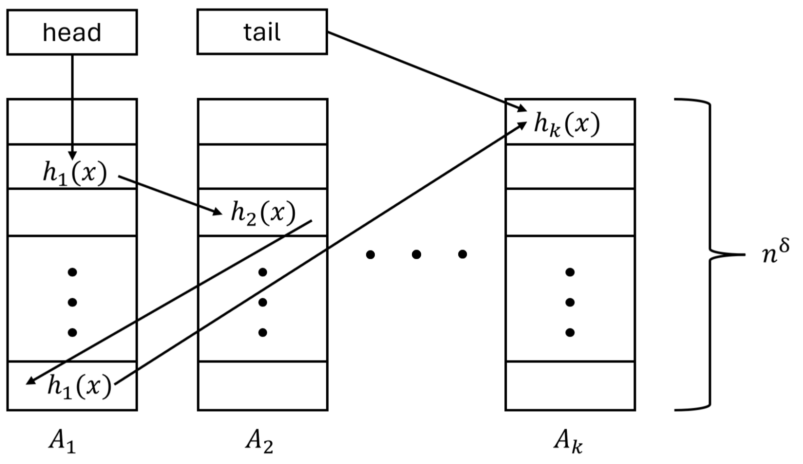


Figure 2: Conceptual View of the Custom Queue Data Structure

Lemma 4.1. *When the queue contains at most m elements, then for any element x, the probability that all of its possible entries $\{A_1[h_1(x)], \ldots, A_k[h_k(x)]\}$ are occupied is upper bounded by $(m/n^\delta)^k$.*

Proof. The probability that any specific entry in one of the A_i arrays is occupied is upper-bounded by m/n^δ, since h_i is a uniform hash function:

$$Pr[\text{Entry } j \text{ is occupied in } A_i] \leq m \cdot Pr[h(x) = j \text{ for any } x \in \mathcal{U}] = \frac{m}{n^\delta}$$

Furthermore, since the h_i are chosen independently at random, the probability that all k possible entries for an element $x \in \mathcal{U}$ are occupied is upper-bounded by $(m/n^\delta)^k$. □

Note. If $m \in o(n^\delta)$, such as $m = O(\log n)$, then this means that the probability of failure can be made as small as n^{-c} for any $c > 0$ by choosing an appropriate constant k. Notice that this is the probability of failure for the queue.

4.2 Cycle Detection Mechanism

4.2.1 Identifying Problematic Elements

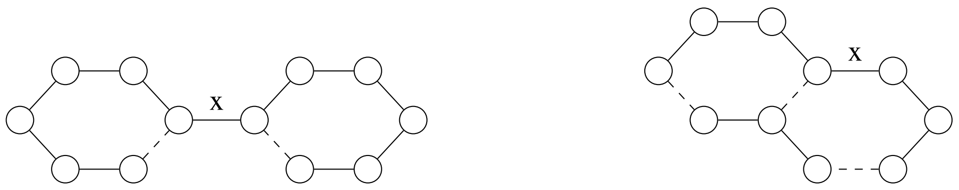
Using the cuckoo graph definition (see Definition 3.1) and according to Lemma 3.1, an element that creates a second cycle in an already connected component of the cuckoo graph will cause a failure. This means that the current elements cannot be stored using the hash functions h_1, h_2 and as such new ones will have to be chosen and elements inserted again (rehashing).

Definition 4.1. *An element x is considered problematic for a cuckoo hash table, with the hash functions h_1, h_2 and that stores the set S, if x creates a second cycle in a connected component of the cuckoo graph of S.*

Note. As such, using Lemma 3.1, if an element x is problematic, then it is impossible to store $S \cup x$ in a cuckoo hash table using the hash functions h_1, h_2.

Of course, this is not the only way that a rehash can happen in the original cuckoo hashing algorithm (see Algorithm 1). There, it may also happen, if the size of the connected component is greater than $MaxLoop$. However, since the insertion procedure is now de-amortized, this case is irrelevant.

The question that remains now is, how can one detect problematic elements efficiently. For this, it helps again to consider the cuckoo graph. In Figure 3, one can see how the insertion procedure might look like for a problematic element in the cuckoo graph. From Lemma 3.1, it is known that the element is inserted into a connected component and creates a second cycle in it.



(a) Case 1 when the two Cycles are Disjoint (b) Case 2 when the two Cycles aren't Disjoint

Figure 3: Insertion Procedure of a Problematic Element x in the Cuckoo Graph

When considering Figure 3, one can observe how the insertion procedure kicks out elements from their nests. In both cases, the following happens: First x is moved to one of its available entries. Second, the loop takes place until it gets back x's current position. Then, x is moved into its other position. It then loops back again through the other cycle and the moves are repeated. This observation helps us to identify problematic elements in the insertion procedure.

Observation 4.1. *The input element x of the insertion procedure is problematic, if during the insertion procedure, at least two nodes in the cuckoo graph are visited at least twice, or one node is visited at least three times.*

Note. In [4], instead of keeping track of the nodes itself, they keep track of the elements. While this is equivalent, keeping track of nodes (using IDs) will require less memory, since $n \leq |\mathcal{U}|$.

4.2.2 A Data Structure to Identify Problematic Elements Efficiently

Using Observation 4.1, one can now construct a data structure that identifies problematic elements. The idea is to store the nodes indices of the cuckoo graph in a set. This way, one can easily model once at least two nodes have been visited twice or one node visited thrice by using a boolean flag in addition (simply set the boolean flag once an element has been inserted that is already present and if it happens again with the flag set, then it is clear that the condition is met).

As mentioned in 4.1, all auxiliary data structures must only use constant time operations. Most, implementations of sets do not support the operations $insert, lookup, reset$ in constant time, hence a custom implementation is proposed here that has these properties. As proposed in [4], one can modify the queue data structure presented in Section 4.1 to support these operations in constant time. We now make this construction explicit. Inspiration for this modification was taken from [6].

As in Section 4.1, one again uses a constant number k of arrays $A_1, \ldots, A_k$, each of size n^δ, for some $\delta < 1$. Each element x will be inserted into the first available entry among $\{A_1[h_1(x)], \ldots, A_k[h_k(x)]\}$, where $h_1, \ldots, h_k$ are hash functions of the domain $h_i : \mathcal{U} \to [n^\delta]$ chosen from a pairwise-independent hash family. This time, however, each element will simply store an index $index$ of another array B. Additionally, an array B of size m is maintained that for each element stores a $points\_to$ to a position in $A_1, \ldots, A_k$ and a data element $elem$. Furthermore, a single integer $members$ is stored that counts how many elements are currently in the set. A conceptual view of the data structure can be seen in Figure 4.

The $lookup(x)$ operation examines all the possible positions $A_1[h_1(x)] \ldots, A_k[h_k(x)]$ for the input element x. If a $A_i[h_i(x)]$ is found that points forth to a position b in B that also points back to $A_i[h_i(x)]$. And b contains x and is valid according to the number of elements in the set (using the index of b in B), then the element is present and $true$ is returned. If no such position is found, then $false$ is returned. In the $insert(x)$ procedure, the first empty spot $A_i[h_i(x)]$ among $A_1[h_1(x)] \ldots, A_k[h_k(x)]$ is found for the input element x. Once it is found, a new entry b is created at $B[members]$ which stores the element and points to $A_i[h_i(x)]$. $A_i[h_i(x)]$ is also set to point to b, and finally the number of elements, represented by $members$, is incremented. If no empty position among $A_1[h_1(x)] \ldots, A_k[h_k(x)]$ is found, then failure is announced. The $reset$ operation simply sets the size of the set, which is denoted by $members$ to 0.

The procedures are also defined in pseudocode in Algorithm 3. Note, that the boolean flag to detect duplicate elements, for the actual cycle detection, is not included here.

Algorithm 3 Procedures of the Cycle Detection Mechanism

```
1: function lookup($x$):
2:     for $i \in [k]$:
3:         $a \leftarrow A_i[h_i(x)]$;
4:         $b \leftarrow B[a.index]$;
5:         if $b.points\_to = a$ and $b.elem = x$ and $a.index < members$ then return true;
6:     return false;

1: function insert($x$):
2:     for $i \in [k]$:
3:         $a \leftarrow A_i[h_i(x)]$;
4:         $b \leftarrow B[a.index]$;
5:         if $b.points\_to \neq a \vee a.index \geq members$ then
6:             $A_i[h_i(x)].index \leftarrow members$;
7:             $B[members].points\_to \leftarrow A_i[h_i(x)]$; $B[members].elem \leftarrow x$;
8:             $members \leftarrow members + 1$;
9:             return
10:    announce failure

1: function reset():
2:     $members \leftarrow 0$;
```

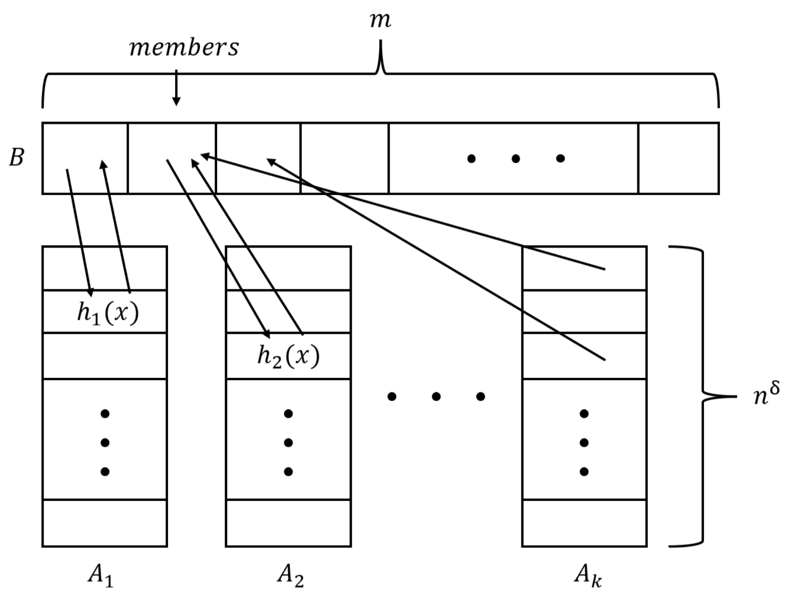


Figure 4: Conceptual View of the Cycle Detection Mechanism

All operations only depend on the constant k and as such they are all constant time. Correctness of the data structure can be reduced to verifying that the *lookup* procedure works correctly, since it is the only function that returns a value. The data structure may fail if for an element x, all of its possible entries $A_1[h_1(x)], \dots, A_k[h_k(x)]$ are occupied. Notice that this is the same failure condition as for the queue, discussed in Section 4.1 and as such it is upper bounded by $(m/n^{\delta})^k$ according to Lemma 4.1.

It should be easy to see that if no *reset* calls are made, then $lookup(x)$ will return true if and only if $insert(x)$ was executed before. In the case that $reset$ was called, there may still exist an entry a (or multiple) in one of the A_i's pointing to one of the positions b in B. However, then either $b \geq members$ or the pointer in b does not point back to a.

4.3 The Construction

As explained in Section 4, a queue is used to insert new elements $x \in \mathcal{U}$ into the data structure that is proposed in [4]. In a high-level way, one performs the normal cuckoo insertion procedure, however instead of only doing it for the insertion element x, one does L ”moves” instead (putting an element into one of its positions in the cuckoo tables and kicking the old one out if it is present). This means that multiple elements might be moved from the queue into the cuckoo tables in one insertion call. In Figure 5, one can see a conceptual overview of the construction. Elements are first inserted into the back of the queue and then fetched from the head. Elements fetched are then inserted into T_0 and T_1 using the normal cuckoo insertion procedure and if insertion is not complete after L total ”moves”, then the element is placed at the head of the queue again. Furthermore, problematic elements (as defined in Section 4.2.1) are detected using the cycle detection mechanism and placed into the back of the queue (i.e. they are stashed).

The $delete$ and $lookup$ procedures are naturally defined by the property that an element x is stored in exactly one of the following $T_0[h_0(x)]$, $T_1[h_1(x)]$, or the queue. This means that each of the possible positions is examined for the element and, in case of the $delete$ procedure, the element is removed.

It is shown that if the constant L is chosen appropriately, then with overwhelming probability, the queue will not contain more than $O(\log n)$ elements at any point in time (Lemma 4.2). And in that case, the construction provided in Section 4.1 will fail with negligible probability if k is chosen appropriately. Also, it will support constant time operations in that case. Furthermore, the cycle detection mechanism is initialized to store $O(\log n)$ elements.

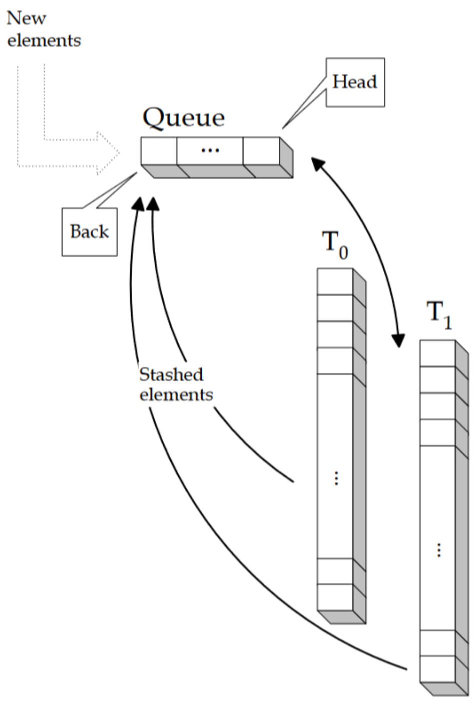


Figure 5: Conceptual View of the De-Amortized Cuckoo Hashing Construction [4]

The operations of the construction are also defined in pseudocode in Algorithm 4. The operations of the $queue$ are defined in Section 4.1 and the operations of the cycle detection mechanism

cdm are defined in Section 4.2.2.

Algorithm 4 Procedures of the De-Amortized Cuckoo Hashing Construction [4]

```
1: function lookup($x$):
2:     return $T_0[h_0(x)] = x \vee T_1[h_1(x)] = x \vee queue.lookup(x)$;

1: function delete($x$):
2:     if $T_0[h_0(x)] = x$ then
3:         $T_0[h_0(x)] = \perp$; return
4:     if $T_1[h_1(x)] = x$ then
5:         $T_1[h_1(x)] = \perp$; return
6:     $queue.delete(x)$

1: function insert($x$):
2:     $queue.push\_back(x, 0)$;
3:     $y \leftarrow \perp$;
4:     loop $L$ times:
5:         if $y = \perp$ then                        ▷ Fetching an element from the queue
6:             if $queue.empty()$ then
7:                 return
8:             else
9:                 $(y, b) \leftarrow queue.pop\_front()$;
10:        if $T_b[h_b(y)] = \perp$ then                            ▷ Successful insert
11:            $T_b[h_b(y)] \leftarrow y$;
12:            $cdm.reset()$;
13:            $y \leftarrow \perp$;
14:        else                      ▷ Entry occupied, need to move elements in the cuckoo table
15:            if $cdm.lookup(y, b)$ then                   ▷ Found a problematic element
16:                $queue.push\_back(y, b)$;                               ▷ Stash it for now
17:                $cdm.reset()$;
18:                $y \leftarrow \perp$;
19:            else                                                  ▷ Normal cuckoo move
20:                $z \leftarrow T_b[h_b(y)]$;
21:                $T_b[h_b(y)] \leftarrow y$;
22:                $cdm.insert(y, b)$;
23:                $y \leftarrow z$;
24:                $b \leftarrow b - 1$;
25:    if $y \neq \perp$ then
26:        $queue.push\_front(y)$;        ▷ Insertion incomplete, continue on next insert call
```

As mentioned in Section 4.2.1, instead of inserting the elements of the universe $\mathcal{U}$ into the cycle detection mechanism (which are of size $\log |\mathcal{U}|$ bits), one can also insert the nodes indices $(h_0(x), 0)$ or $(h_1(x), 1)$ of an element x. This may prove beneficial, since this would only require $\log n + 1$ bits, where n are the amount of entries in T_0 and T_1 (storing the elements directly would require $\log \mathcal{U}$ bits per element).

4.4 Performance Analysis

Since the aim of this report is to give an intuitive understanding of backyard cuckoo hashing, the performance analysis for the construction of de-amortized cuckoo hashing is only viewed on a high level. For a more detailed analysis, we refer the reader to [4].

The memory utilization of the construction is essentially 50% as in the standard cuckoo hashing. Each of the tables T_0 and T_1 has $(1 + \epsilon)n$ entries, and the auxiliary data structures are constructed to only contain $O(\log n)$ elements and require sublinear space. It is shown that if the auxiliary data structures do not overflow or fail, then all operations are performed in constant time. As stated in Lemma 4.1, by choosing an appropriate constant k, the probability of failure for each of the auxiliary data structures can be upper-bounded by n^{-c} and as such, it only has to be ensured that they do not overflow. In Theorem 4.1, it shown that auxiliary data structures do not overflow with high probability.

Definition 4.2. *[4] A sequence π of $insert$, $delete$, and $lookup$ operations is n-bounded if at any point in time during the execution of π the data structure contains at most n elements.*

Lemma 4.2. *For any polynomial $p(n)$ and any constant $\epsilon > 0$, there exists a constant L such that when instantiating the data structure with parameters ϵ and L the following holds: For any n-bounded sequence of operations π of length at most $p(n)$, with probability $1 - 1/p(n)$ over the choice of hash functions from their respective families, the queue contains $O(\log n)$ elements.*

Proof Sketch. As when analyzing cuckoo hashing (Section 3), it is again important to look at the size of the connected components in the cuckoo graph. The proof in [4] works by grouping insertions to be of size $\log n$ and then analyzing these. Particularly, it is shown that the sum of the sizes of the components of one of these groupings (which is of size $\log n$) is only of size $O(\log n)$. And since the insertion time is bounded by the size of the component of an element, this means that the insertion time for this grouping is $O(\log n)$. As such, L can be chosen so that the elements will be inserted during $\log n$ insertion calls. Of course, it could happen that the elements that take longer to insert are at the end of an insertion sequence of a grouping i_1 and then "block" the elements (in the queue) from the next grouping i_2. However, as said, each grouping can be inserted within $O(\log n)$ moves with high probability. As such, within $\log n$ insertions (starting from the last insertion of i_1), all elements of i_1 will be inserted and as such, only $O(\log n)$ elements are in the queue, which will all be inserted again in $O(\log n)$ insertions with high probability.

Furthermore, the probability that a component contains multiple cycles (and as such elements of it cannot be inserted into the cuckoo tables) is low for a load factor of $1/2$ or lower and as such, few elements have to be stashed permanently.

Lemma 4.3. *For any polynomial $p(n)$ and any constant $\epsilon > 0$, there exists a constant L such that when instantiating the data structure with parameters ϵ and L the following holds: For any n-bounded sequence of operations π of length at most $p(n)$, with probability $1 - 1/p(n)$ over the choice of hash functions from their respective families, the cycle detection mechanism contains $O(\log n)$ elements.*

Proof Sketch. The cycle detection mechanism only contains at most as many elements as the size of the maximal component (the component which has the most amount of elements). This size is expected to be around $O(\log n)$ as well, which means that it will only contain $O(\log n)$ elements.

Theorem 4.1. *[4] For any polynomial $p(n)$ and any constant $\epsilon > 0$, there exists a constant L such that when instantiating the data structure with parameters ϵ and L the following holds: For any n-bounded sequence of operations π of length at most $p(n)$, with probability $1 - 1/p(n)$ over the choice of hash functions from their respective families, the auxiliary data structures do not overflow during the execution of π.*

Proof. This follows immediately from Lemma 4.2 and 4.3 since the data structures are initialized to store $O(\log n)$ elements. □

5 Backyard Cuckoo Hashing

The de-amortized cuckoo hashing construction presented in Section 4 improves on the problems of traditional cuckoo hashing. However, it still only has a memory utilization of about 50%. In this section, the backyard cuckoo hashing construction [1] is presented that requires only $(1+\epsilon)n$ memory words to store n elements, while keeping the guarantee of constant-time operations (however, depending on ϵ) with a high probability. The construction is presented in Section 5.1 (with an explanation of the choice of parameters in Section 5.2). Furthermore, in Section 5.3 the performance of the data structure is examined.[1]

Additionally, in Section 5.4 an augmentation for the scheme is presented that, for any $\epsilon = \Omega((\log\log n/\log n)^{1/2})$, eliminates the dependency on ϵ of the operations by using a de-amortized perfect hashing scheme. This holds for any sequence of polynomially many operations, with high probability over the initialization phase. [1]

Finally, in Section 5.5 another augmentation is presented, which reduces the memory utilization of the construction to $(1+o(1))\log_2\binom{u}{n}$ bits, where u is the size of the universe and n is the number of elements stored. This is nearly optimal, as $\log_2\binom{u}{n}$ is the information-theoretic space bound to represent a set of size n taken from a universe of size u. [1]

5.1 The Construction

The construction is split into two levels, the first level is a table T_0 that contains m bins. Each of these bins contain d memory words and can be thought of as an array. As such, the first level can accommodate up to $m{\cdot}d$ elements. Additionally, T_0 is equipped with a hash function $h_0 : \mathcal{U} \to [m]$ that is sampled from a collection of k-wise independent hash functions. The second level of the construction (also referred to as the backyard) is of the same format as the de-amortized cuckoo hashing construction presented in Section 4 (although the operations are defined differently and the tables are now named T_1, T_2).

On a high-level, the operations of the data structure are defined such that it tries to only use the backyard as little as possible, and it is initialized to store only a small fraction of elements (specifically only $\epsilon n/16$ elements). To make sure, that the backyard remains small even in the case of deletions, elements from the second level, are moved back to the first level whenever possible. This can be done efficiently, since the backyard uses cuckoo hashing and as such, each time an element is moved in one of the cuckoo tables, one can also try to insert it back into the first level. [1]

A conceptual view of the construction can be seen in Figure 6. The bins of the first level store the majority of elements, while the backyard mainly acts as a buffer for overflowing elements.

The data structure supports three operations, $lookup$, $insert$, and $delete$. These are defined similar to the operations of the de-amortized cuckoo hashing construction presented in Section 4.3. The main difference is that elements may now also be in the first level and that in the $insert$ procedure, one tries to move elements from the second level to the first level, before moving them around in the second level.

The $lookup(x)$ and $remove(x)$ exploit the fact that an element x may reside either in its bin in the first level $T_0[h_0(x)]$, in one of the cuckoo tables in the positions $T_1[h_1(x)]$ or $T_2[h_2(x)]$, or in the queue.

For the $insert(x)$ operation, an element is first inserted at the back of the queue. Then, for L iterations, the following is repeated: First, an element y is fetched from the head of the queue, then it is placed if possible in its first level bin $T_0[h_0(y)]$. If that fails, then it is moved to one of the cuckoo tables, while keeping track which elements have been moved in the cycle detection mechanism. If the cycle detection mechanism detects a problematic element (defined in Section 4.2.1), then it is moved to the tail of the queue (i.e. it is stashed).

The operations of the construction are also defined in pseudocode in Algorithm 5.

Algorithm 5 Procedures of the Backyard Cuckoo Hashing Construction [1]

```
1: **function** lookup($x$):
2:     **return** $x$ is in bin $T_0[h_0(x)] \vee T_1[h_1(x)] = x \vee T_2[h_2(x)] = x \vee queue.lookup(x)$;

1: **function** delete($x$):
2:     **if** $x$ is stored in bin $T_0[h_0(x)]$ **then**
3:         remove $x$ from bin $T_0[h_0(x)]$; **return**
4:     **if** $T_0[h_0(x)] = x$ **then**
5:         $T_0[h_0(x)] = \bot$; **return**
6:     **if** $T_1[h_1(x)] = x$ **then**
7:         $T_1[h_1(x)] = \bot$; **return**
8:     $queue.delete(x)$

1: **function** insert($x$):
2:     $queue.push\_back(x, 1)$;
3:     $y \leftarrow \bot$;
4:     **loop** $L$ **times:**
5:         **if** $y = \bot$ **then**                                    ▷ Fetching an element from the queue
6:             **if** $queue.empty()$ **then**
7:                 **return**
8:             **else**
9:                 $(y, b) \leftarrow queue.pop\_front()$;
10:        **if** there is a vacant place in bin $T_0[h_0(x)]$ **then**           ▷ Move $y$ back to first level
11:            Store $y$ in bin $T_0[h_0(y)]$;
12:            $y \leftarrow \bot$;
13:        **else**
14:            **if** $T_b[h_b(y)] = \bot$ **then**                                        ▷ Successful insert
15:                $T_b[h_b(y)] \leftarrow y$;
16:                $cdm.reset()$;
17:                $y \leftarrow \bot$;
18:            **else**                ▷ Entry occupied, need to move elements in the cuckoo table
19:                **if** $cdm.lookup(y, b)$ **then**                        ▷ Found a problematic element
20:                    $queue.push\_back(y, b)$;                                    ▷ Stash it for now
21:                    $cdm.reset()$;
22:                    $y \leftarrow \bot$;
23:                **else**                                                       ▷ normal cuckoo move
24:                    $z \leftarrow T_b[h_b(y)]$;
25:                    $T_b[h_b(y)] \leftarrow y$;
26:                    $cdm.insert(y, b)$;
27:                    $y \leftarrow z$;
28:                    $b \leftarrow 3 - b$;
29:        **if** $y \neq \bot$ **then**
30:            $queue.push\_front(y, b)$;          ▷ Insertion incomplete, continue on next $insert$ call
```

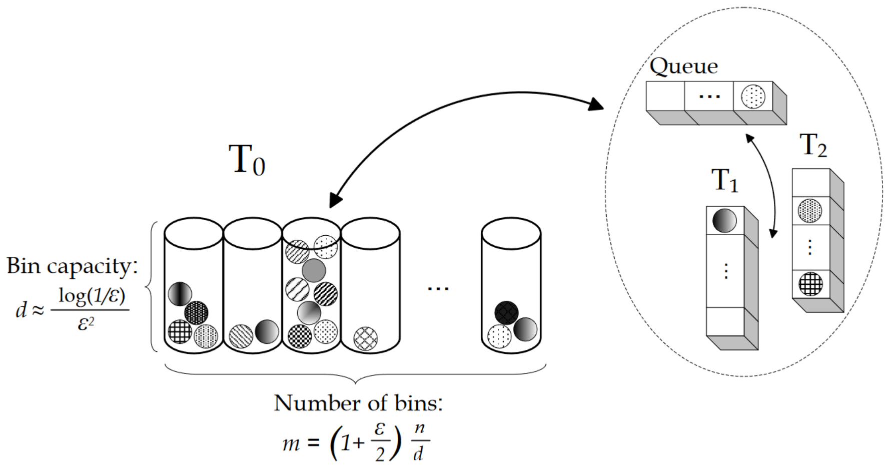

Queue
T$_2$
T$_1$
T$_0$
Bin capacity:
$d \approx \frac{\log(1/\varepsilon)}{\varepsilon^2}$
Number of bins:
$m = \left(1+\frac{\varepsilon}{2}\right)\frac{n}{d}$

Figure 6: Conceptual View of the Backyard Cuckoo Hashing Construction [1]

5.2 Choosing Parameters

In this section, we will try to give an intuition on why the parameters, specifically the number of bins and the size of each bin, of the data structure are chosen the way they are. One of the downsides of the de-amortized cuckoo hashing construction presented in Section 4 is that its memory utilization is only about 50%. Specifically, the goal for this construction was to only use $(1 + o(1))n$ memory words. As such, the number of bins m times the size of the bins d has to be $(1 + o(1))n$.

The number of bins is chosen to be $m = \lceil (1 + \epsilon/2)n/d \rceil$ and as such, as long as $\epsilon \in o(1)$, the goal of only using $(1 + o(1))n$ memory words is achieved.

For the size of the bins, it is more complicated. The problem is that there is a tradeoff between having big bins and small bins, and as such something in between has to be chosen. If one has very big bins, then one must still be able to find elements in them in constant time. If the bins would for example be of size $\Theta(n)$, then one would need another dynamic dictionary inside each of these bins, and we would not have solved our problem. However, if one chooses very small bins, then it is likely that many of them overflow and as such the backyard would have to be initialized to store many elements. As such, it would not be possible to achieve the goal of only using $(1 + o(1))n$ memory words, because the backyard would need more space then. We delve deeper into this issue later in the experiment presented in Section 7.1.

As such, the size of the bins has to be chosen not to be too small and not to be too big. d is roughly $\log n/ \log\log n$, for $\epsilon \in \Theta((\log\log n/ \log n)^{1/2})$, which is a very slowly growing function and suitable for this application.

5.3 Performance Analysis

Lemma 5.1. *After n insertions, the backyard contains at most $\epsilon n/16$ elements with high probability.*

Proof Sketch. In [1] they prove this by grouping together bins into blocks and show that these blocks all get a number of elements that is close to the expectation of what they should get. They then show that the number of non-overflowing elements (of the first level) of a block B_i is $(1 - \epsilon/16)$ times the number of elements that were assigned to that block. This happens with high probability for the choice of $d = O(\log(1/e)/e^2)$. As such, the number of overflowing elements is less than $\epsilon n/16$ for all bins with high probability.

Theorem 5.1. *[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 choice of hash functions from their families, all insertions are performed in time $O(d) \sim O(\log(1/\epsilon)/\epsilon^2)$ in the worst case. Deletions and lookups are always performed in time $O(d) \sim O(log(1/\epsilon)/\epsilon^2)$ in the worst case.*

Proof Sketch. First, consider the memory used by the first level of the construction. $m = \lceil(1+\epsilon/2)n/d\rceil$ implies that the first level uses $\approx (1+\epsilon/2)n$ memory words. The second level is initialized to store at most $\epsilon n/16$ elements and in Section 4.4, it was shown that it has a memory utilization of about 50%. It suffices to assume, that it uses at most $\epsilon n/4$ memory words. As such, the total number of memory words for the construction is $md + \epsilon n/4 \leq (1+\epsilon)n$.

Second, consider the insertion time of an element. If an element will be placed into the first level, then this will take time linear to the size of the bin, as such $O(d)$. If an element is placed into the second layer, then this will happen in constant time, according to the analysis for the de-amortized cuckoo hashing construction in Section 4.4. This is only true, however, if the backyard does not overflow, but according to Lemma 5.1 this will not happen with high probability, since the backyard is initialized to store $\epsilon n/16$ elements. As such, elements are inserted in time $O(d)$ in the worst-case with high probability (since there is a chance that the backyard might overflow). Deletions and lookups can always be done in time $O(d)$, since these operations will never overflow the backyard.

5.4 Perfect Hashing for Bins

The construction can be augmented by using a perfect hashing scheme to eliminate the dependency on ϵ. A perfect hashing scheme maps elements from its domain to its range without collisions, as compared to most hash functions, which usually have collisions. Upon storing an element in one of the bins of the first level, the perfect hash function is reconstructed to accommodate the element. In the worst case, this will still take time linear in the size of the bins, $O(d) = O(\log(1/\epsilon)/\epsilon^2)$, which is dependent on ϵ. However, using an augmentation, that de-amortizes the perfect hashing scheme, can reduce this to constant worse case time that is independent of ϵ, for any $\epsilon \in \Omega((\log\log n/\log n)^{1/2})$. Here, the specific perfect hashing scheme forces this restriction, as such another one which satisfies the same properties may allow more freedom for the choice of ϵ. For the perfect hashing scheme, it is required that for any sequence of insertions and deletions, that lead to a set of size at most $d-1$, for any element $x \notin S$, that is currently being inserted, it holds that:

Property 1: [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: [1] With probability $O(1/d)$ rehashing is required, and the rehashing time takes $O(d)$ in expectation.

These properties are required to ensure that the perfect hashing scheme can successfully be de-amortized to require only constant worst case time for insertions, by using the queue.

The approach relies on the fact that the same scheme is applied across all bins and uses the queue from the backyard to de-amortize the procedure (similar to how the insertion procedure was de-amortized in Section 4.3). As such, the number of operations it takes for the bin to be rehashed is included in the L steps of the insertion procedure. To still be able to support operations for the bin that is currently rehashed, it will be copied to a dedicated memory address, where the rehash is taking place, so the original bin is still usable. Note, that only one bin may be rehashed a t time, because of the queue. [1]

5.4.1 The Construction

The perfect hashing scheme works by using two functions $h : \mathcal{U} \to [d^2]$ and $g : [d^2] \to [d]$ for mapping any element $x \in \mathcal{U}$ to an index of the bin $i \in [d]$. h is drawn from a family of pairwise-independent hash functions and can be evaluated in constant time and only uses a constant amount of memory. The second function g can be thought of as a lookup table. In [1], they store the description of g as at most d pairs taken from $[d^2] \times [d]$, requiring at most $d(\log d^2 + \log d) = 3d \log d$ bits. However, this can be reduced to only using $2d \log d$ bits, by defining the order of the d numbers implicitly. This means that instead of representing the function in the following way:

$$(z_1 \in [d^2], y_1 \in [d]), \ldots, (z_d \in [d^2], y_d \in [d])$$

Since it is known that for any pair y_i, y_j they will be of a different value and since there are d, their union will form the set $[d]$. As such, we can order them, according to the value of y_k (which makes y_k implicit and as such, we can remove it).

$$(z_1 \in [d^2], 0), \ldots, (z_d \in [d^2], d-1)$$

However, it is not clear that it is always possible to evaluate g in worst-case constant time. Since the same scheme is applied in all the bins, one can afford to use a lookup table that takes the description of the function (the z_k's) and the input value $x \in [d^2]$ and computes the output value $g(x) \in [d]$. There are $2^{2d \log d + 2 \log d}$ combinations of descriptions and input values, since the description of g is $2d \log d$ bits long and the description of an input value $x \in [d^2]$ is $\log d^2 = 2 \log d$ bits long. As such, for each of these combinations, the output value $g(x) \in [d]$ has to be stored, which will take $\log d$ bits. The total memory utilization of the lookup function for g (which can be used across all bins) is $2^{2d \log d + 2 \log d} \cdot \log d$ bits. To implement such a lookup table, one can take the bit-presentation of the input values, concatenate them, interpret them as a number, which is then the index to an array, where each entry uses $\log d$ bits.

Furthermore, when inserting and deleting a new element, g has to be updated to accommodate the new element. For this, another lookup table is used, similar to the one for evaluation. This lookup table takes as input the old description of g and an element to add/delete (depending on if it is present) $r \in [d^2]$. As such, there are $2^{2d \log d + 2 \log d}$ combinations of input values to the function. The new description of g also takes $2d \log d$ space and as such, the total memory utilization for this lookup function is $2^{2d \log d + 2 \log d} \cdot 2d \log d$.

This perfect hashing scheme has to rehash only if there is a collision in h for a set of elements $S \subseteq U$. To simplify the proof of properties 1 and 2, "forced rehashing" is introduced, which always rehashes after $\nu \in [d]$ updates (unless rehashing was done earlier due to a collision in h). ν is chosen randomly after every rehash for every bin. Note, that this does not hurt properties 1 or 2.

Lemma 5.2. *[1] Let $1 \leq \mu < d$, fix a sequence σ of operations leading to Set S of size μ, and assume that S does not have any collisions under the currently chosen function $h : \mathcal{U} \to [d^2]$. Then, for any sequences of memory configurations and rehashing times that occurred during the execution of σ, and for any element $x \notin S$, the probability over the choice of h that x will form a collision with an element of S is $O(1/d)$.*

Proof. First, assume that there are only insertions and no deletions (see Note 5.4.1, for why deletions are not allowed at first). Then the current hash function h is uniformly distributed in the collection of pairwise independent functions, subject to not having any collisions in the set S. Therefore, for any element $x \in S$ it holds that.

$$\begin{aligned}
&Pr[x \text{ collides with an element of } S \mid h \text{ has no collisions on S}] \\
&= \frac{Pr[x \text{ collides with an element of } S \wedge h \text{ has no collisions on S}]}{Pr[h \text{ has no collisions on S}]} \\
&\leq \frac{Pr[x \text{ collides with an element of } S]}{Pr[h \text{ has no collisions on S}]}
\end{aligned}$$

Since h is sampled from a family of pairwise-independent hash functions, the following holds:

$$Pr[x \text{ collides with an element of } S] \leq \frac{|S|}{d^2} \leq \frac{d}{d^2} = \frac{1}{d}$$

To compute the probability that there are no collisions of h on S, we can instead compute the inverse probability. For this, we have to consider that at least one collision happens. There are $|S|$ elements and for each pair (of which there are $|S|(|S|-1)/2$) a collision will happen with probability $1/d^2$ (since h is pairwise-independent).

$$\begin{aligned}
Pr[h \text{ has no collisions on S}] &= 1 - Pr[h \text{ has at least one collision on } S] \\
&= 1 - \frac{|S|(|S|-1)}{2d^2} \geq 1 - \frac{d(d-1)}{2d^2} \geq \frac{1}{2}
\end{aligned}$$

As such.

$$Pr[x \text{ collides with an element of } S \mid h \text{ has no collisions on S}] \leq \frac{2}{d} \in O(1/d)$$

The analysis above was done under the assumption that there are no deletions. If that is not the case, however, then the current hash function is not uniformly distributed anymore (subject to not having any collisions in S). However, since we always rehash after at most d updates, then even if we include the "deleted" elements since the last rehash, then we are left with a set of size at most $3d/2$. And for that, the same analysis as above holds. [1] □

Note. If deletions were allowed in the beginning of the proof of Lemma 5.2, then this would change the way the probability is calculated that h has no collisions on S. Since, then h might have to be 1-1 on different sets of the same size (e.g. for the set $\{10, 20, 30\}$ and for the set $\{10, 30, 40\}$).

Using the perfect hashing scheme, we can update Theorem 5.1 to the following:

Theorem 5.2. *[1] For any n there exists a dynamic dictionary with the following properties:*

1. *The dictionary stores n elements using $(1+\epsilon)n$ memory words, for $\epsilon \in \Theta((\log\log n/\log n)^{1/2})$.*
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 choice of hash functions from their families, all operations are performed in constant time, independent of ϵ, in the worst case.*

5.5 Matching the Information-Theoretic Space Bound

The information-theoretic space bound for a representing a set of size n taken from a universe $\mathcal{U}$ of size u denotes the minimum amount of required information to represent such a set uniquely. There are $\binom{u}{n}$ distinct subsets of $\mathcal{U}$ of size n, so if one assigns a unique label (a number) to all of them, one could distinguish which subset is currently stored. Representing the numbers $\{0, 1, \ldots, \binom{u}{n}\}$ requires at least $\lceil \log_2 \binom{u}{n} \rceil$ bits, which is the information-theoretic space bound.

In [1], it is explained how the backyard cuckoo construction can be augmented to achieve the information-theoretic space bound. However, only the general idea is presented here. For the interested reader, we refer to [1].

The idea is to split each element x into two parts x_L, the quotient, and x_R, the remainder, as shown in Figure 7. x_L are the left-most $\log m$ bits of x and x_R are the right-most $\log(u/m)$ bits of x. Remember, that m was the number of bins used in the first level. Since x_L is exactly $\log m$ bits, one can interpret it as a number and use it as an index to a bin. Then, only x_R is stored inside the bin, instead of the whole bit vector of x (note, that while x_R might not be unique across all bins then, it is unique inside of bin x_L). Note, that inside of the backyard, elements are still stored in the traditional way.

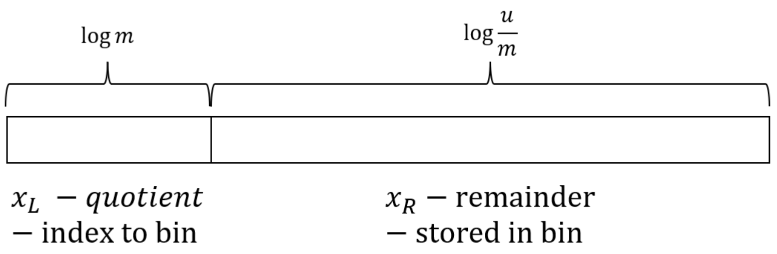


Figure 7: Separation of x into a quotient x_L and remainder x_R

In general, it cannot be assumed that the input to a dictionary is fully random. As such, it might happen that a lot of elements start with the same bits in the quotient x_L. If that is the case, then the associated bin will likely overflow quicker than expected (according to the analysis in Section 5.3). However, one can avoid this dilemma by using a permutation function $\pi : \mathcal{U} \to \mathcal{U}$, which will act as a randomizer for the input. Instead of using x_L for the index of the bin, one uses $\pi_L(x)$ (the left-most $\log m$ bits of $\pi(x)$). Similarly, one now stores $\pi_L(x)$ instead of x_L. Note that this approach only works, since π is bijective.

6 Implementation Details, Challenges, and Simplifications

In this section, implementation aspects of the backyard construction are presented. In Section 6.1 the chosen hash functions are discussed and the challenges that come with them. And in Section 6.2 we present on how we deal with the failure of data structures in our implementation.

6.1 Hash Functions

In [1], hash functions that have different properties are used in the proposed data structures. The queue presented in Section 4.1 uses a 2-wise independent hash function, as well as our concretization of the cycle detection mechanism (presented in Section 4.2). They do not propose a specific hash function to use here and as such we decided to use the Carter-Wegman hash function [7], since it uses a constant amount of space and has constant evaluation time. We provide a brief presentation of it in Section 6.1.1.

For the backyard construction (presented in Section 5), specifically, the tables T_0, T_1, T_2 a k-wise independent hash function is used to hash elements into their spots. In [1] it is proposed to use the construction of Siegel [8] or Dietzfelbinger [9]. However, our supervisor advised us to not implement these, as that is highly complicated. Instead, we use a form of tabulation hashing, since these have been shown to have good practical results for cuckoo hashing [10]. Simple tabulation hashing is presented in Section 6.1.2. Specifically, we use tornado tabulation hashing [11], which is an enhanced form of tabulation hashing. It is designed to break dependencies in the calculation that are usually found in tabulation hashing. It is presented in Section 6.1.3.

Finally, Section 6.1.4 and 6.1.5 discuss two challenges encountered when implementing hash functions, along with the strategies we use to address them.

6.1.1 Carter-Wegman Hashing

In [7] a 2-wise-independent hash family is proposed which contains functions of the form $h : \mathbb{Z} \to [m]$ with $m > 0$. The family is defined by picking a prime number $p \geq m$ and then defining it over the parameters $a \in \{1, \dots, p-1\}$ and $b \in [p]$ in the following way:

$$h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$$

The function only uses a constant amount of space and can be evaluated in constant time in the unit cost RAM model.

6.1.2 Simple Tabulation Hashing

Known constructions of fully random hash functions are not very practical. They generate and store a random output value for every input value. Assuming keys and values are 32-bit integers, this would require $2^{32} \cdot 32 \approx 1.28 \cdot 10^{11}$ bits. However, for small input spaces, this becomes feasible. Simple tabulation hashing uses this fact to generate fully random hash functions for partitions of its input values.

Keys (which are bit-strings) are split into c partitions $x = (x_1, \dots x_c)$ of equal length, where each $x_i \in \Sigma$. Then c fully random hash functions $H_1, \dots H_c$ are independently generated, which map each partition to the output domain $\mathcal{R} = 2^r$, for some $r > 0$. Finally, the outputs of the $H_i[x_i]$ are XORed. [10]

$$h(x) = H_1[x_1] \oplus H_2[x_2] \oplus \cdots \oplus H_c[x_c]$$

Lemma 6.1. *Simple Tabulation Hashing is not 4-independent.*

Proof. To show that h is not 4-independent, it suffices to show that there exist any four distinct elements $a, b, c, d \in \mathcal{U}$ and $a', b', c', d' \in \mathcal{R}$ for which

$$Pr[h(a) = a' \land h(b) = b' \land h(c) = c' \land h(d) = d'] > \frac{1}{|\mathcal{R}|^4}$$

We will first show, that this holds for $c = 2$ and then extend it to the general case. Assume that h is 4-independent. Note that this also implies that h is 3-independent. Now consider the elements $a = \alpha \cdot \beta$, $b = \alpha \cdot \alpha$, $c = \beta \cdot \beta$, $d = \beta \cdot \alpha$ (here $\cdot$ is the concatenation symbol for the bit-strings α, β). As such $a_1 = \alpha, a_2 = \beta$, and $h(a) = H_1(a) \oplus H_2(b)$ (the same applies to b, c, d).
Reminder that $s \oplus s = 0$ for any bit-string s and that the XOR operator is commutative, as such:

$$h(a) \oplus h(b) \oplus h(c) = H_1(\alpha) \oplus H_2(\beta) \oplus H_1(\alpha) \oplus H_2(\alpha) \oplus H_1(\beta) \oplus H_2(\beta) =$$

$$H_1(\alpha) \oplus H_1(\alpha) \oplus H_1(\beta) \oplus H_2(\beta) \oplus H_2(\beta) \oplus H_2(\alpha) = 0 \oplus H_1(\beta) \oplus 0 \oplus H_2(\alpha) = h(d)$$

As such, the values of a, b, c determine the value of d, which means that:

$$Pr[h(a) = a' \land h(b) = b' \land h(c) = c' \land h(d) = d' = h(a) \oplus h(b) \oplus h(c)] =$$

$$Pr[h(a) = a' \land h(b) = b' \land h(c) = c'] = \frac{1}{|\mathcal{R}|^3} > \frac{1}{|\mathcal{R}|^4}$$

Now for the general case where c is some positive integer, one can add a concatenation of Γ to all strings a, b, c, d, such that $a_1 = \alpha, a_2 = \beta, a_{3,\ldots,c} = \Gamma$ (the same applies for b, c, d). Notice that $h(a) \oplus h(b) \oplus h(c) = h(d)$ still holds and as such the same analysis as above holds. $\square$

While simple tabulation hashing is not even 4-independent, it "provides many of the guarantees that are normally obtained via higher independence [...]." [10] In [10] they also show that it is a suitable function for cuckoo hashing.

Theorem 6.1. *[10] Any set of n keys can be placed in two tables of size $m = (1 + \epsilon)$ by cuckoo hashing and simple tabulation with probability $1 - O(n^{-1/3})$. There exist sets on which the failure probability is $\Omega(n^{-1/3})$.*

Note. With truly random hashing, this bad event happens with probability $\Theta(1/n)$. [10]

6.1.3 Tornado Tabulation Hashing

Tornado tabulation hashing [11] extends simple tabulation hashing by adding additional derived "characters" to the input keys. These derived "characters" are generated by applying a hash function to all preceding characters (original and derived), which adds additional randomness. This helps disrupt patterns and dependencies in the input keys.

A tornado tabulation hash function h has $d \in O(c)$ derived characters. For each $i = 0, \ldots, d$, let $\tilde{h}(x) : \Sigma^{c+i-1} \to \Sigma$. For a key $x \in \Sigma^c$, define its derived key $\tilde{h}(x) \in \Sigma^{c+d}$ as $\tilde{x} = \tilde{x}_1 \ldots \tilde{x}_{c+d}$ where

$$\tilde{x}_i = \begin{cases} x_i & \text{if } i < c \\ x_c \oplus \tilde{h}_0(\tilde{x}_1 \ldots \tilde{x}_{c-1}) & \text{if } i = c \\ \tilde{h}_{i-c}(\tilde{x}_1 \ldots \tilde{x}_{i-1}) & \text{if } i > c \end{cases}$$

Finally, another simple tabulation hash function $\hat{h} : \Sigma^{c+d} \to \mathcal{R}$ is applied to the derived key to get the output $h(x) = \hat{h}(\tilde{x})$.

The C code for 32-bit keys with $\Sigma = [2^8], c = 4, d = 4$, and $\mathcal{R} = [2^{24}]$ is presented in Algorithm 6. There, the input parameter H is an array of $c + d$ tables of size Σ that are filled with independently drawn 64-bit values. From the implementation, it becomes apparent that only simple bit-operations and lookups are used. It only uses $O(c + d)$ operations.

Algorithm 6 Tornado Tabulation Hashing C Code [11]

```
INT32 Tornado(INT32 x, INT64[8][256] H) {
    INT32 i; INT64 h=0; INT8 c;
    for (i=0; i<3; i++) {
        c=x;
        x>>=8;
        h^=H[i][c];
    }
    h^=x;
    for (i=3; i<8; i++) {
        c=h;
        h>>=8;
        h^=H[i][c];
    }
    return ( (INT32) h );
}
```

Theorem 6.2. *[11] Let $h : \Sigma^c \to \mathcal{R}$ be a random tornado tabulation hash function with d derived characters. For any fixed $X \subseteq \Sigma^c$, if $|X| \leq |\Sigma|/2$, then h is fully random on X with probability at least*

$$1 - 7|X|^3(3/|\Sigma)^{d+1} - 1/2^{|\Sigma|/2}$$

This means that if only a small set of keys (i.e. $< |\Sigma|/2$ keys) is hashed, then with high probability h behaves fully random. For larger sets of keys (e.g Σ^3), the authors of [11] show that using tornado tabulation hashing makes a diverse set of algorithms behaves almost the same as when using a fully random hash function. For details about the properties of tornado tabulation hashing, we refer the interested reader to [11].

6.1.4 Sampling a Prime

The Carter-Wegman hash function introduced in Section 6.1.1 uses a prime number p to compute its output values, which is not uncommon for hash functions. This prime number should be randomly sampled. However, it is unclear if there are any benefits to uniformly sampling a prime number from all prime numbers in a range or if it is enough to use a pre-defined list of prime numbers and sample one of these. We decided to use the latter approach for its simplicity. It would be interesting to see however, if there are any noticeable benefits to using the former approach.

Uniformly sampling a prime number p in a range of values $[a, b]$ can be done in the following way. One can uniformly sample a random number $r \in [a, b]$ and then check if r is a prime number. If that is the case, then r is returned. Otherwise, one can again uniformly sample a random number $r' \in [a, b]$ and check if it is a prime number and return it if that is the case. This procedure is repeated until a prime number is found. Since, all the sampled numbers are uniformly chosen, this implies that the found prime number must also be uniformly sampled. However, it is unclear on how long this algorithm needs to terminate, since in theory, it could keep sampling non-prime numbers (Las Vegas algorithm). The expected running time is $(b - a + 1)/\Pi(a, b)$, where $\Pi(a, b)$ is the number of primes in the interval $[a, b]$. According to the prime number theorem, $\Pi(1, b) > b/\log_2 b$, which means that prime numbers are ”dense” and the algorithm will terminate in expected logarithmic time for a reasonable choice of a and b.

6.1.5 Dealing with Different Data Types

It is desirable to store many different data types in a dictionary. However, many constructions for dictionaries (including the one proposed here) rely on using hash functions to store elements. The hash functions are often defined on bit-level. However, for practical applications, this comes with challenges. For example, if one wants to compute the hash values of strings. These are usually quite long compared to simpler keys, like numbers, which makes the computation time of hashes take a considerable amount of time. This might make it preferable to use different hash functions for these. In addition, operating on the bit-level representation is usually more complex than dealing with higher level data types.

For these reasons, we decided to implement the hash functions only for certain data types, instead of writing a generalized implementation that operates on the bit-level of inputs.

6.2 Failure of Data Structures

Some of the data structures described in this work may fail and even if this is under a negligible probability, it is still highly undesirable in practice. Examples are the queue (see Section 4.1), the cycle detection mechanism (see Section 4.2), and others. This failure mostly arises because of the choice of hash functions from the hash family. Luckily, this also means that simply choosing new hash functions until the data structure is not failing anymore solves this problem. Nevertheless, it is still important to keep track of how often these failures happen, since they have a high cost. Then, adjustments to the parameters of the data structure could be made, to prevent these failures in the future.

7 Experiments - Setting Parameters

Backyard Cuckoo Hashing uses many parameters that can be set (e.g. number of bins, size of bins, size of the backyard, etc.). While [1] provides ideas of how to parametrize these (using O/Θ-Notation), it is not clear how to set the parameters to obtain good practical performance. In this section, different parameter settings are evaluated empirically.

In Section 7.1 it is examined how different bin sizes impact the number of elements that cannot be stored in these (and thus in the first level of the backyard construction). And in Section 7.2, the parameters for the second level of the backyard construction are examined, which includes the queue, the cycle detection mechanism, and the cuckoo tables, as well as setting the number of iterations per insertion.

7.1 Parameters of the First Level of the Backyard Construction

In the backyard construction, elements are sequentially added to one of the bins in the first level (if there is space in that bin), and otherwise inserted into the backyard. This problem is known under the name Balls into Bins [12] and is well-studied on a theoretical level.

In this section, an experiment is presented that determines which percentage of elements overflow when throwing n balls into m bins, that are each of size d. This is vital for being able to set the parameters of the second level of the backyard construction, so that one knows approximately how many elements will need to be stored in the second level.

7.1.1 Experimental Setup

The experiment is conducted by generating $n = 100,000$ unique unsigned 32-bit integers (balls) from $[0, 2^{32} - 1]$ and then sequentially inserting these into m bins of size d. A bin is assigned to an element using a hash function $h : [0, 2^{32} - 1] \to [m]$. Notice, that since the used hash function is uniform and the elements are fully random, the distribution into bins is likely fully random. If there is space in the assigned bin, then the element is inserted there and the remaining space decreased by 1. Otherwise, the counter for overflowing elements is increased by 1.

The experiment is done for different load factors (elements inserted divided by space used) and bin sizes d. Load factors are chosen to be close to 1, since full memory utilization was one of the main goals of backyard cuckoo hashing. Bin sizes are chosen to be small, since otherwise access times in bins are slow (since they have linear access time and the perfect hashing augmentation presented in Section 5.4 only works for small bin sizes). The number of bins is then computed with the formula $m = \lfloor n/(d \cdot l) \rfloor$. Each parametrization is repeated 100 times to account for random variability and to ensure the robustness of the results. The average and standard deviation of these is then taken and shown in the figures.

7.1.2 Results

In Figure 8 one can see the average percentage of overflowing elements for different bin capacities and load factors (which together determine the number of bins). Each data point also shows the standard deviation using two horizontal lines.

One may expect that with increasing bin size, the number of overflowing elements decreases. This seems to be the case. The intuition for this hypothesis is that if one considers the situation with only one bin, then all elements will be contained (assuming $l \leq 1$). And if there are only two bins, then both will have nearly 50% of elements and there will be very few overflowing elements. This seems to extend to all bin capacities.

However, as already discussed in Section 5.2, with increasing bin capacity the time it takes to store, lookup, or remove an element also increases linearly. While the perfect hashing augmentation presented in Section 5.4 can make these operations work in constant time, it is only applicable for small bin capacities (e.g. $d = 4$). For example, taking bin capacity $d = 8$ would already result into one of the lookup tables to be of size $2^{54} \cdot 48 \approx 8.6 \cdot 10^{17}$ bits. As such, there is a tradeoff between minimizing the number of overflowing elements and the time it takes to operate a bin.

Furthermore, one would expect that the number of overflowing elements decreases, as the load factor decreases, since there is more space for the same number of elements. While this seems to be the case, there seems to be a diminishing reward for setting higher and higher bin capacities.

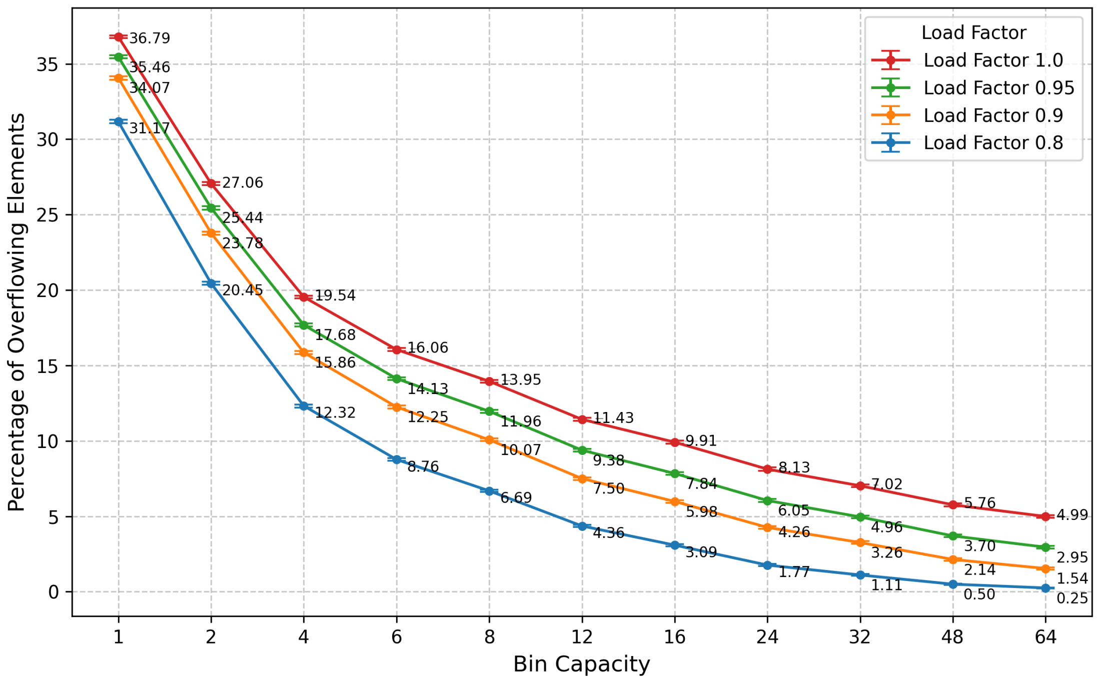


Figure 8: Balls into Bins Experiment with Random 32-Bit Data and Tornado Hash

7.2 Parameters of the Second Level of the Backyard Construction

The second level of the backyard construction uses many data structures, i.e. the queue, the cycle detection mechanism, and the cuckoo tables T_1, T_2 that must all be parameterized. Also, the number of iterations per insertion, i.e. L, must be parameterized. Using the information from the experiment done in Section 7.1, one can now approximately say how many elements will end up in the second level of the backyard construction for different bin capacities.

In this section, an experiment is presented which gives insight on how many elements end up in the different data structures of the second level of the backyard construction using different parameters for L. This is necessary, to make sure that these structures neither overflow nor waste memory by overallocating.

7.2.1 Experimental Setup

The experiment is conducted by inserting $n = 10,000$ unique unsigned 32-bit integers from $[0, 2^{32} - 1]$ into the backyard construction. The experiment is done for different values of $L \in \{2, 4, 8, 12, 16, 20\}$, which determines how many iterations are performed in the insert function (see Algorithm 5). Furthermore, the experiment is done for different bin capacities (the same ones, as in Section 7.1). For each configuration, the experiment is repeated 100 times to account for random variability and ensure robustness of the results.

The queue and the cycle detection mechanism are initialized in a way to ensure that they do not overflow. The cuckoo tables are initialized to store elements depending on how many elements are expected to overflow. The expected number of overflowing elements $overflowing$ is taken from Section 7.1.2, and then with the formula $overflowing \cdot 2/3$ the size of each cuckoo table is set (of which there are two). For example for bin capacity 8, it is expected that 13.95% of elements overflow and as such each cuckoo table is initialized to store at most $n \cdot 0.1395 \cdot 2/3 = 930$ elements. We decided to do this in this way, since this will make the cuckoo tables end up with a load factor of about 75%, for which insertions in it should take a considerable amount of time (for an explanation, see Section 3). This way, we hope to see which values of L suffice for different load factors of the cuckoo tables. It should be noted that operations in the first level bins are executed in one iteration of the insertion loop (even though these take $O(d)$ time).

7.2.2 Results

While, the experiment was done for all bin capacities that were examined in Section 7.1.2, we decide to only show it for bin capacity $d = 8$ here, although the same patterns can be noticed for other bin capacities as well. The figures for different bin capacities can be found in our GitHub repository[†].

In Figure 9 and Figure 10 one can see the average size of the queue and the average load factor of the cuckoo tables during the insertion process respectively. The cuckoo tables only start getting used, once about 40% of elements are inserted and then start getting used more and more. This is to be expected, since an element only ends up in the second level if their respective bin in the first level is already full. And for low load factors of the whole construction, most elements will end up in the first level (which was demonstrated by the experiment done in Section 7.1.2).

Furthermore, as discussed in Section 3, one would expect that as the load factor of the cuckoo tables increases, the required number of iterations to insert an element also increases. This seems to hold true, since the load factors of the cuckoo tables seems to be the same across different values of L, until the load factor reaches about 50%. Then, higher values of L seem to be required so that elements can be inserted successfully. Figure 9 also supports that claim, since also, at around 9000 inserted elements, the average size of the queue starts to differ for different values of L. There seems to be a diminishing return for increasing L, as the lines for $L \in \{12, 16, 20\}$ are all closer together than the lines for $L \in \{2, 4, 8\}$. Additionally, the average queue size starts to increase dramatically when the load factors of the cuckoo tables reach about 55%. This is highly undesirable, since one would want the queue to remain very small, since it has a high memory usage relative to the number of elements that it can store. It seems sensible to set the sizes of the cuckoo tables in a way, so that their load factor is around 50%. This in combination with a sensible value of L (e.g. $L \in [8, 12]$) will keep the queue small, which is highly desirable.

In Figure 11 and Figure 12 one can see the maximum size over repetitions for the queue and the cycle detection mechanism respectively. It is important to not only consider average sizes for these auxiliary data structures, since a failure of one of those will result into the failure of the whole backyard construction (or at least the rehashing of the auxiliary data structure). This is why, we also show maximum sizes for these data structures. Figure 11 seems to show as well that choosing values too small for L can sometimes lead to the explosion of the queue size (which is especially visible for $L = 2$ here). As such, one should take caution choosing a small value for L.

The size of the cycle detection mechanism seems to not be dependent on L, which is expected. This is because the size is at most the maximum size of a connected component of the cuckoo graph (as discussed in Lemma 4.3) and thus not dependent on L. Again, the cycle detection mechanism size stays small until the load factor of the cuckoo tables approaches 50%. This is expected, since for lower load factors, the size of connected components is smaller. Unfortunately, even at load factors of about 50% of the cuckoo tables, the cycle detection mechanism already needs a capacity of about 100. Perhaps, one could also not use it at all and instead resolve to a maximum number of moves per element, as it is done for standard cuckoo hashing (see Section 3).

[†]`https://github.com/mgroling/Backyard-Cuckoo-Hashing/tree/main/experiments/auxiliary_structures_size/plots`

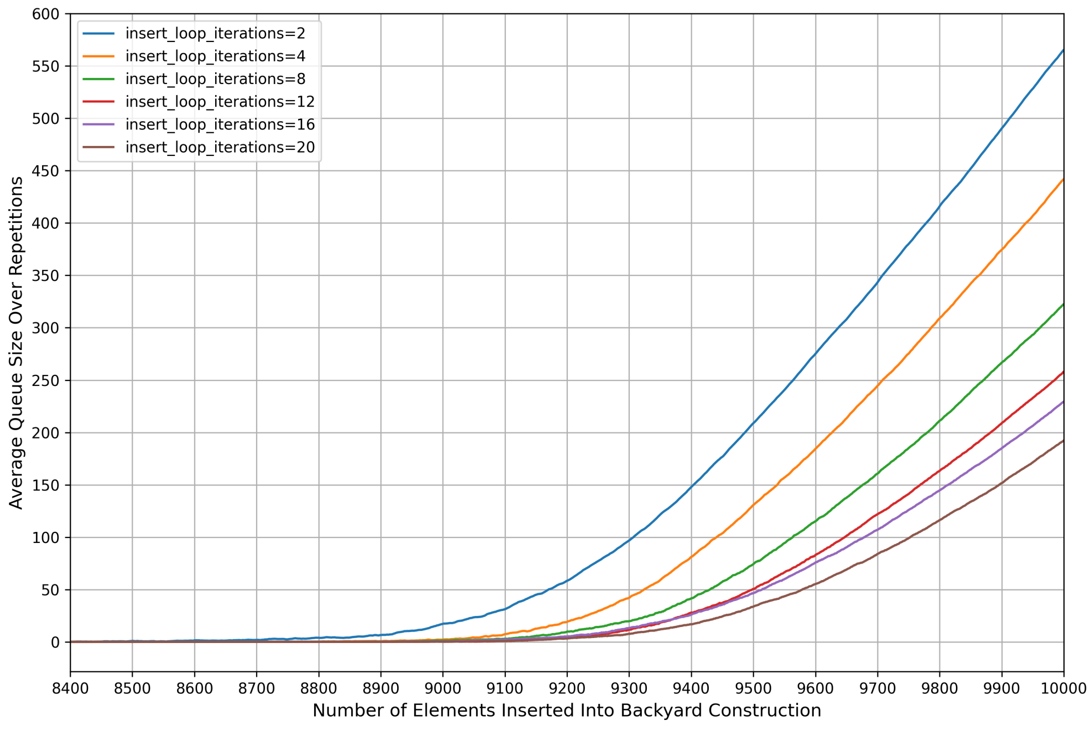


Figure 9: Average Queue Size during the Insertion of Elements into the Backyard Construction for Bin Capacity 8

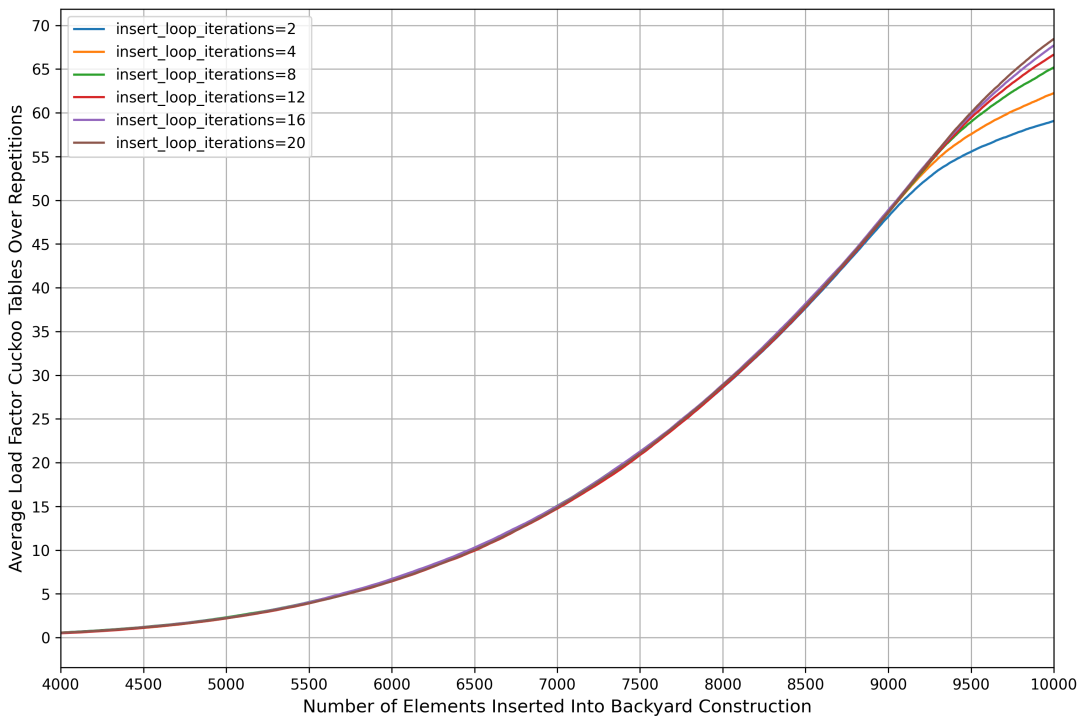


Figure 10: Average Load Factor of the Cuckoo Tables during the Insertion of Elements into the Backyard Construction for Bin Capacity 8

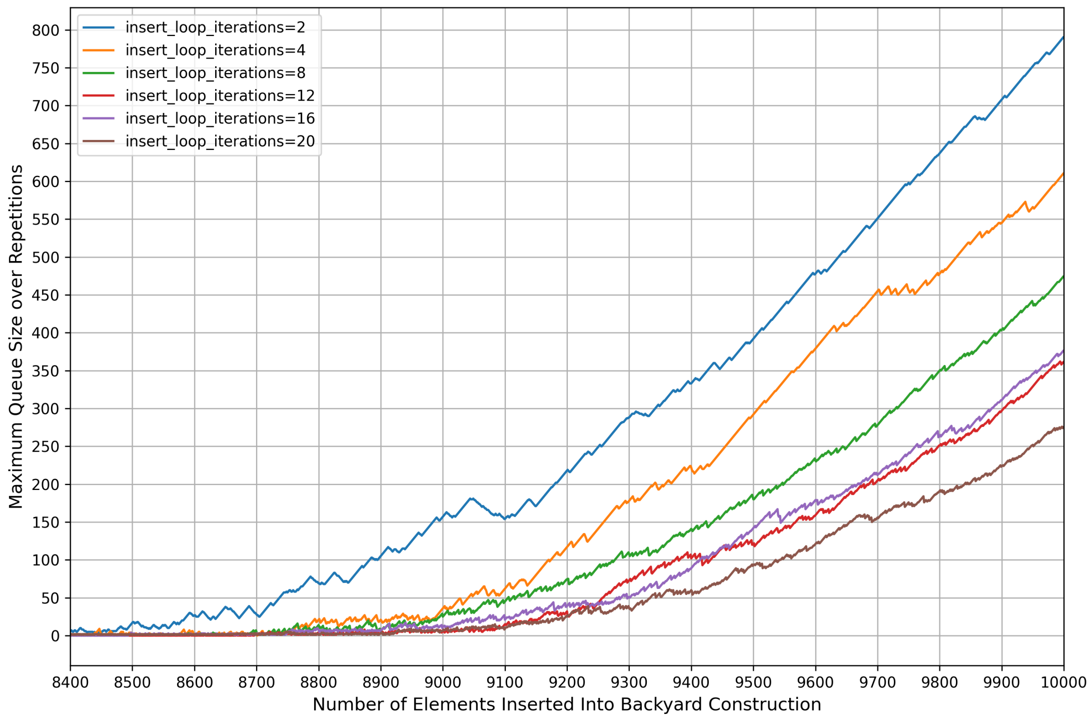


Figure 11: Maximum Queue Size during the Insertion of Elements into the Backyard Construction for Bin Capacity 8

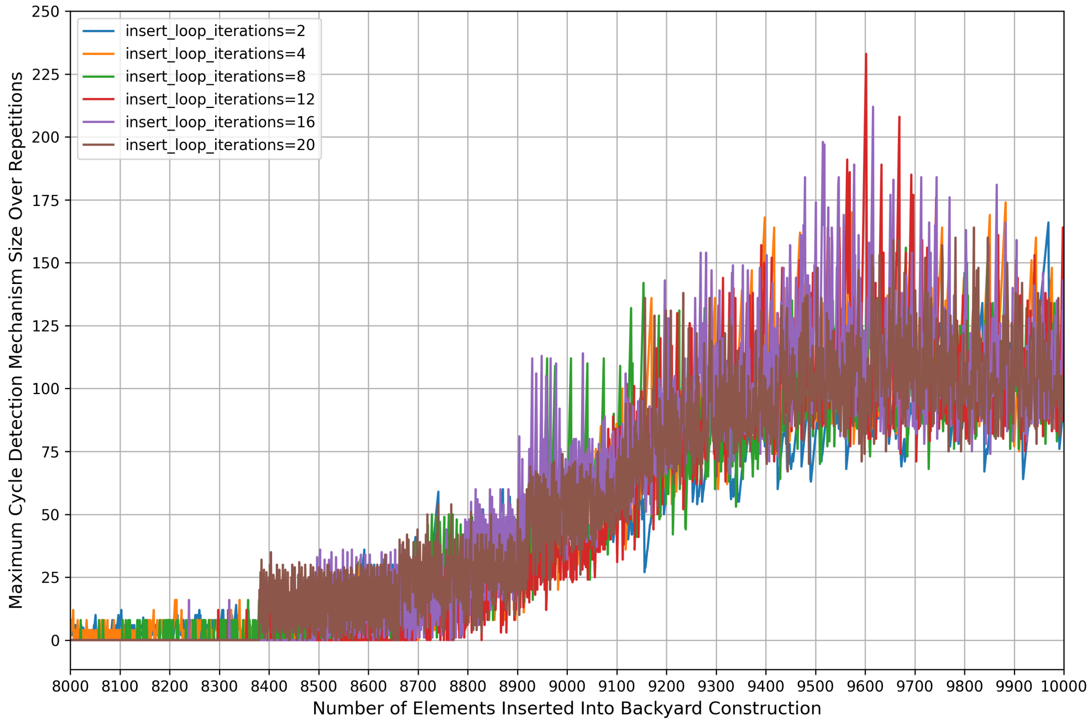


Figure 12: Maximum Cycle Detection Mechanism Size During the Insertion of Elements into the Backyard Construction for Bin Capacity 8

8 Discussion and Outlook

In this work, backyard cuckoo hashing [1] was examined on a theoretical and practical level. The theory of the construction was presented in a more extensive and hopefully easier understandable way. Furthermore, the construction was implemented, and different experiments were conducted to explore suitable parameters.

While the construction promises highly desirable guarantees in theory, such as high memory usage and constant time operations, it is still not clear how practical it is. This may be done by comparing it with other state-of-the-art dictionary constructions. Unfortunately, this was not feasible within this work.

As such, future work on this topic is still necessary. Work on this topic could be, but is not limited to the following:

8.1 Further Testing of Parameters for Good Practical Performance

We performed two experiments to find suitable parameters for practical use cases in this work. However, in these experiments we only obtained an approximation for which parameters might lead to good practical performance. It would be interesting to compare instantiations with different parameter settings to each other and see which work best.

8.2 Exploring Modifications of the Construction and their Performance Impact

As already mentioned in [1], there are many small adjustments that could be made to the backyard construction that would perhaps perform better:

- Instead of using a single hash function to hash elements to the first level bins, one could use multiple. This would likely reduce the number of overflowing elements further, allowing the backyard to be smaller (which has worse memory usage than the first level).
- Instead of using a simple lookup-array for the first level bins, each could be a small cuckoo hash table as well. This would allow us to increase the bin capacity, which will also reduce the number of elements that end up in the second level.
- Perhaps, removing the cycle detection mechanism entirely and instead limiting the number of moves in the cuckoo tables to a constant could lead to better performance, as it reduces overhead.
- For our experiments, we used Tornado Tabulation Hashing [11] as the hash function for the first level bins and the cuckoo tables of the second level. It would be interesting to see how different hash functions perform against each other.

Of course, for the construction to be practically relevant, a comparison with other state-of-the-art dictionaries (e.g. PaCHash [13]) is necessary as well.

8.3 Examining the Practical Impact of the Proposed Augmentations

Two augmentations of the backyard are proposed, and it would be interesting to see how these would impact the practical performance of the dictionary.

- The first one was to use a perfect hashing scheme (see Section 5.4) for the first level bins. However, this limits the bin capacity to be very small (i.e. a bin capacity of 4 already requires one of the lookup tables to use $2^{20} \cdot 16 \approx 1.67 \cdot 10^7$ bits). Using lower bin capacities has the drawback of having more overflowing elements from the first level. It is questionable if this scheme fares better than simple using an array with linear time operations, since this has less overhead. Although, one significant advantage of using the perfect hashing is that only one comparison of keys has to be done (in comparison with linear in the size of the bins many for the alternative approach). As such, for large key sizes the perfect hashing approach might be advantageous, while otherwise the simple array approach might fare better.

- The second one was to use an approach to only store parts of the keys in the first level bins and use the other part to ”hash” to a bin (see Section 5.5). This approach required the use of a permutation function, however. It is questionable if this augmentation results in better practical performance due to its overhead.

References

[1] Y. Arbitman, M. Naor, and G. Segev, “Backyard cuckoo hashing: Constant worst-case operations with a succinct representation,” in *2010 IEEE 51st Annual symposium on foundations of computer science*, pp. 787–796, IEEE, 2010.

[2] R. Sedgewick and K. Wayne, *Algorithms*. Addison-wesley professional, 2011.

[3] R. Pagh and F. F. Rodler, “Cuckoo hashing,” *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[4] Y. Arbitman, M. Naor, and G. Segev, “De-amortized cuckoo hashing: Provable worst-case performance and experimental results,” in *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pp. 107–118, Springer, 2009.

[5] A. Kirsch and M. Mitzenmacher, “Using a queue to de-amortize cuckoo hashing in hardware,” in *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, vol. 75, 2007.

[6] P. Briggs and L. Torczon, “An efficient representation for sparse sets,” *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 59–69, 1993.

[7] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” in *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 106–112, 1977.

[8] A. Siegel, “On universal classes of extremely random constant-time hash functions,” *SIAM Journal on Computing*, vol. 33, no. 3, pp. 505–543, 2004.

[9] M. Dietzfelbinger and M. Rink, “Applications of a splitting trick,” in *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pp. 354–365, Springer, 2009.

[10] M. Pătraşcu and M. Thorup, “The power of simple tabulation hashing,” *Journal of the ACM (JACM)*, vol. 59, no. 3, pp. 1–50, 2012.

[11] I. O. Bercea, L. Beretta, J. Klausen, J. B. T. Houen, and M. Thorup, “Locally uniform hashing,” in *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1440–1470, IEEE, 2023.

[12] M. Raab and A. Steger, ““balls into bins”—a simple and tight analysis,” in *International Workshop on Randomization and Approximation Techniques in Computer Science*, pp. 159–170, Springer, 1998.

[13] F. Kurpicz, H.-P. Lehmann, and P. Sanders, “Pachash: Packed and compressed hash tables,” in *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 162–175, SIAM, 2023.