Fast and Simple Compact Hashing via Bucketing

Koppl, Dominik

2022-09

Koppl , D , Puglisi , S J & Raman , R 2022 , ' Fast and Simple Compact Hashing via Bucketing ' , Algorithmica , vol. 84 , pp. 2735-2766 . https://doi.org/10.1007/s00453-022-00996-y

http://hdl.handle.net/10138/347139
https://doi.org/10.1007/s00453-022-00996-y

Algorithmica (2022) 84:2735–2766
https://doi.org/10.1007/s00453-022-00996-y

Fast and Simple Compact Hashing via Bucketing

Dominik Köppl[1] · Simon J. Puglisi[2] · Rajeev Raman[3]

Received: 2 September 2020 / Accepted: 3 April 2022 / Published online: 29 June 2022

Abstract
Compact hash tables store a set S of n key-value pairs, where the keys are from the universe $U = \{0, \ldots, u-1\}$, and the values are v-bit integers, in close to $\mathcal{B}(u, n) + nv$ bits of space, where $\mathcal{B}(u, n) = \log_2 \binom{u}{n}$ is the information-theoretic lower bound for representing the set of keys in S, and support operations insert, delete and lookup on S. Compact hash tables have received significant attention in recent years, and approaches dating back to Cleary [IEEE T. Comput, 1984], as well as more recent ones have been implemented and used in a number of applications. However, the wins on space usage of these approaches are outweighed by their slowness relative to conventional hash tables. In this paper, we demonstrate that compact hash tables based upon a simple idea of bucketing practically outperform existing compact hash table implementations in terms of memory usage and construction time, and existing fast hash table implementations in terms of memory usage (and sometimes also in terms of construction time), while having competitive query times. A related notion is that of a compact *hash ID map*, which stores a set $\hat{S}$ of n keys from U, and implicitly associates each key in $\hat{S}$ with a unique value (its ID), chosen by the data structure itself, which is an integer of magnitude $O(n)$, and supports inserts and lookups on $\hat{S}$, while using space close to $\mathcal{B}(u, n)$ bits. One of our approaches is suitable for use as a compact hash ID map.

Keywords Compact hashing · Dynamic dictionary · Closed addressing · Word-packing

Parts of this work have already been presented at the 18th International Symposium on Experimental Algorithms [16].

✉ Dominik Köppl
dominik.koeppl@uni-dortmund.de

1 M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan

2 Department of Computer Science, University of Helsinki, Helsinki, Finland

3 Department of Informatics, University of Leicester, Leicester, UK

1 Introduction

In this paper, we consider practical *compact* representations of dynamic *dictionaries*. A dictionary is arguably the single most important abstract data type, formulated as follows. We are given a dynamic set S of key-value pairs $\langle x, y\rangle$, where the key x comes from a *universe* $U = \{0, 1, \ldots, u\}$ and the value y is from $\{0, 1, \ldots, 2^v\}$. Furthermore, all keys in S are distinct. A dictionary supports the following operations:

lookup(x, S): Given $x \in U$, if there is a pair $\langle x, y\rangle$ in S, return y, or a null value otherwise.
insert$(\langle x, y\rangle, S)$: Add the pair $\langle x, y\rangle$ to S if S does not have x as a key.
delete(x, S): Delete the pair (if any) of the form $\langle x, y\rangle$ from S.

Although it is possible to support the insertion of the same key with different values, we do not treat this case here for simplicity.

Dictionaries can be implemented using a number of data structures such as hash tables and balanced trees, and many standard libraries use these approaches. Our interest, however, is in highly space-efficient approaches to the dictionary problem. In the worst case, a dictionary cannot use less space than the information-theoretic lower bound needed to store S. If $|S| = n$, the lower bound for storing the keys in S is $\mathcal{B}(u, n) = \log_2 \binom{u}{n} = n \log_2 u - n \log_2 n + O(n)$ bits. The lower bound on the space for the key-value pairs in S is thus $\mathcal{B}(u, n) + nv$ bits. In what follows we abbreviate $\mathcal{B}(u, n)$ by $\mathcal{B}$, use the notation $[i] = \{0, 1, \ldots, i-1\}$ for a non-negative integer i, and write lg and ln for the logarithm to base two and base e, respectively, with $\lg^{(1)} n = \lg n$ and $\lg^{(i)} n = \lg(\lg^{(i-1)} n)$ for $i > 1$.

1.1 Dictionaries in Literature

Following standard terminology, we refer to dictionaries that use $O(\mathcal{B} + nv)$ bits as *compact* and those that use $\mathcal{B} + nv + o(\mathcal{B} + nv)$ bits as *succinct*. In recent work [3, 22], a number of applications have been highlighted for compact dictionaries including compact representations of graphs, tries and arrays containing variable-length entries. In all these applications, the $\Omega(n(\lg u + v))$-bit space usage of a traditional dictionary (even those with low wasted space such as [12, 18]) is prohibitively large.

Building on earlier work on succinct *static* dictionaries [4, 20], succinct dynamic dictionaries were proposed by Raman and Rao [26] and Arbitman et al. [1]. In the transdichotomous model with word size $w = \lg u$ bits, the above solutions use $(\mathcal{B} + nv)(1 + o(1))$ bits of space, answer lookup queries in $O(1)$ worst-case time, and perform updates in $O(1)$ expected amortized or worst-case time.[1] A slightly different data structure using $O(\mathcal{B} + nv)$ bits of space was discussed in [9]. Finally, Blandford and Blelloch [3] generalized the notion of $\mathcal{B}$ when keys are variable-length bit-strings of length at most w, and gave a dictionary with a space usage of $O(\mathcal{B}+nv)$ bits. However, these data structures are complex, and although Arbitman et al. [1] discussed ideas to make their data structure more practical, we are not aware of any implementations along these lines.

[1] These two results differ regarding the model of dynamic memory allocation used.

We are concerned with practical approaches to compact dynamic dictionaries. A practical solution by Blandford and Blelloch [2] uses $O(\mathcal{B} + nv)$ bits of space, but takes $O(\lg n)$ time to perform (a much wider range of) operations. The only other practical compact dictionary we are aware of is Cleary's *compact hash table (CHT)* [6]. For any constant $\epsilon > 0$, Cleary's CHT uses $(1+\epsilon)n(\lg(u/n)+v)+O(n) = (1+\epsilon)(\mathcal{B}+nv)+O(n)$ bits and supports lookup in $O(1/\epsilon^2)$ expected time, and updates in $O(1/\epsilon^3)$ expected amortized time. Poyias et al. [23] proposed a variant of the CHT, called the *displacement* CHT or dCHT, which supports lookup in $O(1/\epsilon)$ expected time. However, it uses $\Omega(n)$ bits more space than the CHT (a simplified dCHT [23] in fact takes $\Theta(n \lg^{(5)} n)$ bits more space than the CHT), and particularly as ϵ approaches 0, the practical performance of these approaches deteriorates significantly.

1.2 Hash ID Maps

Related to a dynamic dictionary is the notion of a *hash ID map*, which stores a set $\hat{S}$ of n keys (without user-provided values) from a universe U, and associates each key in $\hat{S}$ with a unique integer, chosen by the data structure, from a range $[\rho]$. If $x \in \hat{S}$, lookup(x) returns the integer associated with x. A requirement is that the integer associated with x does not change during the lifetime of the data structure, although our solution, as well as the previous solutions, require that the data structure is destroyed and rebuilt after $\Theta(n)$ update operations. In addition, we would like ρ to be in $O(n)$. Finally, the space usage of a compact hash ID map should be close to $\mathcal{B}(u,n)$, which is the information-theoretic lower bound for storing $\hat{S}$. A compact hash ID map has many applications, including compact representations of tries [14, 23] (and potentially other compact data structures), LRU cache management [27], and naming in string processing [19]. Implicit in the work of Darragh et al. [8] is a compact hash ID map whose space usage is $(1+\epsilon)\mathcal{B}$ bits, has $\rho = \Theta(n \lg n / \lg\lg n)$ with high probability, supports $O(\epsilon n)$ updates before requiring rebuilding, and performs insert and lookup in $O(1/\epsilon^2)$ expected time. Implicit in the work of Poyias et al. [23] is a compact hash ID map whose space usage is $(1+\epsilon)\mathcal{B}$ bits, has $\rho = O(n)$, supports $O(\epsilon n)$ updates before requiring rebuilding, and performs insert and lookup in $O(1/\epsilon)$ expected time. Here $\epsilon > 0$ is a user-specified constant.

In this article we consider the use of *bucketing* to design practical CHTs and compact hash ID maps, an approach that is distinguished by its simplicity, and is the basis of the theoretical work of Raman and Rao [26], as well as earlier CHTs.

We should also address practical hash tables here, in particular those with small space or SIMD instructions.

1.3 Our Contributions

We propose simple and practical CHTs with:

- $\mathcal{B} + nv + O(n \lg\lg u)$ bits, performing insert and lookup in $O(\lg(u/n))$ and $O(\min\{\lg(u/n), \lg\lg u\})$ expected time respectively. This approach, despite being theoretically inferior, is extremely simple. It also yields a compact hash ID map with $\rho = O(n)$, $\mathcal{B} + O(n \lg\lg u)$ bits of space usage with support for $\Omega(n)$ updates

before requiring rehashing, with $O(\lg(u/n))$ expected time for both insert and lookup.
- $\mathcal{B} + nv + O(n)$ bits, performing insert in $O(\lg(u/n))$ worst-case time and lookup in $O(1)$ expected time. However, this approach is not suitable for use as a compact hash ID map.

These results are obtained on the word RAM model with word size $w = \lg u$ and under the *simple uniform hashing assumption*, i.e., we assume that there is a hash function $f : [u] \rightarrow [h]$ with the probability of $1/h$ that $f(x_1) = f(x_2)$ for two pairwise different elements $x_1, x_2 \in [u]$.

Despite their poor asymptotic complexities, these approaches are simple and designed for practical performance. In this evaluation, we consider two distinct scenarios:

- $\lg(u/n)$ and the value bit width v are both small (for instance at most eight), motivated by the CDRW-array [23], which compactly stores a dynamic array A, most of whose entries can be stored in very few bits. The CDRW-array works as follows: For every bit-width $v \geq 1$, let I_v denote the set of indices such that $A[i]$ with $i \in I_v$ is a v-bit value. Then the CDRW-array creates for each bit-width v a CHT, and stores the key-value pairs $(i, A[i])$ in this CHT for all $i \in I_v$. For small v, the set of indices containing v-bit values are often a large proportion of all indices, so $\lg(u/n)$ is small as well. In this scenario, keeping space usage low is a priority.
- $\lg(u/n)$ and the value bit width v are both relatively large, for instance larger than eight. This models a number of scenarios such as storing the adjacency matrix of a fairly sparse weighted or labeled graph. In this scenario, we would be competing against other memory-efficient implementations of conventional hash tables, and speed would also be an important criterion.

On the implementation side, we offer three novel contributions. Firstly, our hash table needs to be periodically resized as elements are added. Rather than resizing based on the overall number of keys, we resize based on the size of the maximum bucket. Secondly, we use SIMD accelerated techniques [28] to accelerate searches in a bucket. Last but not least, we propose and study a combination with an overflow hash table to defer the need for rehashing.

1.4 Discussion

We briefly summarize how our approaches differ from other compact hash tables. Compact hash tables store keys not in plain form, but apply *quotienting*, which can be understood as a function mapping a key to a slot in the hash table and a quotient that needs at most as many bits as the original key. This mapping is invertible in the sense we can restore the key by knowing the computed slot and its quotient. Unfortunately, the computed slot may not be vacant, so additional information needs to be stored if the quotient is placed in a different slot. We call this information *displacement information*.

Regardless of whether a hash table is compact, hash tables are usually implemented either (a) as *open addressing*, whereby all keys are stored in a single table, or (b) as *closed addressing*, where keys are mapped to buckets. However, combining open addressing with compact hashing leads to difficulties, as compact hashing does not store entire keys, but only *quotient* information [6], and the overhead of storing and maintaining additional information to recover the keys from the quotients is quite high, since open addressing schemes need to have some mechanism for resolving collisions, such as linear probing [6, 23]. The problem is exacerbated by the use of quite high *load factors* to keep the space usage low. The use of high load factors can be avoided by switching to a *sparse* hash table layout.[2] In contrast to standard open addressing (storing elements in a single array), the sparse hash table layout uses a bit-string to mark positions at which elements are stored, and represents the keys themselves in a collection of small, variable-sized arrays. This allows for low load factors with a moderate space overhead, and works well with compact hashing [11, see Section "Outlook"]. However, the overhead of decoding the displacement information for restoring a key from its quotient remains a concern.

In contrast, closed addressing does not use collision resolution: in closed addressing, a distribution of the keys to buckets is performed. A *bucket* can be seen itself as a dictionary providing the methods insert, delete, and lookup. Although it is therefore possible to define recursive data structure, a bucket is usually represented by a singly linked list like in the `unordered_map` of the C++ standard library `libstdc++` [5, Sect. 22.1.2.1.2]. Such a bucket representation is also called *chaining*. The overhead of chaining can make hash tables using it space-consuming and slow, and modern implementations of hash tables tend to focus on open addressing.

In a compact hash setting, the buckets contain quotients of keys. In contrast to hashing via open addressing, we do not need to maintain any information to recover a key from its quotient. However, buckets have obvious overheads such as pointers to them and auxiliary information such as their sizes. The challenge is how to balance the overheads of the buckets (which grow as the number of buckets increases) with the size of the quotients stored in the buckets (which reduces as the numbers of buckets increase). Theoretical solutions (such as [26]) to this problem are complex (e.g., recursing on the quotients in a bucket). We give up on asymptotic worst-case performance in order to find solutions that work in practice.

1.5 Paper Overview

We begin with a review of some probabilistic bounds in Sect. 2, and then start with a theoretical description of our algorithms in Sect. 3. In Sect. 4 we describe the implementations, which depart from theory in some ways. Section 5 presents our main experimental evaluation. Section 6 supplements our experimental results with other hash tables and changes in the experimental settings. In Sect. 7, we focus on broadword and SIMD instructions to accelerate the search of a key or quotient in a bucket, where it turns out that SIMD instructions are especially useful if we have a good lower bound on the number of elements n to hash. How space reservation helps us to speed up this search while keeping the space requirements low is studied briefly in Sect.

[2] https://github.com/sparsehash/sparsehash.

7.2.2. Our last experiment in Sect. 8 studies the use of overflow hash tables to defer rehashing. Finally, Sect. 9 concludes and states directions for further work.

Compared to the conference version [16], we elaborated on the theoretical foundation of the distribution of elements performed by our hash table layout, and enhanced our experiments with a comparison against a richer set of hash tables, and with variants using SIMD instructions or overflow tables.

2 Probabilistic Bounds

Before describing our new hash table, we study some tail bounds on binomial distributions that we use to argue and/or estimate the performance of our hash table. Let $\Pr[X > v]$ denote the probability that a random variable X yields a value larger than v. We start with the following form of the Chernoff bound:

Theorem 1 [25, Theorem 4.1] *Suppose $X_1, \ldots, X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = E[X]$ denote the sum's expected value. Then for every $\delta > 0$,*

$$\Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

We derive two conclusions from this theorem, which we will use for arguing about the insertion into our hash table, and for the rehashing.

Corollary 1 *Let $h > 0$, $u > 0$ be powers of 2, and let $t \geq 0$ be a constant. If we place $h \lg u$ balls into h bins uniformly at random then with probability $1 - u^{-t}$ no bin will receive more than $3.4(t+1)\lg u$ balls.*

Proof Fix a particular bin b and let X_i be a random variable such that $X_i = 1$ if the i-th ball is placed in bin b and 0 otherwise. By the uniform distribution of the balls, $\Pr[X_i = 1] = 1/h$, and $X = \sum_{i=1}^{h \lg u} X_i$ is the random variable that specifies the number of balls that are placed in bin b. Then $\mu = \sum_{i=1}^{h \lg u} \Pr[X_i = 1] = \lg u$ is the expected number of balls in b. Using Theorem 1 with $\delta = 3.4(t+1) - 1 > 0$ we see that

$$\begin{aligned}\Pr[X > (3.4(t+1))\mu] &< \left(\frac{e^{(3.4(t+1)-1)}}{(3.4(t+1))^{3.4(t+1)}}\right)^{\lg u} \\ &< \left(\frac{e^{(3.4(t+1))}}{(3.4)^{3.4(t+1)}}\right)^{\lg u} \\ &< \left(\frac{e^{3.4}}{(3.4)^{3.4}}\right)^{(t+1)\lg u} \\ &< (0.5)^{(t+1)\lg u} < u^{-(t+1)},\end{aligned}$$

where we used that $e^x/x^x < 0.5$ for all $x \geq 3.34432$. Finally, we note that the probability that *any* bin receives more than $(3.4(t+1))\lg u$ balls is at most $u \cdot u^{-(t+1)}$ or u^{-t}. □

We now derive a bound that we use later for the rehashing. The first corollary is used for the redistribution of a single bucket, and the subsequent one for the rehashing of the complete hash table.

Corollary 2 *Let $t \geq 0$ and $u \geq 2^{12}$ be two constants. Further let b be a bin containing at most $30(t+1)\lg u$ balls. If we randomly place each ball in b into two new bins b_1 and b_2 then with probability $u^{-(t+1)}$ none of the two new bins will receive more than $(2/3) \cdot (30(t+1)\lg u)$ balls.*

Proof First, assume that b contains exactly $30(t+1)\lg u$ balls, and that the number of balls in b is divisible by 2. Pick one of the two new bins, say b_1, and let $X_1, \ldots, X_{30(t+1)\lg u}$ be random variables so that $X_i = 1$ if the i-th ball goes to b_1 and 0 otherwise (i.e., it goes to b_2). Then $X = \sum_i X_i$ is the random variable that gives the number of balls in b_1, with $\Pr[X_i = 1] = 1/2$. The expectation is that b_1 receives half of the balls of b, i.e., $\mu = \sum_{i=1}^{30(t+1)\lg u} \Pr[X_i = 1] = 15(t+1)\lg u$. By Theorem 1, $\Pr[X > (2/3) \cdot (30(t+1)\lg u)] = \Pr[X > (1+1/3)\mu]$ can be bounded as:

$$\Pr[X > (1+1/3)\mu] < \left(\frac{e^{1/3}}{(1+1/3)^{1+1/3}}\right)^{15(t+1)\lg u} < 0.5u^{-(t+1)},$$

where we used the fact that $\left(\frac{e^{1/3}}{(1+1/3)^{1+1/3}}\right)^{\lg u} < 0.5u^{-1}$ for $\lg u \geq 12$. Finally, if b had fewer than $30(t+1)\lg u$ balls, the chance that b_1 receives more than $(2/3) \cdot (30(t+1)\lg u)$ balls cannot increase. The claim follows by summing the equal probabilities for b_1 and b_2. □

Corollary 3 *Let $h \geq 0$ and $u \geq 2^{12}$ be powers of 2, such that $h \leq u$, and let $t \geq 0$ be a constant. Let there be h bins $b_1, \ldots, b_h$, containing $\leq 30(t+1)\lg u$ balls each. If we redistribute all balls of the i-th bin b_i randomly into two new bins $b_{i,1}$ and $b_{i,2}$, for every $i \in [1..h]$, with probability $1-u^{-t}$ none of the new bins $b_{1,1}, b_{1,2}, \ldots, b_{h,1}, b_{h,2}$ will receive more than $(2/3) \cdot (30(t+1)\lg u)$ balls.*

Proof We apply Corollary 2 to all h bins, and sum up their probabilities: The probability that none of the $2h \leq 2u$ new buckets receives more than $(2/3) \cdot (30(t+1)\lg u)$ balls is at most u^{-t}, as required. □

3 Compact Hashing via Bucketing: a Theoretical View

Equipped with the probabilistic results of Sect. 2, we here outline our approach from a theoretical perspective. We use the word RAM model with word size $w = \lg u$ bits. Let $y \bmod x \in [x]$ denote the modulus of y by x, and $y \operatorname{div} x = \lfloor y/x \rfloor$ the truncated division of y by x. Let h be the number of buckets of our hash table – we assume, for

Table 1 Overview of used symbols

Symbol	Meaning
h	Number of buckets or group-buckets
q	Quotient bit width
d	Quotient value
x	Key
y	Value
n	Number of stored keys
u	Size of the universe
v	Bit width of the values
z	Constant for Chernoff bound
$b_{\max}$	Maximal elements a bucket or group-bucket can store $= z \lg u$
s	Elements in a group-bucket
m	Number of sub-buckets in a group-bucket

ease of description, that the size of the universe u and h are powers of two. We start with an invertible function $f : [u] \to [u]$, and use f to map key-value pairs to the h buckets. Now, given a key-value pair $\langle x, y\rangle$, we compute $f(x)$, and assign $\langle x, y\rangle$ to the bucket $r = f(x) \bmod h$. In the bucket r, the key x is represented by its quotient value $d = f(x) \operatorname{div} h$; observe that the key x can be recovered with $f^{-1}(dh + r)$ and that the quotient value takes $q = \lg u - \lg h$ bits. We analyze the hash tables assuming the simple uniform hashing assumption [7, Sect. 11.2], which means for all keys x in S, $f(x)$ is uniformly and independently distributed over $[h]$. This is a common and long-standing assumption, which works well in many (but not all) practical settings, and how to realize this assumption in the worst-case complexity setting is the subject of continuing research [31]. To easy the reading, we provide a list of used symbols in Table 1.

3.1 Balancing Bucket Sizes

We begin by discussing our approach to rebalancing buckets, which differs in some ways from standard approaches. Firstly, we maintain a collection of h buckets at any given time, and set a parameter $b_{\max}$ which is the maximum bucket size. If an insert causes any bucket to have $b_{\max}$ keys, we rehash as follows:

1. Allocate $2h$ new buckets (which are initially empty).
2. For each old bucket, rehash all its keys into the new buckets. Observe that a key x in bucket i is rehashed to either bucket $2i$ or $2i + 1$ based upon the "next" bit of $f(x)$, meaning that the $(\lg h + 1)$-st least significant bits of $f(x)$ (represented as a binary value) now determine the bucket.
3. Delete each of the old buckets after processing all its keys.
4. Set $h \leftarrow 2 \cdot h$.

There are a number of advantages to rehashing this way. First, rehashing by doubling the number of buckets each time gives the rehashing procedure good locality of reference, since we move the elements of an old bucket i into at most two different new buckets ($2i$ and $2i+1$). Second, our hash tables do not experience memory peaks as observed by open addressing hash tables that need to keep (mostly) all cells of the old and the new hash table in memory during the whole rehashing process. Finally, having a fixed $b_{\max}$ is helpful for using the hash table as a hash ID map, as we will see below. It also has some practical benefits from a code perspective, such as allowing header information (such as the number of keys in a bucket) to be stored in variables of fixed bit-width. Finally, it ensures we can claim worst-case time for lookup, which might be useful in some applications.

Letting n be the number of keys currently stored in the hash table, we would like to ensure the following invariants:

- With high probability, $\Omega(n+h)$ insert operations take place between successive rebuildings, and
- Buckets on average contain $\Omega(b_{\max})$ keys.

The first point ensures that the amortized cost of rehashing is $O(1)$. The second ensures that the number of buckets is low relative to the number of keys; since each bucket has a fixed overhead, this reduces the overhead per key, improving the space utilization.

To achieve this, we set $b_{\max} = z \lg u$, where z is a sufficiently large constant. We derive the following consequences from Sect. 2:

(a) For any constant $t > 0$ there is a constant $c = O(t)$ such that inserting $h \lg u$ keys into h initially empty buckets results in all buckets being of size at most $c \lg u$ with probability $1 - u^{-t}$ (Corollary 1).
(b) For u large enough and a constant $t > 0$ there is a constant d such that splitting h buckets each with $\leq d \lg u$ keys into two (in the manner described above, by rehashing the keys in a bucket into two buckets), the largest bucket remaining after rehashing will have size at most $(2/3) \cdot d \cdot \lg u$ keys with probability $1 - u^{-t}$ (Corollary 3).

The constants c and d are linearly dependent on t. In what follows, if an event happens with probability $1 - u^{-t}$, where t is a user-controllable parameter, we say it happens *with high probability (whp)*. If we choose $z = \max\{3c, d\}$, then when we rehash, we are guaranteed by (b) that the largest bucket will have size at most $(2z/3) \lg u \leq (z-c) \lg u$ keys whp. By (a), after the rehash, we can support $h \lg u = \Omega(n+h)$ insert operations whp before any bucket overflows. Note that this also ensures that the minimum average size of a bucket is at least $\lg u$, as the average size of a bucket is halved by rehashing, but whp at least $h \lg u$ keys need to be inserted into the hash table (which are distributed across all buckets) before a bucket becomes full again.

In what follows, we present two realizations of our up so far abstractly described approach on hashing by bucketing. Table 2 gives an overview on the differently set parameters of each approach.

Table 2 Juxtaposition of our two proposed variants in Sects. 3.2 and 3.3

Measure	bucket Sect. 3.2	grp Sect. 3.3
h	$\Omega(n/\lg u)$	$\Omega(n/\lg u)$
q	$\lg(u/n) + O(\lg\lg u)$ bits	$\lg(u/n) + O(1)$ bits
space	$\mathcal{B} + nv + O(n \lg\lg u)$ bits	$\mathcal{B} + nv + O(n)$ bits
lookup time	$O(\lg(u/n) + \lg\lg n)$ or	$O(\lg(u/n))$ or
	$O(\lg\lg u)$ if sorted	$O(1)$ expected

3.2 Simple Compact Hashing

The above scheme can be realized as follows: The buckets are represented by an array of h pointers, each pointing to an array of w-bit words that stores the key quotients and the values. The overhead per bucket is a constant number of machine words representing the unused space within the last word of an array, plus the pointer to the bucket. Since the average number of keys in a bucket is $\Omega(b_{\max})$, the number of buckets is $O(n/b_{\max})$, and since $b_{\max} = \Theta(\log u) = \Theta(w)$, summed over all buckets, this overhead is $O(n)$ bits.

Each key is represented by a quotient using $\lg u - \lg h$ bits. Since $h = O(n/\lg u)$ by the above discussion, the number of bits for a quotient is $q = \lg(u/n) + O(\lg\lg u)$ bits, and the overall space bound is $\mathcal{B} + nv + O(n \lg\lg u)$ bits as claimed, where v is the bit width of the values. A bucket, with a total of s key-value pairs in it, is represented as two arrays of length s, one with each entry being q bits wide (which holds the quotients), and one with each entry being v bits wide (which holds the values). The i-th entry in each array corresponds to the same key-value pair.

To perform a lookup, the quotients in a bucket are scanned. The scanning can be done using standard word-parallel tricks [15] in time proportional to the number of words that represent the quotients in a bucket, or in $O(\lg \frac{u}{n} + \lg\lg u)$ time. To perform an insert operation, a new array of the appropriate size is allocated, the existing bucket is copied over to the new array, and the new key is added to the end of the new array: this also can be done in $O(\lg \frac{u}{n} + \lg\lg u)$ time. A delete operation is processed analogously. We first search the position in the respective bucket of the search key, delete its entry, and shift all succeeding entries to the left to fill up the empty space. A rehashing can be triggered if a minimum number of elements in a bucket is reached.

To use this hash table as a compact hash ID map, suppose that a key is stored at the j-th position of the i-th bucket and $j < b_{\max}$.[3] Then the ID associated with this key is the integer $i \cdot b_{\max} + j$. Clearly, the ID only remains valid until the data structure undergoes rehashing, but there are $\Omega(n)$ insert operations between successive rehashings. To support delete, we can use the classic tombstone strategy to change a deleted element into an invalid entry.

Sorted Buckets If we do not need a hash ID map, we can improve the time for lookup by keeping the buckets sorted with respect to the quotient values. Keeping the quotients

[3] More precisely, the quotient of this key.

sorted after an insert operation can again be done in $O(\lg(u/n) + \lg\lg u)$ time. This allows us to perform a binary search on a bucket for lookup, thus reducing the time to $O(\lg\lg u)$. As quotients are assumed to be uniformly distributed, we can also run interpolation search [21] to perform lookup in $O(\lg^{(3)} u)$ average time.

Duplicate Keys In scenarios that require duplicate keys (which we do not treat in the rest of the paper), the sorting can become additionally appealing in practical terms. That is because we can answer a lookup query for unique keys by finding an occurrence of the queried key in the assigned bucket, while we must scan the complete bucket to collect equal keys when allowing duplicate keys. Hence, when allowing duplicate keys, we expect that a lookup operation can be considered as slow as an unsuccessful lookup, i.e., when the queried key is not present in the hash table. By sorting the buckets, we group together the elements having the same key, and do not need to scan the entire bucket.

3.3 Space-Efficient Compact Hashing

In our more space-efficient variant, we add a hierarchical layer splitting the role of the buckets into *group-buckets* and *sub-buckets*: We group together $m = \Theta(b_{\max})$ consecutive sub-buckets into a *group-bucket* such that we have h group-buckets and $m \cdot h$ sub-buckets in total. On the one hand, group-buckets play the same role as buckets previously did in terms of rehashing, i.e., rehashing is done when a group-bucket has more than $b_{\max}$ keys hashed to it. On the other hand, sub-buckets are the buckets which elements are hashed into. Specifically, a key x is mapped to sub-bucket j within group-bucket i if $i = f(x) \bmod h$ and $j = (f(x) \operatorname{div} h) \bmod m$. Within its sub-bucket, the key x is represented by its quotient value $d = f(x) \operatorname{div} (h \cdot m)$.

A group-bucket, with a total of s key-value pairs in it, is represented as two arrays of length s, one for the quotients and one for the values. The quotients of all keys hashed to the same sub-bucket are stored in a consecutive range in the array, and a bit-string of length $m+s = O(w)$ demarcates the sub-bucket boundaries by concatenating the sizes of the sub-buckets, written in unary. Specifically, if the sub-buckets in a group-bucket have sizes $n_1, \ldots, n_m$, the bit-string $0^{n_1}10^{n_2}1\ldots0^{n_m}1$, which is of length $m + s$, is stored to demarcate bucket boundaries. The overheads of this approach (including the demarcating bit-string) are at most $O(n)$ bits. However, the quotients stored in each sub-bucket now only take $\lg u - \lg n + O(1) = \lg(u/n) + O(1)$ bits. To see that, we observe that our hash table has $h \cdot \Theta(m) = h \cdot b_{\max} = \Theta(n)$ sub-buckets. That is because the average number of elements in a group-bucket is $\Omega(b_{\max})$ (cf. the two invariants in Sect. 3.1), and therefore $h = \Theta(n/b_{\max})$. Hence, we hash the keys into $\Theta(n)$ sub-buckets, and therefore their quotients have a bit-length of $\lg u - \lg n + O(1)$. Thus, the overall space usage is $\mathcal{B} + nv + O(n)$ bits.

To perform an operation for a given key, we need to find the sub-bucket which the key belongs to. Random access to this sub-bucket can be obtained by performing a broadword select operation [32, 33] on the demarcating bit-string in $O(1)$ time. Since the number of sub-buckets is $\Theta(n)$, the expected sub-bucket size is $O(1)$ and search within a sub-bucket takes $O(1)$ expected time, and lookup is consequently supported in $O(1)$ expected time. insert is done by rewriting the entire bucket, which takes

$O(\lg(u/n))$ time as before. (Since each insert causes potentially all keys in bucket to move, this approach is not suitable for use as a hash ID map.)

3.4 Memory Allocation

Our two approaches described in Sects. 3.2, and 3.3 allocate and free a number of relatively small arrays, which can result in memory fragmentation if used with conventional memory allocators. Since each bucket (resp. group-bucket) is of size $\Omega(b_{\max})$, the number of arrays is $O(n/\lg u)$, or $O(n/w)$. From a worst-case asymptotic perspective, fragmentation can be eliminated by using customized allocators such as [13, Theorem 6]—the overhead of the allocator is $O(w^4 + n(\lg w)/w)$ bits, which is a lower-order term. This allocator, however, uses quite a lot of indirection to achieve the above space bounds, and would be unlikely to work well in practice. Given that the arrays we allocate come from a relatively small range of sizes, we can address this by use of simple allocators such as the one used in [29, Appendix A] in our implementation, albeit with a loss of worst-case guarantees.

4 Implementation

We implemented the simple approach and the space-efficient approach described in Sect. 3.2 and 3.3, and call the implementations bucket and grp, respectively. Each of the two implementations maintains h buckets (resp. group-buckets), where h is a power of two. Following the discussion of different invertible functions given in [11, Sect. 3.2], we select a fixed multiplicative function $f : [u] \rightarrow [u]$ with $u = 2^{64}$ for the experiments, which is $f(x) = 9223372036854775291 \cdot x \mod u$ with inverse $f^{-1}(x) = 3657236494304118067 \cdot x \mod u$. We use the last $\lg h$ bits (resp. $\lg(b_{\max} \cdot h)$) of its return value for the index of the assigned bucket (resp. sub-bucket) and the other bits for the quotient. We did not follow the approach of sorting the buckets as described at the end of Sect. 3.2. Instead, we append new elements at the end of the respective bucket/sub-bucket. If we set $b_{\max}$ small enough, this choice does not hurt the query times due to data locality while keeping insertions reasonably fast. For simplicity, a deletion of an element does not trigger a rehashing (hence it can happen that a hash table has empty buckets after excessive delete operations).

4.1 Maximum Bucket Size and Rehashing

For both variants, we fix $b_{\max} = 255$ instead of setting $b_{\max} = z \lg u$ as suggested in Sect. 3.1. We discuss this relatively large choice of $b_{\max}$ in Sect. 8. For now, we note that it helps to reduce the per-bucket overhead (at least 17 bytes, see Sect. 4.2).

The main additional parameter in grp versus bucket is the number of sub-buckets in a group-bucket, which we denoted by m: Given that the average number of total elements in a group-bucket is s, a bit-string demarcating the sub-bucket boundaries in a group-bucket costs us $m + s$ bits on average, and therefore a group-bucket storing s elements costs us $m+s+sv+sq$ bits on average, where $q = \lg u - \lg h$ is the quotient bit width. By doubling the number of sub-buckets per group-bucket (but keeping the total number of group-buckets), the quotient bit width decreases by one such that the average space of a group-bucket becomes $2m + s + sv + s(q - 1)$ bits, which is smaller than the original size if $m < s$. However, changing m has an effect on the time point when rehashing occurs—remember that a group-bucket can hold up to $b_{\max}$ elements. Hence, when running an insertion benchmark on two identical hash tables with differently set m, both hash tables can have different values for s after the insertions have been performed. In our experiments, determining m by counting s in the old hash table during a rehashing resulted in an overestimation of s. Therefore, we stuck with the empirically evaluated constant $m = 64$.

4.2 Buckets

We focus on three different representations for the buckets:

1. Storing quotients and values combined as pairs of the form $(q_1, v_1), \ldots (q_\ell, v_\ell)$. This is a common representation used by major hash tables such as the C++ STL `std::unordered_map`, which employs singly linked lists storing these pairs. For queries, it excels when only a few elements have to be visited. Otherwise, it is less cache-friendly than the next two approaches.
2. Storing quotients and values separately in a list, i.e., $q_1, \ldots, q_\ell, v_1, \ldots, v_\ell$. Since the quotients are stored consecutively, more quotients can be loaded into a cache line than with the previous approach. However, inserting a new element into this bucket representation is more time consuming than the above approach, because we need to shift the values before inserting a new quotient before v_1.
3. Storing quotients and values separately in a quotient bucket $q_1, \ldots, q_\ell$ and a value bucket $v_1, \ldots, v_\ell$. This approach supports fast insertions and has a good cache behavior as well, but we need to spend an additional pointer for maintaining two buckets instead of one like in the two previous approaches.

During a preliminary evaluation, we found that Approach 2 is considerably slower than Approach 3 during the insertion, while Approach 3 needs insignificantly more memory than Approach 2. We therefore choose Approach 3 for our bucket (resp. group-bucket) representation in bucket (resp. in grp). The quotient bucket stores the quotients bit-compact in a byte array by using bit operations. Consequently, the number of bits used by a quotient bucket is quantized at eight bits (the last byte of the array might not be full). We resize a bucket with the C function `realloc`. Whether we need to resize a bucket on inserting an element depends on the policy we follow. With the incremental policy, we increase the size of the bucket to be exactly one element longer, just enough for a new element to fit in. This policy saves memory as only the minimum required amount of memory is allocated. Because buckets store at most $b_{\max}$ elements, the resize takes $O(b_{\max})$ time. In practice, much of the time for resizing depends on the speed of the memory allocator. Our second resize policy, *half increase*, increases the bucket size by 50%, taking the burden off the allocator at the expense of having some unused memory.[4]

[4] Since grp is the more memory-efficient variant, there is no grp_{50}. Moreover, since a group-bucket of grp must support insertions at all ending positions of its sub-buckets, such an insertion is much more involving

5 Core Experiments

We implemented the approaches described in Sects. 3.2 and 3.3 in C++17, and refer to them bucket and grp, respectively. We subscript bucket with '++' or '50' to indicate whether the hash table resizes a bucket by, respectively, one element or by 50% (cf. Sect. 4.2). Implementations are available at https://github.com/koeppl/separate_chaining.

Evaluation Setting Our experiments were run on an Ubuntu Linux 18.04 machine equipped with 32 GiB of RAM and an Intel Xeon CPU E3-1271 v3 clocked at 3.60GHz, having, respectively, 32KB, 256KB, and 8192KB of L1, L2, and L3 cache. We measured memory usage by instrumenting calls to `malloc`, `realloc`, and `free`. The compiler was g++ version 7.5.0 with flags `-O3 -DNDEBUG -march=native`. Our benchmarks for the operations insert, lookup, and delete are available at https://github.com/koeppl/hashbench.

5.1 Bonsai Tables

We compare our implementations bucket and grp with practical implementations of compact hash tables implemented in the tudocomp project.[5] The first is called cleary, which is an implementation of Cleary's CHT [6] using linear probing. The second, called layered, is based on the dCHTs of Poyias et al. [23]: layered stores the displacement information in two associative array data structures. The first is an array storing 4-bit integers, and the second is `unordered_map` (the C++ STL hash table implementation) for displacements larger than 4 bits. There is also another hash table variant called elias, the evaluation of which we consign to Sect. 6 due to its very slow performance. All tables apply linear probing and support a sparse table layout. We refer to these methods collectively as *Bonsai tables*, and append in subscript 'P' or 'S' if the respective variant is in its plain or sparse form, respectively. We used a maximum load factor of 0.95 for all Bonsai table implementations.

5.1.1 Insertions and Lookups

Our first and main experiment measures the time and space requirements when (a) filling the hash tables with data (insert) and (b) querying the data afterwards (lookup). We filled the hash tables with 32-bit keys and 1-bit values, with keys generated via `std::rand`. Fig. 1 shows the measured time and peak memory usage when inserting an increasing number of elements into the hash tables (*construction* in the plots) and also times for lookup queries. The queries are performed in the same order as the insertions.

Time Considering construction time, $\text{layered}_{\text{P}}$ and bucket_{50} are the fastest options, while the sparse Bonsai tables $\text{layered}_{\text{S}}$ and cleary_{S} are the slowest options. Between

Footnote 4 continued
than merely appending an element to a bucket of bucket. We are unsure whether a different resize policy pays off.

5 https://github.com/tudocomp/compact_sparse_hash.

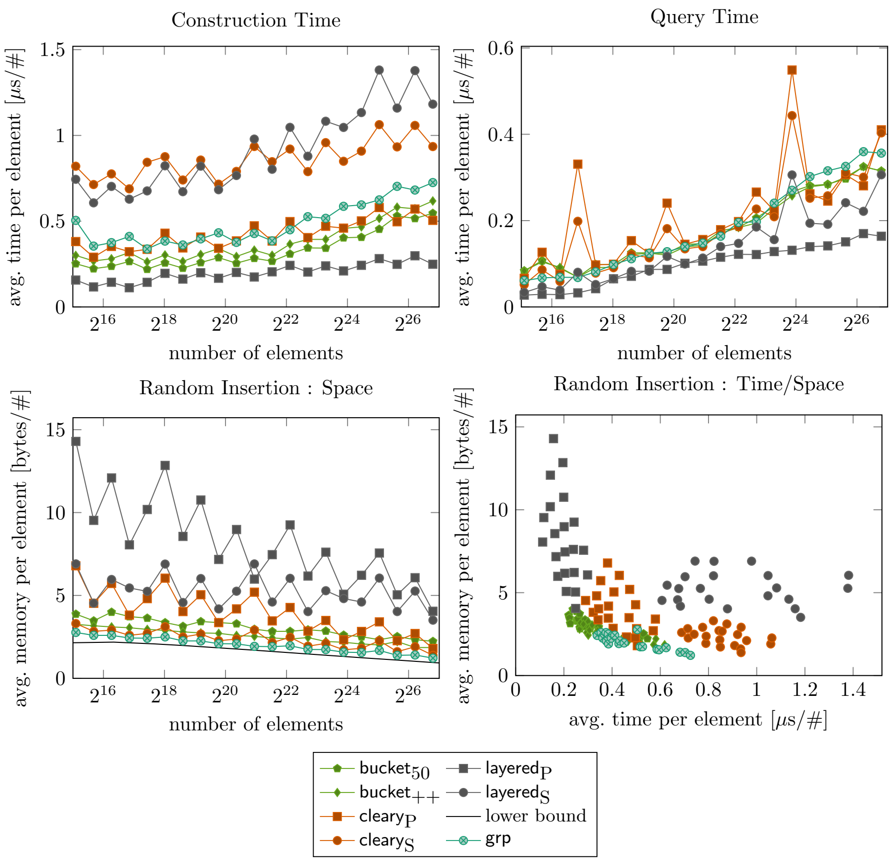

Fig. 1 *Top Left*: Time for inserting $x \mapsto 2^{10} \cdot (3/2)^x$ randomly generated 1-bit values and 32-bit keys into a compact hash table, for $x \geq 0$ (cf. Sect. 5.1). *Top Right:* Time for querying all inserted elements. *Bottom Left*: Peak memory needed during construction. *Bottom Right*: Memory and time per stored element. The lower bound is based on Stirling's approximation of $\mathcal{B}$

them are bucket$_{++}$, cleary$_{\mathrm{P}}$ and grp. Considering the query time for large instances, the difference to the construction time is that bucket and grp are here slower than all variants of cleary and layered, where again layered$_{\mathrm{P}}$ is the fastest option.

Unsuccessful Search Figure 2 shows times for unsuccessful searches (i.e., when the query is for a key not present in the table). layered$_{\mathrm{P}}$ is again generally the fastest (though with somewhat less consistent performance). grp is faster than bucket for unsuccessful searches. We can conclude that from the following fact: While grp spends additional time for finding the queried sub-bucket in a group-bucket, this pays off since the sub-buckets in grp are far smaller than the buckets in bucket.

Space grp, followed by bucket$_{++}$ and then by bucket$_{50}$, has the lowest peak memory requirements during the construction. The main reason is that, unlike the other approaches, we do not need to store displacement information. The size of a group-bucket in grp approaches $b_{\max}$ much better than a bucket in bucket, which helped grp

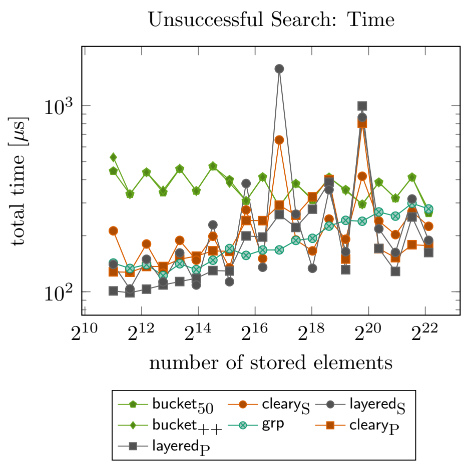

Fig. 2 Time for looking up 2^{10} random keys that are not present in the compact hash tables

to delay rehashings. cleary$_S$ beats on some instances bucket$_{++}$ (but not grp) while having significantly slower construction times than bucket or grp. However, the relatively large memory reallocation during a rehashing prevents the memory requirement of cleary$_S$ to stay below bucket$_{++}$ for a longer time.

In summary, layered$_P$ is consistently the fastest compact hash table in our experiments for both insert and lookup queries and is also the most space consuming. cleary$_P$, layered$_S$, cleary$_S$ offer different trade-offs, being either faster at insertions, lookups, or using less space. bucket and grp have the lowest space requirements, but relatively high query times. The maximum bucket size $b_{\max}$ gives us a dial for trading speed for space-usage with bucket and grp.

5.1.2 Spikes in Time and Memory

The Bonsai tables have clear spikes in their time and space-usage plots, which our hash tables do not have. To understand this phenomenon, recall that the Bonsai tables use linear probing with a maximum load factor of 0.95. (Another evaluation with a factor of 0.5 follows in Sect. 6 with Fig. 6.) As the Bonsai tables approach fullness, insertion and query times deteriorate, which is the case just before a peak in the space usage plot (Fig. 1, bottom left). The peak itself is the consequence of rehashing having been performed. Peaks are more pronounced for the non-sparse variants, which keep all buckets of both the old and new hash tables in memory during rehashing.

Analysis We observe that, while the query times for cleary degrade dramatically before rehashing, query times improve considerably immediately afterwards. This reflects the way in which cleary and layered deal with displacement information. Due to the highly set maximum load factor (0.95), with high probability elements with the same hash value become mixed, resulting in long lists of consecutive elements. Given that we consult an element at the i-th position in the hash table at which such a long list of elements is stored, layered and cleary have to consult the displacement information of i. While layered stores this information in two separate data structures, cleary may

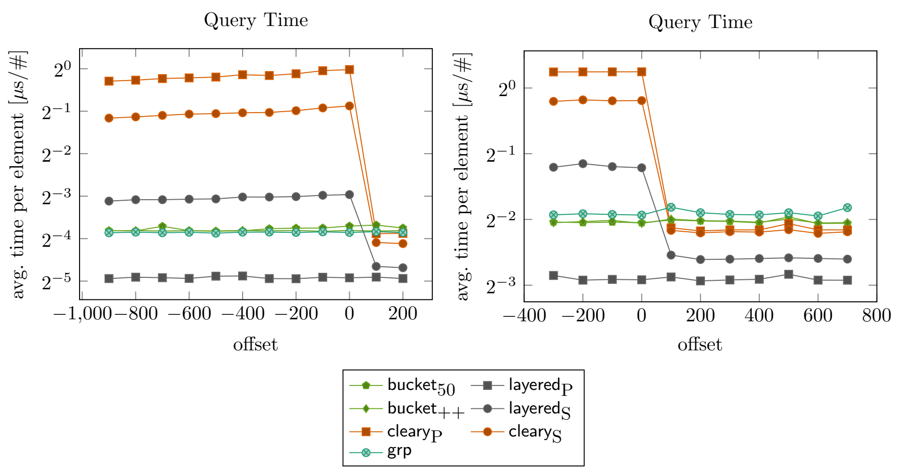

Fig. 3 Focus on the peaks during the query in Fig. 1 at which the hash tables have around 124498 $\approx 2^{17}$ or 15938343 $\approx 2^{24}$ elements stored, where we set the origin of the x-axis to 124498 and 15938343 stored elements, respectively, and the x-axis gives the offset with respect to that number of elements

have to scan all consecutive elements to the left of i. When inserting an element at the i-th position, linear probing scans to the right end of this list, requiring Bonsai tables to lookup displacements of all visited elements. Our new hash tables bucket and grp have smoother performance because of the way they handle rehashing: the contents of the i-th bucket is moved to the $2i$-th and $(2i + 1)$-th bucket after which the original bucket is freed.

Two Spikes in Detail In Figs. 1 and 2, the hash tables were filled $(3/2)^n 2^{10}$ elements for $n \geq 1$. However, this sampling is far too coarse to have a complete image of the query execution times. In fact, since a Bonsai hash table doubles its size on reaching the maximum load factor, such a doubling occurs within three consecutive data points. Chances are that one of our data points in the chosen sampling is near the point where the hash table needs to resize. In this case, due to the high load factor and the nature of linear probing, the query times become very deteriorated. After a rehashing the performance improves significantly, as we can observe in Fig. 3, which sets focus on the vicinity of two different spikes. There, each data point measures the time it takes to query a hash table for all stored elements (in a random order). The drop of the computation time after a rehashing is significant for cleary (note that the times scale is logarithmic!) and clearly visible for other hash tables with sparse layout like layered$_\mathrm{S}$.

5.2 Non-Compact Hash Tables

Figures 4 and 5 show an additional comparison with 8-bit values and two highly-optimized non-compact hash tables, namely:[6] Google's sparse hash table[2] google and Tessil's sparse map[7] tsl.

[6] Unfortunately, we could not evaluate other hash tables mentioned in the introduction. The implementation of [12] lacks resize capabilities (http://algo2.iti.kit.edu/sanders/programs/cuckoo/).

[7] https://github.com/Tessil/sparse-map.

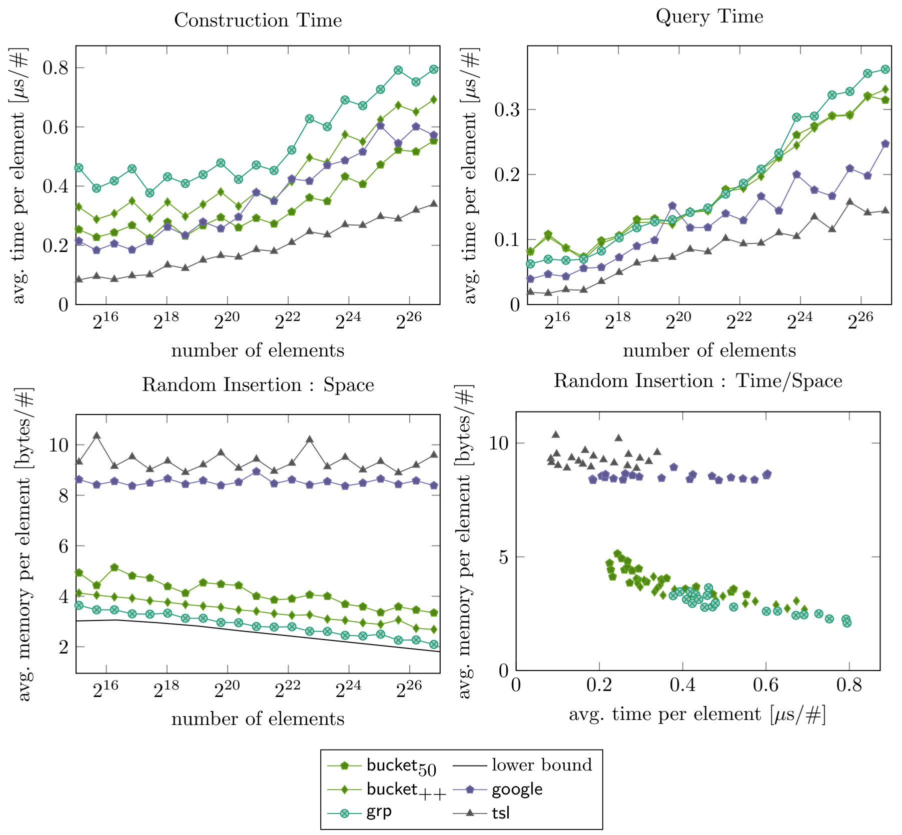

Fig. 4 Setting of Fig. 1 with randomly generated 8-bit values and 32-bit keys into a not-necessarily compact hash table (cf. Sect. 5.2)

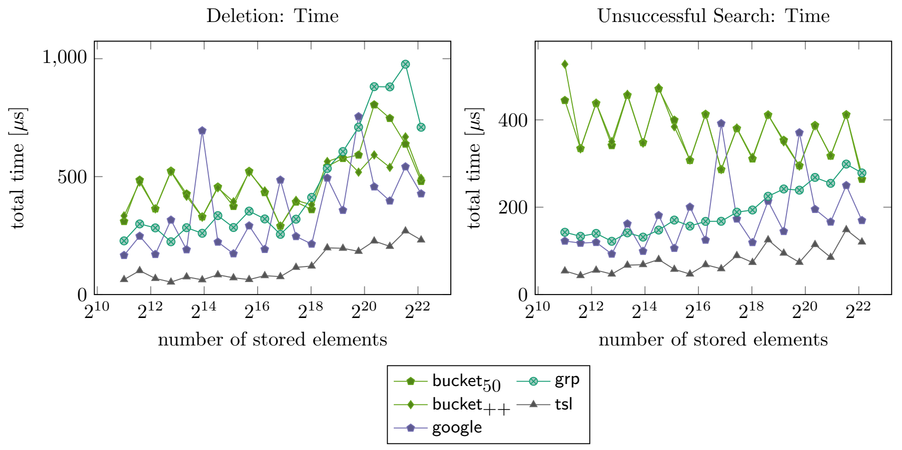

Fig. 5 Deletion and unsuccessful search benchmark of our proposed hash tables against the time-efficient approaches google and tsl. *Left*: Time for erasing 2^{10} random keys that are present in the hash tables. *Right*: Time for looking up 2^{10} random keys that are not present in the hash tables. In both figures, the number of elements (x-axis) is the (logarithmic) number of elements a hash table contains (cf. Sect. 5.2.2)

Both google and tsl are *sparse*, resolve collisions with quadratic probing, use the SplitMix hash function [30], and had maximum load factor set to 0.95. Section 6 provides a broader evaluation that includes the widely used STL implementation `unordered_map`, which we omitted here because it was always at least three times bigger than the other hash tables.

5.2.1 Insertions and Lookups

In Fig. 4, we conducted the same experiment as in Sect. 5.1.1 on the non-compact hash tables. While our implementations are slower for queries (grp is sometimes almost three times slower than tsl), they consistently use half of the memory, sometimes even less. bucket_{50} also shades google during construction.

5.2.2 Removing Elements

Figure 5 left shows the time to remove 2^{10} random elements. We used the hash tables created during the construction benchmark (Fig. 4) with 8-bit values. Bonsai tables are not included because their current implementations do not support element removal. Our hash tables are again consistently slower than tsl, while times for google fluctuate above and below ours. grp becomes slower than bucket on large instances. Experiments for unsuccessful searches (Fig. 5 right) show a similar pattern.

6 Extended Evaluation

For a clear visualization, we presented a careful selection of hash tables within a fixed setting in Sect. 5. There, we chose the quite high maximum load factor of 0.95 in the experiments resulting in unfavorably slow query times. We did so despite that a sparse layout seems more favorable for low load factors. To support our decision, we reran our evaluation with a maximum load factor of 0.5 in Fig. 6, where we observe, compared to Fig. 1, that sparse compact hash tables need considerably more space than with the maximum load factor of 0.95. We should add a direct comparison between max load factors of 0.5 and 0.95, perhaps with a table?

Next, we want to complement our selection of hash tables in Sect. 5 by providing benchmark results including the Bonsai table elias and the non-compact hash tables spp and std:

elias A variant of the dCHTs of Poyias et al. [23] provided by the tudocomp project. It partitions the displacement into integer arrays of length 1024, which are encoded with Elias-γ [10].

spp Gregory Popovitch's Sparsepp,[8] a derivate of Google's sparse hash table.

std The `unordered_map` implementation of the STL library `libstdc++`. This implementation uses separate chaining (closed addressing). We used the default maximum load factor 1.0, i.e., we resize the hash table after the number of stored elements exceeds the number of buckets.

[8] https://github.com/greg7mdp/sparsepp.

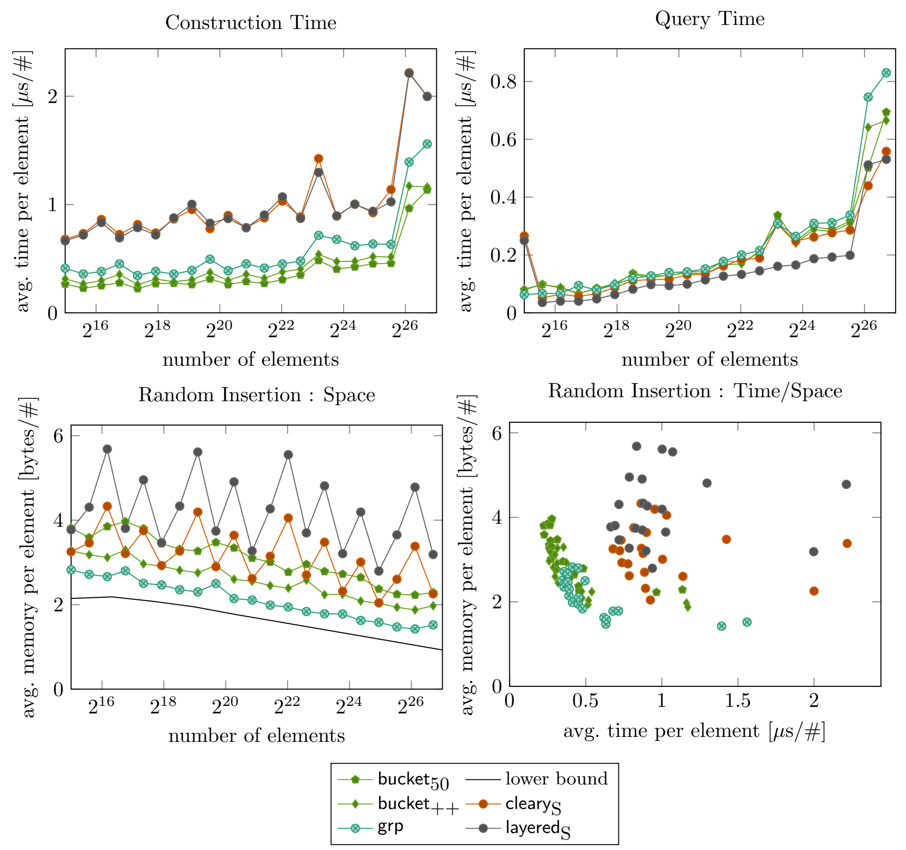

Fig. 6 Experiment of Fig. 1 with maximal load factor set to 50% (cf. Sect. 6)

We set the maximum load factor of spp to 0.95, but stuck to the default maximum load factor of 0.5 for elias due to time performance issues. We compared elias in Fig. 7, and std and spp in Fig. 8. We additionally conducted the deletion experiment of Sect. 5.2.2 in Fig. 9. Our observations are as follows:

elias Even with a load factor of only 0.5, elias has the worst time performance (cf. Fig. 7). elias_P is inferior to cleary_S both in time and space. When setting the load factor to 0.95, elias_S may have chances to become more lightweight than cleary_S, but is far away from being practical due to the additional time penalty of this high load factor.

std and **spp** The construction of std and spp becomes much slower than bucket_{50} for large instances, where bucket_{50} starts beating spp on all larger data points, and on some data points also std (cf. Fig. 8). The largest drawback of std is its highly memory consumption skyrocketing the space requirements. Nevertheless, for successful and unsuccessful queries, std is one of the best options, while spp stays at the center (cf. Fig. 9).

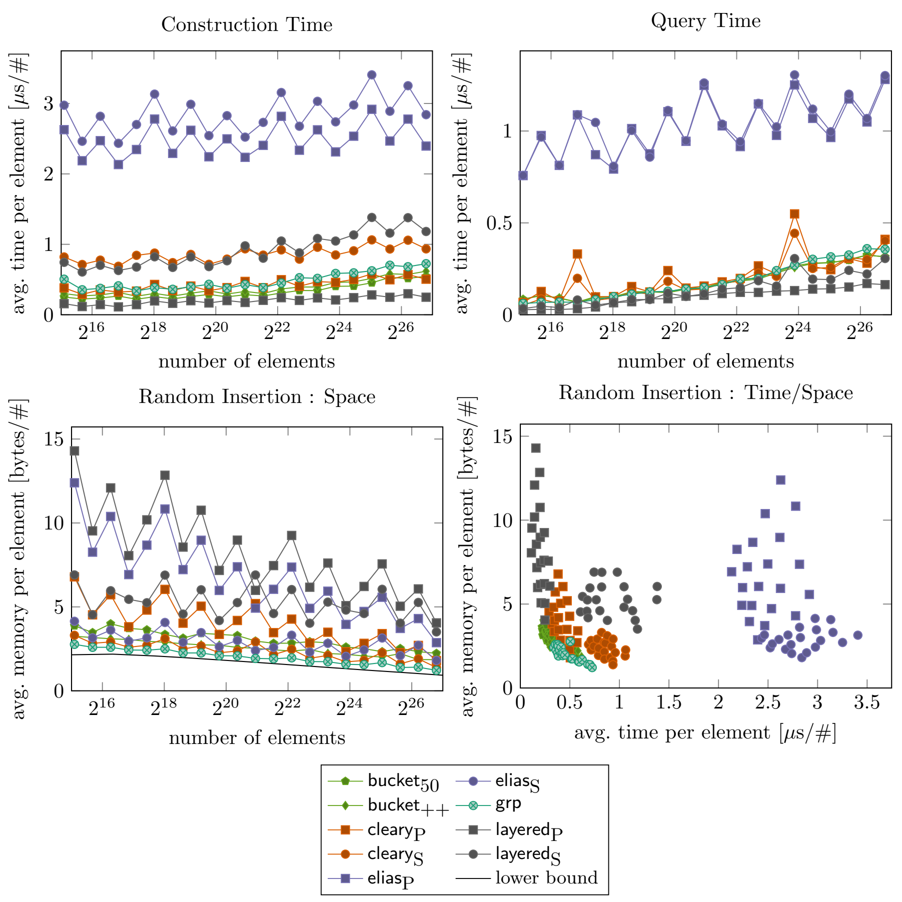

Fig. 7 Experiment of Fig. 1 with the elias$_P$ and elias$_S$ implementations described in Sect. 6

8-bit values. We also evaluated the compact hash tables with 8-bit values in Fig. 10. Compared to 1-bit values (Fig. 1), we only see a constant shift in the memory requirement per element for all hash tables.

7 SIMD Instructions

A drawback of our proposed hash tables is the slow lookup operation. Since quotients for large h become relatively small, we want to exploit the effect of scanning a bucket of quotients by broadword search or SIMD techniques in the following.

7.1 Broadword Search

For our implementation of bucket, we propose two different approaches in how to extract the quotients from its bit-compact byte array B. The first approach processes

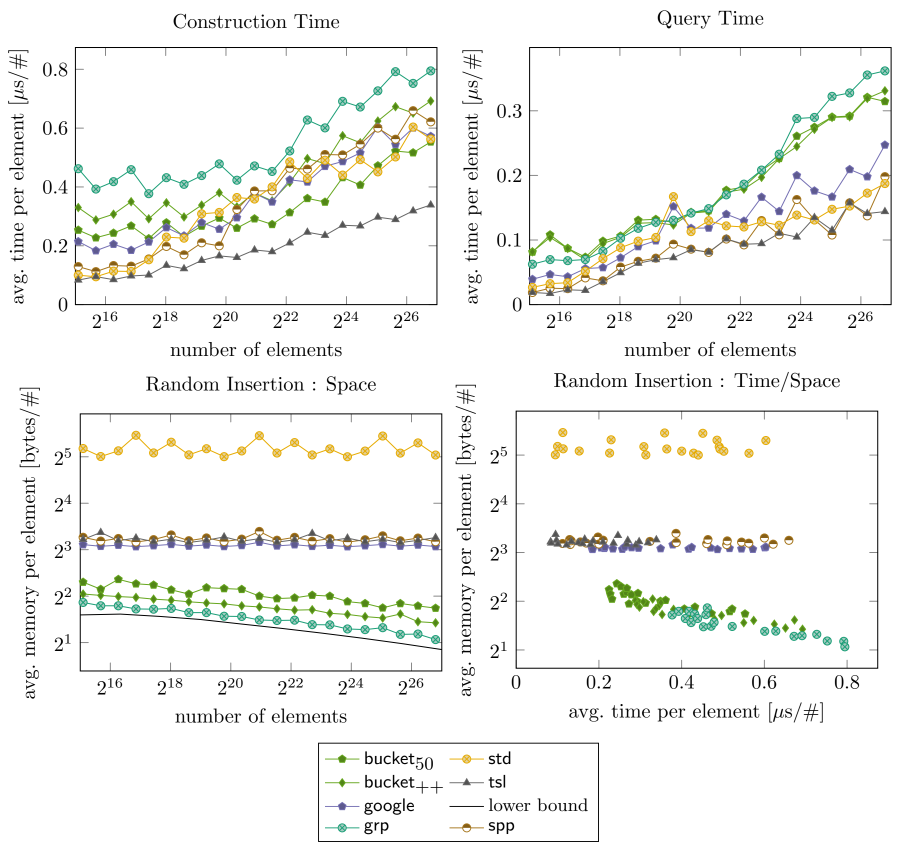

Fig. 8 Experiment of Fig. 1 with randomly generated 8-bit values and 32-bit keys evaluated on not-necessarily compact hash tables

B sequentially during the search of a quotient d. The second approach accelerates this search by storing $\lfloor w/q \rfloor$ times consecutively the bit representation of d in a w-bit integer p, where q is the quotient bit width, and compares the same number of quotients in B with $B[i..i+w-1] \otimes p$ for $i = cq\lfloor w/q \rfloor$ with an integer c, where we interpret B as a bit-string. Using bit-shift and bitwise AND operations, we can compute a bit-string C such that $C[j] = 1 \Leftrightarrow d = B[i+(j-1)q..i+jk-1]$ for $1 \leq j \leq \lfloor w/q \rfloor$, in $O(b_{\max} q/w)$ time by using bit parallelism [15, Sect. 7.1.3]. The experiments in Fig. 11 empirically show that the broadword search is faster than sequential search when the bucket size exceeds 64 items for a machine word size of 64 bits. We therefore benefit from employing the broadword search on buckets exceeding this size in our bucket implementation.[9]

[9] The measured sizes of our grp buckets are much smaller than this threshold.

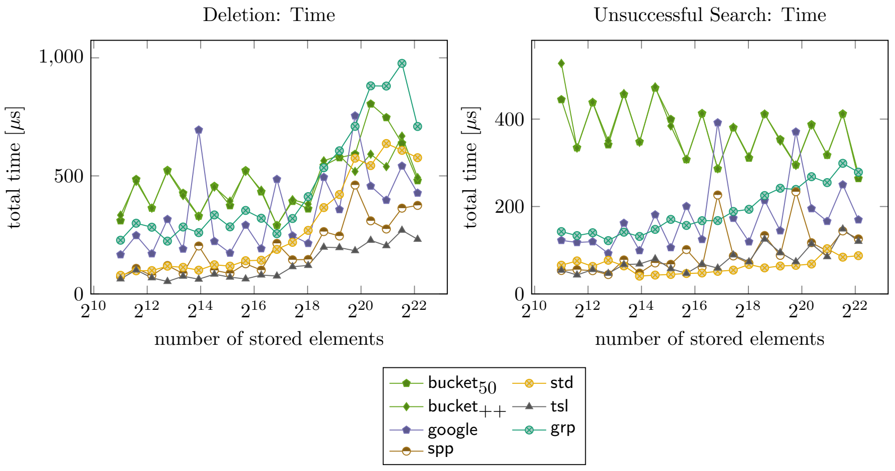

Fig. 9 Evaluation of Fig. 5 and Sect. 5.2.2 with more hash tables. *Left*: Time for erasing 2^{10} random keys that are present in the hash tables. *Right*: Time for looking up 2^{10} random keys that are not present in the hash tables. In both figures, the number of elements (x-axis) is the (logarithmic) number of elements a hash table contains (cf. Sect. 6)

7.2 AVX Representation

Another representation of the quotient bucket, called avx, applies SIMD instructions to speed up the search of a quotient in a large bucket. For that, it restricts the quotients to be quantized at 8 bits. We use the AVX2 instructions `_mm256_set1_epi`q and `_mm256_cmpeq_epi`q for loading a quotient with q bits and comparing this loaded value with the entries in the bucket, respectively. The `realloc` function for resizing a bucket cannot be used in conjunction with avx since the allocated memory for avx must be 32-byte aligned [17, Chapter 15].

7.2.1 Benchmarks

Following the setting of Sect. 5.1.1, we measured the time for insertions and lookups for our AVX variants in Fig. 12. Here, we added the variant plain of bucket using a fixed bit width for the quotients quantized by eight bits (i.e., plain only supports selecting the number of bytes for the quotient representation). plain serves as a baseline for the query time benchmarks of the AVX variants: While plain uses the bucketing technique, it does not need to store the quotients bit-compactly, and is therefore at least as fast as bucket. For that, it uses the trivial injective transform $f_h : x \mapsto (x, h(x) \bmod N)$ for a given hash function $h(x)$ such that d is the quotient and r is the assigned bucket index for a key x with $f_h(x) = (d, r)$. Effectively, plain therefore treats the keys as quotients, i.e., it does not apply compact hashing. Like plain, avx uses the same transform f_h. For the experiments, we use the SplitMix function for h as we did for the other non-compact hash tables. In Fig. 12, avx$_{50}$ is faster than plain$_{50}$ during the construction, and far superior when it comes to searching keys.

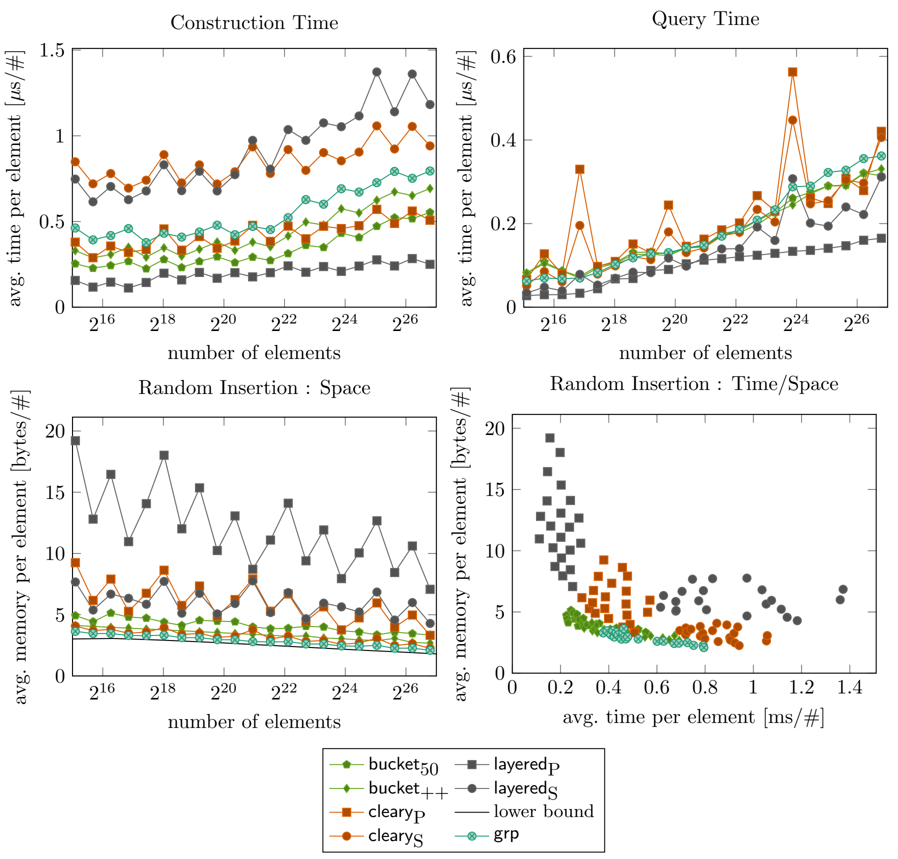

Fig. 10 Experiment of Fig. 1 with 8-bit values described in Sect. 6

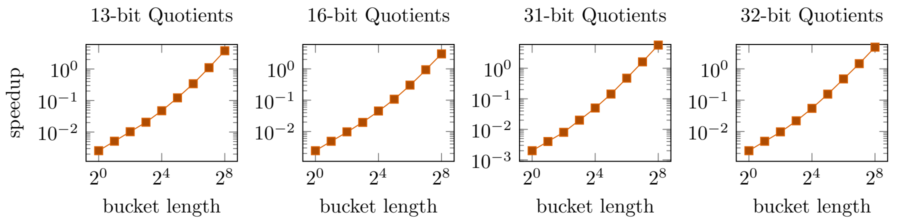

Fig. 11 Speedup for searching quotients of a specific bit width (13, 16, 31, and 32) in a bucket of length 2^x (x-axis) with our broadword search described in Sect. 7.1 in comparison with sequential quotient-by-quotient comparison approach. We take the cumulative time needed for retrieving all stored quotients of the respective bucket. The machine word size w is 64 bits. Although we would have expected that the search becomes slower for bit widths non-dividable by eight, the figures show the same characteristics for various bit widths

While the discrepancy in construction time between the incremental and the half increase policy is small for plain, the construction of avx_{++} takes considerably longer than avx_{50} since we cannot resort to the fast `realloc` for allocating *aligned* memory required for the SIMD operations.

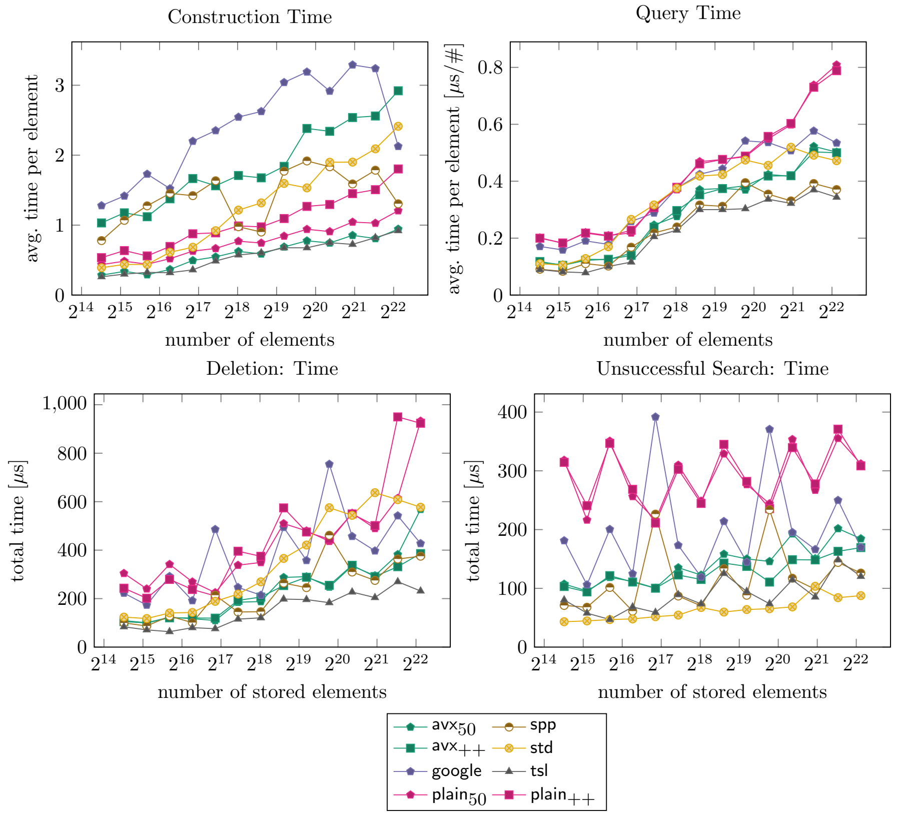

Fig. 12 Setting like in Fig. 10 (*Top*) and Fig. 5 (*Bottom*) with focus on our AVX variants (cf. Sect. 7). *Top Left*: Time for inserting randomly generated 8-bit values and 32-bit keys into a hash table. *Top Right:* Time for querying all inserted elements. *Bottom left*: Time for erasing 2^{10} random keys that are present in the hash tables. *Bottom right*: Time for looking up 2^{10} random keys that are not present in the hash tables

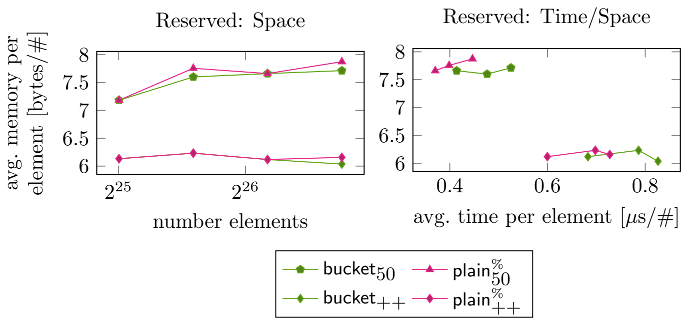

Fig. 13 Time for inserting $x \mapsto 2^{25} \cdot (3/2)^x$ random elements for integers $x \geq 0$. The hash tables are prepared to reserve $2^9 \cdot (3/2)^k$ buckets before the insertions start

We also evaluated the time for deletions and unsuccessful searches (bottom of Fig. 12). For deletions, only tsl is faster than our provided solutions. For unsuccessful searches, std and tsl are the fastest solutions, followed by spp and avx. plain is the slowest solution for all types of operations.

Combining avx_{50} with compact hashing can lead to a fast and memory-efficient hash table if there are precise lower bounds on the number of elements that need to be stored (cf. Fig. 12 for the time and plain_{50} in Fig. 13 for the space). In what follows, we study the memory savings when exploiting such a lower bound with plain applying compact hashing.

7.2.2 Reserved Space

Like in Fig. 10, we fill the hash tables with n random elements for increasing $n \geq 2^{16}$. However, this time we let the hash tables reserve 2^{16} buckets in advance. We added a percent sign in superscript to the plain hash tables that (a) apply compact hashing and (b) take (additionally) advantage of the fact that they only need to store quotients of at most 16 bits. The results are visualized in Fig. 13. Like in Fig. 12, a major boost for lookups can be observed if we exchange plain with avx (cf. Sect. 7.2),[10] which takes the same amount of space as plain.

8 Choice of $b_{\max}$ and Overflow Tables

In the theoretical description of bucket in Sect. 3.2, we choose $b_{\max} = z \lg u$, for some constant z that satisfied various constraints. Applying Theorem 1 directly, we can check that $c = 3.1$ suffices to ensure the point (a) in Sect. 3 with probability $1 - u^{-2}$, and $d = 55.18$ satisfies point (b) with the same probability. This would suggest $z = 55.18$ is appropriate, which would imply $b_{\max} \approx 3600$, assuming $\lg u = 64$. One can play with these numbers a bit more. For instance, changing (2/3) to (4/5) in point (b) and choosing $z = \max\{5c, d\}$ allows us to choose $z = 18.24$, and one could optimize this a bit further.

However, this analysis is very conservative for a variety of reasons. For example, while it may be likely that $h \log u$ insert operations cause some bucket to increase in size by nearly $c \log u$ (cf. point (a)), it is not likely that this bucket is one of those that was relatively large during the previous rehashing. Unfortunately, we are not aware of any precise analysis of this process (similar to the "balls into bins" process, studied extensively by Raab and Steger [24]). We have therefore performed two kinds of simulations.

Firstly, we have simulated $n = 2^{28}$ insertions of keys into a hash table, but choosing $b_{\max}$ based upon the number of buckets currently being used. Specifically, we chose $b_{\max} = \lfloor c \lg h \rfloor$ for $c = 0.5, 1, 2$ and 4. Thus, each time there is a rehashing, $b_{\max}$ likely increases as h doubles. In this test, we focus primarily on how the average bucket fullness (average bucket size divided by $b_{\max}$) evolves with increasing number

[10] avx is not shown in Fig. 13 since our memory allocation counting library does not count aligned allocations needed for avx.

of insertions; we measure the average bucket size just before a rehashing. Studying the bucket fullness in this way should give a couple of useful pointers:

- In case that the fullness remains roughly steady as n grows, such an observation would suggest that rehashing has an acceptable amortized cost (because it shows that the keys roughly double between rehashes). It also shows that the bucket space overhead per key is constant as n rises.
- We note that the lower the bucket fullness, the greater the number of buckets relative to the number of keys, and the greater the $O(1)$ space cost per key arising from the bucket overheads.

We argue that this kind of test would give some indications as to how a given choice of $b_{\max}$ would perform as n increases. We noticed that, for example, with $c = 4$, the average bucket fullness was varying between 50% and 56%. This gives us confidence that choosing $b_{\max} = 256$ will continue to perform fairly well – in terms of the theoretical considerations underlying (a) and (b) – even as the number of buckets approaches 2^{64}.

Another simulation we performed is to fix different values of $b_{\max}$ (keeping this fixed throughout the series of insertions), and perform a varying number of insertions into the hash table. This time, however, we measure the bucket fullness at the end of the series of insertions, thus possibly providing a slightly less "worst-case" measure of fullness. Fig. 14 shows overflow tables, which have not been introduced up till now. Maybe we should use percentages in the y-axis to visualize how close we are away from 100%? This test is shown in Fig. 14, where we focus on the line of '0.0'; the meaning of the other lines will be explained later. In this figure, '0.0' depicts the average bucket size for a fixed $b_{\max}$ when scaling the number of elements n. We observe that fullness decreases as n increases, for any given $b_{\max}$. For smaller values of $b_{\max}$ (e.g., 32) the fullness is quite low even at $n \sim 2^{26}$, and the drop in the fullness is quite considerable. Although we see a drop even for $b_{\max} = 255$, the drop is only about 10% from $n = 2^{15}$ to 2^{26}. We can draw the conclusion that the overflow tables help significantly in keeping the buckets fuller when choosing a small $b_{\max}$. In the setting of hash ID maps, overflow tables are interesting even for large $b_{\max}$, as they help us to deter a rehashing.

Overflow Tables One idea to improve the fullness of the buckets, and to ensure greater robustness of this approach, is to use *overflow* tables. The starting point of this analysis is to note that when inserting keys into buckets, it is likely that some buckets will grow significantly larger than average. However, this number is small: applying Theorem 1 (more specifically, a simplification thereof, see [25, Equation (4.12)]), we see that when inserting n keys into a collection of initially empty buckets, the probability that a bucket exceeds the expected size $\bar{s} = n/h$ by more than $O(\sqrt{\bar{s}} \lg s)$ decreases as $1/\bar{s}^{O(1)}$. This suggests the idea of putting a limit on the bucket sizes of the expected size plus a modest amount, and putting all keys that would end up in buckets whose sizes are already at their limit $b_{\max}$ into an *overflow table*. The goal is to let only very few keys go into this overflow table, while letting all bucket sizes be quite close to $b_{\max}$. This benefits not only the overall space usage, but also the constant factor in the $O(n)$ bound on the range ρ of values, should we use this approach to implement a hash ID

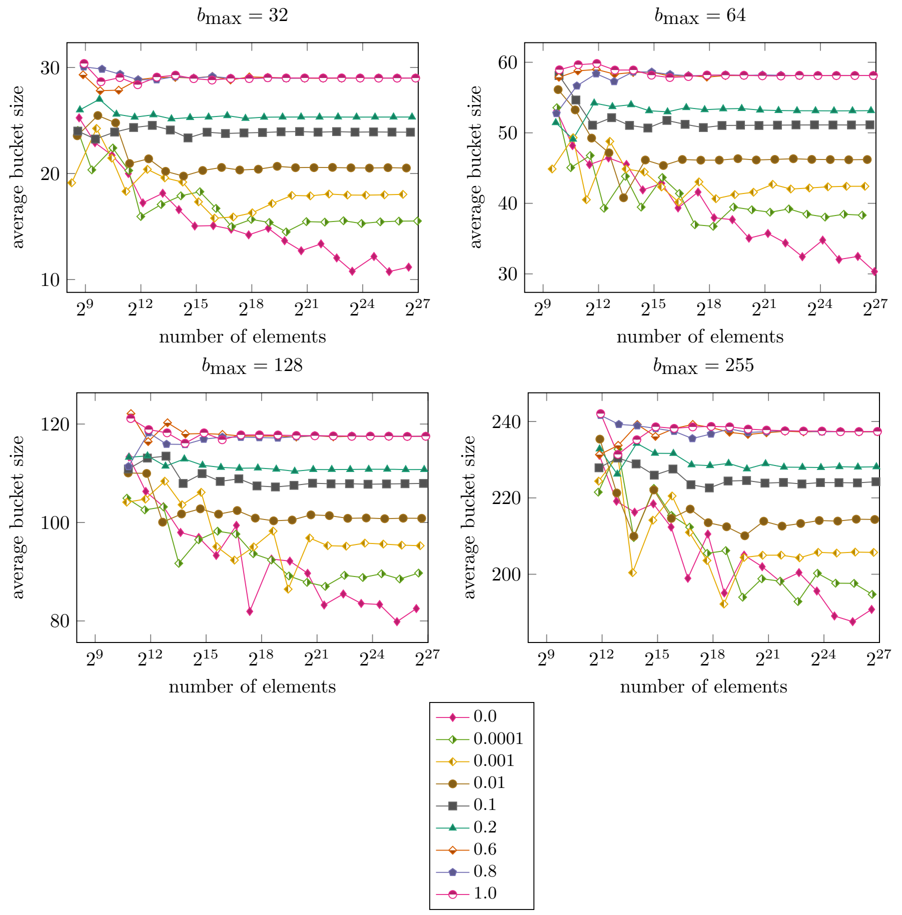

Fig. 14 Influence of an overflow table on the average bucket sizes (cf. Sect. 8). In this setting, an overflow table has a maximum size that is the product of the number of buckets of bucket and a fractional number o_{frac}, which we set between 0 (i.e., no overflow table) and 1 (the overflow table can contain at most as many elements as bucket has buckets). The benchmark shows the average bucket sizes (y-axis) with increasing number of elements (x-axis). Each plot uses a different $b_{\max}$. Each curve represents a different value o_{frac} (see the legend on the right)

map, and finally also ensures that a given value of $b_{\max}$ (such as 255) continues to work well for much larger values of n.

We have implemented this as follows: Whenever we want to insert an element in a full bucket (resp. full group-bucket in grp), we put this element in the overflow table and mark the bucket in a bit-string of length b indicating that whenever we want to search for an element in this bucket, we also have to consult the overflow table. The overflow table itself has a maximum size.[11] Whenever this size is reached, we rebuild the entire data structure. We evaluated the influence of the overflow table in Fig. 14, where we set the maximum size of the overflow table in relation to the number of

[11] Our implementation with overflow tables is only used in Figs. 14 and 15.

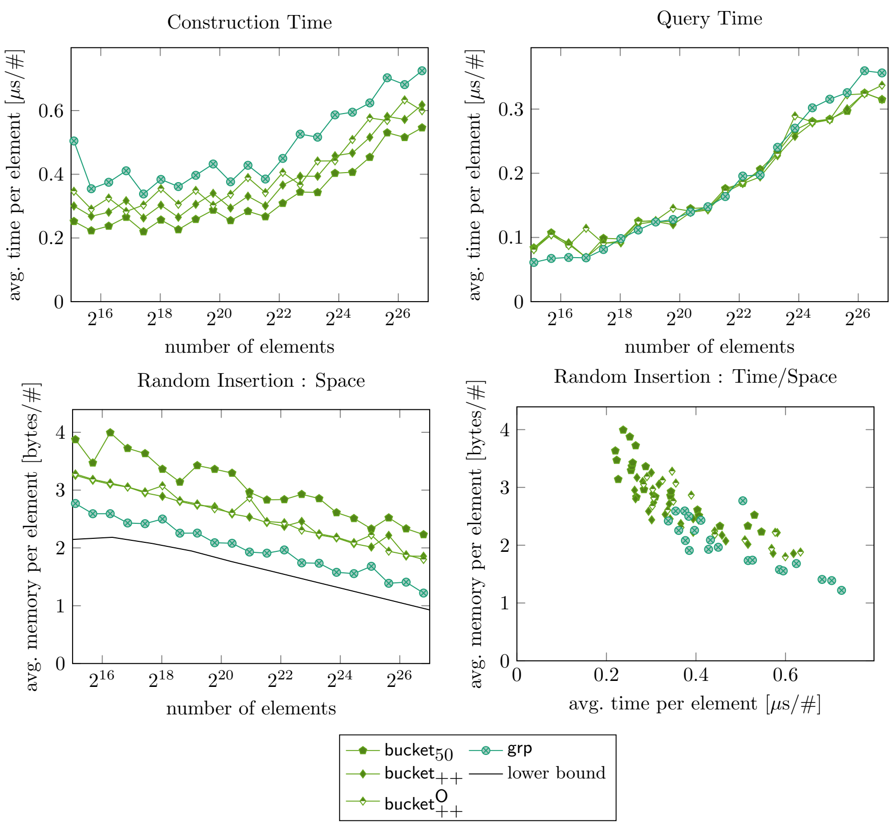

Fig. 15 Setting like in Fig. 1 with 1-bit values featuring the variant bucket$^{O}_{++}$ equipped with an overflow hash table (cf. Sect. 8)

buckets with a fractional number o_{frac}. We equipped bucket$_{++}$ with layered$_{P}$ as an overflow table with $o_{\text{frac}} = 0.6$, and call the resulting approach bucket$^{O}_{++}$. (Doing the same with grp only lead to worse performance both in time and space since the group-bucket sizes in the experiments already closely approach $b_{\max}$.) As we see, even with very small overflow tables, the bucket fullness stays steady as n increases, even for smaller values of $b_{\max}$. However, we see in Fig. 15 that the time and space bounds for the construction are roughly the same as without the overflow table. Note that in these tests, the information-theoretic lower bound per key is relatively high, so the bucket space overhead is anyway not a major part of the space usage.

We draw the conclusion that hash tables with overflow tables are performant even with small values of $b_{\max}$. Hence, with overflow tables we do not need to tweak the $b_{\max}$ parameter dependent on the input data set, which can be much larger than we evaluated here.

9 Conclusions and Future Work

We have suggested a simple approach for implementing compact hash tables, and our implementations show positive results. The experiments reveal that our hash tables grp and bucket use the least space of all tested approaches. Moreover, their time and space requirements scale smoothly with the problem size, unlike the other compact (i.e., Bonsai) tables tested, whose performance is periodically adversely affected by rehashing. The new tables are also faster to construct than other hash tables with similar memory requirements—only hash tables with much higher memory requirements have faster construction times.

The main weakness of grp and bucket is the slower lookup time, both for successful and unsuccessful searches. This is the price we pay for low space usage, which is achieved by keeping all buckets of bucket and grp as close to $b_{\max}$ as possible, resulting in long scan times.

There are numerous avenues for future work. A more refined analytical treatment of the space usage of the hash table (whose rebuilding is triggered by the parameter $b_{\max}$), and the expected frequency of rehashes, as well as a better understanding of the use of overflow hash tables, would be welcome. In the experiments, the measured memory is the number of allocated bytes. The resident set sizes of our hash tables may differ significantly to this quantity, as we allocate small sizes of A dedicated memory manager can reduce this space overhead.

The AVX2 SIMD instruction set provides a major performance boost over earlier instruction sets like SSE — with benchmarks for comparing strings indicating a speed boost of more than 50% for long strings.[12] We wonder whether we can gain an even steeper acceleration in our hash tables when working with the newer AVX256 instruction set. The current implementation allows only for fixed quotients of bit widths 8, 16, 32, and 64. It is possible to also support the bit widths 1, 2, and 4. Larger bit widths like 24 need more operations and seem therefore unfavorable. However, to support arbitrary quotient widths, we can split a quotient into two bit chunks, where the former has a width equal to one of those fixed bit widths, say q', while the latter stores the remaining bits. We then use the former exactly as in the current SIMD implementation such that we also check only the first q' bits for a queried quotient. This can give false positives, so we also have to check the remaining part, which we then do without SIMD instructions.

Thinking about different hash table layout could also spawn future research. For instance, we could think about linearizing the buckets into a single array like `ska::bytell_hash_map`.[13] This approach stores pointers of a fixed bit width to the next element in the bucket to simulate the linked list of the separate chaining resolution scheme. To be useful, this fixed bit width has to be small, consequently supporting only small values for $b_{\max}$. Complemented with an efficient overflow table, this table could have faster query times than bucket while using not considerably more space.

[12] https://github.com/koeppl/packed_string.

[13] https://probablydance.com/2018/05/28/a-new-fast-hash-table-in-response-to-googles-new-fast-hash-table/.

Acknowledgements We thank Marvin Löbel for the implementations of the Bonsai tables with the additional support of a sparse table layout. This research was funded by JSPS KAKENHI with Grant Nos. JP21H05847 and JP21K17701.

References

1. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: constant worst-case operations with a succinct representation. In: Proceedings of FOCS, pp. 787–796 (2010)
2. Blandford, D.K., Blelloch, G.E.: Compact representations of ordered sets. In: Proceedings of SODA, pp. 11–19 (2004)
3. Blandford, D.K., Blelloch, G.E.: Compact dictionaries for variable-length keys and data with applications. ACM Trans. Algorithms **4**(2), 17:1-17:25 (2008)
4. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM J. Comput. **28**(5), 1627–1640 (1999)
5. Carlini, P., Edwards, P., Gregor, D., Kosnik, B., Matani, D., Merrill, J., Mitchell, M., Myers, N., Natter, F., Olsson, S., Rus, S., Tavory, A., Wakely J.: The GNU C++ Library Manual. FSF, Johannes Singler (2018)
6. Cleary, J.G.: Compact hash tables using bidirectional linear probing. IEEE Trans. Comput. **33**(9), 828–834 (1984)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
8. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Softw. Pract. Exper. **23**(3), 277–291 (1993)
9. Demaine, E.D., Meyer auf der Heide, F., Pagh, R., Patrascu, M.: De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In: Proceedings of LATIN, volume 3887 of LNCS, pp. 349–361 (2006)
10. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974)
11. Fischer, J., Köppl, D.: Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In: Proceedings of SPIRE, volume 10508 of LNCS, pp. 191–207 (2017)
12. Fotakis, D., Pagh, R., Sanders, P., Spirakis, P.G.: Space efficient hash tables with worst case constant access time. Theory Comput. Syst. **38**(2), 229–248 (2005)
13. Jansson, J., Sadakane, K., Sung, W.-K.: CRAM: compressed random access memory. In: Proceedings of ICALP, volume 7391 of LNCS, pp. 510–521 (2012)
14. Kanda, S., Köppl, D., Tabei, Y., Morita, K., Fuketa, M.: Dynamic path-decomposed tries. ACM JEA **25**(1), 1.13:2-1.13:28 (2020)
15. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, 12th edn. Addison-Wesley, Boston (2009)
16. Köppl, D., Puglisi, S.J., Raman, R.: Fast and simple compact hashing via bucketing. In: Proceedings of SEA, volume 160 of LIPIcs, pp. 7:1–7:14 (2020)
17. Kukunas, J.: Power and Performance: Software Analysis and Optimization. Elsevier, Amsterdam (2015)
18. Maier, T., Sanders, P., Walzer, S.: Dynamic space efficient hashing. Algorithmica **81**(8), 3162–3185 (2019)
19. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)

20. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Comput. **31**(2), 353–363 (2001)
21. Peterson, W.W.: Addressing for random-access storage. IBM J. Res. Dev. **1**(2), 130–146 (1957)
22. Poyias, A., Puglisi, S.J., Raman, R.: Compact dynamic rewritable (CDRW) arrays. In: Proceedings of ALENEX, pp. 109–119 (2017)
23. Poyias, A., Puglisi, S.J., Raman, R.: m-Bonsai: A practical compact dynamic trie. Int. J. Found. Comput. Sci. **29**(8), 1257–1278 (2018)
24. Raab, M., Steger, A.: "Balls into Bins"-A simple and tight analysis. In: Proceedings of RANDOM, volume 1518 of LNCS, pp. 159–170 (1998)
25. Raghavan, P., Motwani, R.: Randomized Algorithms. Cambridge International Series on Parallel Computation, Cambridge University Press, Cambridge (1995)
26. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Proceedings of ICALP, volume 2719 of LNCS, pp. 357–368 (2003)
27. Robinson, J.T., Devarakonda, M.V.: Data cache management using frequency-based replacement. In: Proceedings of SIGMETRICS, pp. 134–142 (1990)
28. Ross, K.A.: Efficient hash probes on modern processors. In: Proceedings of ICDE, pp. 1297–1301 (2007)
29. Schlegel, B., Gemulla, R., Lehner, W.: Memory-efficient frequent-itemset mining. In: Proceedings of EDBT, pp. 461–472 (2011)
30. Steele, G.L., Jr., Lea, D., Flood, C.H.: Fast splittable pseudorandom number generators. In: Proceedings of OOPSLA, pp. 453–472 (2014)
31. Thorup, M.: Fast and powerful hashing using tabulation. Commun. ACM **60**(7), 94–101 (2017)
32. Vigna,S.: Broadword implementation of rank/select queries. In: Proceedings of WEA, volume 5038 of LNCS, pp. 154–168 (2008)
33. Zhou, D., Andersen, D.G., Kaminsky, M.: Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In: Proceedings of SEA, volume 7933 of LNCS, pp. 151–163 (2013)