Fast and Simple Compact Hashing via Bucketing

Dominik Köppl
Department of Informatics, Kyushu University, Japan Society for Promotion of Science (JSPS), Fukuoka, Japan
https://dkppl.de/
dominik.koeppl@inf.kyushu-u.ac.jp

Simon J. Puglisi
Helsinki Institute for Information Technology, Espoo, Finland
Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Rajeev Raman
School of Informatics, University of Leicester, United Kingdom
r.raman@leicester.ac.uk

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).

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 close to $\mathcal{B}(u,n)$ bits. One of our approaches is suitable for use as a compact Hash ID map.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases compact hashing, hash table, separate chaining

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.7

Related Version http://arxiv.org/abs/1905.00163

Supplementary Material Implementations are available at https://github.com/koeppl/separate_chaining. Our benchmarks for the operations insert, lookup, and delete are available at https://github.com/koeppl/hashbench.

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number JP18F18120.
Simon J. Puglisi: Academy of Finland Grant 319454.

Acknowledgements We thank Marvin Löbel for the implementations of the Bonsai tables with the additional support of a sparse table layout.

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* $U = [u]$[1] and the value y is from $[2^v]$. 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.

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 $\mathcal{B}(u, n) = \lg \binom{u}{n} = n \lg u - n \lg n + O(n)$ bits (in what follows we abbreviate $\mathcal{B}(u, n)$ by $\mathcal{B}$). The lower bound on the space for the key-value pairs in S is thus $\mathcal{B} + nv$ bits. 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, 15], 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 [10, 12]) is prohibitively large.

Building on earlier work on succinct *static* dictionaries [4, 14], succinct dynamic dictionaries were proposed by Raman and Rao [18] 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[2]. A slightly different data structure using $O(\mathcal{B} + nv)$ bits of space was discussed in [8]. 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.

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. [16] 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 [16] in fact takes $\Theta(n \lg^{(5)} n)$ bits[3] more space than the CHT), and particularly as ϵ approaches 0, the practical performance of these approaches deteriorates significantly.

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

[1] For non-negative integers i, $[i] = \{0, 1, \ldots, i-1\}$.
[2] These two results differ regarding the model of dynamic memory allocation used.
[3] Throughout lg denotes the logarithm to base two with $\lg^{(1)} n = \lg n$ and $\lg^{(i)} = \lg(\lg^{(i-1)} n)$ for $i > 1$.

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 not "much" more than 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 [16, 11] (and potentially other compact data structures), LRU cache management [19], and naming in string processing [13]. Implicit in the work of Darragh et al. [7] 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. [16] 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 paper 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 [18], as well as earlier CHTs.

Our Contributions. We propose simple and practical CHTs with:

- $\mathcal{B} + nv + O(n \lg\lg u)$ bits, performing insert and lookup in $O(\lg u)$ expected time. 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 the same operation complexities as above.
- $\mathcal{B} + nv + O(n)$ bits, performing insert in $O(\lg u)$ worst-case time and lookup in $O(1)$ expected time.

Despite their poor asymptotic complexities, these approaches are simple and designed for practical performance, which we verified in the evaluation described in Section 4. In this evaluation, we consider two distinct scenarios:

- $\lg(u/n)$ and the value bit width v are both small (using only a few bits), motivated by the CDRW-array [16], which compactly stores a dynamic array A, most of whose entries can be stored in very few bits. For every integer $v \geq 1$, the CDRW-array takes all indices in A containing v-bit values, and stores the key-value pair $(i, A[i])$ in a CHT for that integer 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. 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 two 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 [20] to accelerate searches in a bucket.

Discussion. We briefly summarize how our approaches differ from other compact hash tables. 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, 16]. 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[4]. 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 low load factors with a moderate space overhead, and works well with compact hashing [9, 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; all keys are simply stored in their buckets. The usual representation of a bucket is via *chaining*, i.e., storing a bucket as a linked list, as in the `unordered_map` of the C++ standard library `libstdc++` [5, Sect. 22.1.2.1.2]. 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 [18]) 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.

Paper Overview. We begin with a theoretical description of our algorithms in Section 2. In Section 3 we describe the implementations, which depart from theory in some ways. Section 4 describes the experimental evaluation and Section 5 concludes and states directions for further work.

2 Compact Hashing via Bucketing: a Theoretical View

In this section, we outline the approach that we take from a theoretical perspective. This section assumes the *transdichotomous model* with word size $w = \lg u$ bits. For simplicity we consider insertion-only hash tables. Let N be a parameter with $N \leq n < 2N$ (we discuss what happens when this is violated in Section 2.3), and b the number of buckets of our hash table. We start with an invertible function $f : [u] \to [u]$, and use f to map key-value pairs to the b buckets. We assume that b, the size of the universe u, and N are powers of two.

Now, given a key-value pair $\langle x, y\rangle$, we compute $f(x)$, and assign $\langle x, y\rangle$ to the bucket $r = f(x) \bmod b$. In the bucket r, the key x is represented by its quotient value $q = f(x) \operatorname{div} b$; observe that the key x can be recovered as $f^{-1}(qb + r)$ and that the quotient value takes $\lg u - \lg b$ bits. We analyze the hash tables under the assumption of uniform and fully independent hashing. The key design decisions are the representation of the buckets and the choice of b, both of which we discuss in Section 2.3.

[4] `https://github.com/sparsehash/sparsehash`

2.1 Simple Compact Hashing

We start with the simple variant, which maintains $b = O(N/\lg u)$ buckets. The buckets are represented by an array of b pointers, each pointing to an array of w-bit words that stores the key-value pairs of the respective bucket. The overhead per bucket is $O(w)$ bits; summed over all buckets this overhead is $O(n)$ bits. Since each key is represented by a quotient using $\lg u - \lg b = \lg(u/N) + \lg\lg u$ bits, 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. To perform a lookup, an entire bucket is scanned. 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. A standard argument [17] shows that the maximum bucket size $b_{\max}$ is bounded by $O(\lg u)$ with probability $1 - 1/u^{O(1)}$. The running times of insert and lookup are therefore $O(\lg u)$ with high probability.

To use this as a compact hash ID map, we fix $b_{\max} = c \lg u$ for a constant $c > 0$ chosen large enough such that the probability of every bucket exceeding $b_{\max}$ is at most $(1/u)^2$ [17]. Suppose that a key[5] is stored at the j-th position of the i-th bucket and $j < b_{\max}$. Then the ID associated with this key is just $i \cdot b_{\max} + j$. We discuss the length of the "lifetime" of this data structure below.

2.2 Space-Efficient Compact Hashing

Our space-efficient variant maintains $b = N$ *sub-buckets*. Each sub-bucket is not represented individually, but $\lg u$ sub-buckets are *grouped* together in a bucket, which we call *group-bucket* in the following to avoid confusion. A group-bucket, with a total of s key-value pairs in it, is represented as an array of size s. In this array, all keys hashed to the same sub-bucket are stored in a consecutive range, and a bit-string of length $\lg u + s$ demarcates the sub-bucket boundaries by concatenating the sizes of the sub-buckets, written in unary, for example. The overheads of this approach (including the bit-string) are at most $O(n)$ bits. However, the quotients stored in each sub-bucket now only take $\lg u - \lg N = \lg(u/n) + O(1)$ bits. Thus, the overall space usage is $\mathcal{B} + nv + O(n)$ bits. We now consider the time for an operation. Using the same argument as in the previous section (Section 2.1), we can see that the size s of each group-bucket is at most $O(\lg u)$ with high probability, and the bit-string is therefore of length $O(\lg u)$ bits with high probability as well, which means that the range in the group-bucket corresponding to a sub-bucket can be located with a broadword select operation in $O(1)$ time with high probability. Since the expected sub-bucket size is $O(1)$, 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 group-bucket each time, and takes $O(\lg u)$ time with high probability as a result. (Since each insert causes a group-bucket to be re-written, this approach is not suitable for use as a hash ID map.)

2.3 Rebuilding

Insertions will cause the invariant $N \le n < 2N$ to be violated. A standard approach would double N and rehash. From a coding perspective, however, it can speed up search if it is known that all buckets have a fixed maximum size $b_{\max}$ that stays unchanged throughout the execution of the algorithm – for example, a loop that searches through a bucket can be unrolled. Asymptotically, $b_{\max}$ cannot be a constant. Even if we use N buckets, it is well known that some bucket will have $\Theta(\log n / \log\log n)$ keys [17, Thm. 1].

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

Despite this, we set $b_{\max}$ to be a fixed parameter, and let b be the current number of buckets. As soon as any bucket exceeds $b_{\max}$ in size we rebuild the data structure by doubling b. When we do this, the keys in the old bucket i are distributed at random into new buckets $2i$ and $2i+1$ (details in the next section). Once the rebuilding is complete, the insertions can resume. We say that a value of $b_{\max}$ is *stable* for a particular range of n, if the space between rebuildings is roughly $\Theta(n)$.

The stability can be improved through the use of *overflow* tables: when a new key is inserted and is hashed to a bucket of size $b_{\max}$, we put the key-value pair into a single global hash table, the *overflow table* (obviously, when searching, we may need to search the overflow table as well). The benefits of using an overflow table are as follows.

If we throw $n \ln n$ balls into n buckets randomly, the expected maximum bucket size is approximately $e \ln n$ [17]. Heuristically, this would suggest $b_{\max}$ should be set to be three times the average bucket size to ensure stability. However, the probability that a bucket exceeds $\ln n + O(\sqrt{\lg n})$ balls is $1/(\log n)^{O(1)}$. Thus, we can set $b_{\max}$ to much closer to the average bucket size. By doing so, the number of keys that go to the overflow table would only be $n/(\log n)^{O(1)}$, and the overflow table would therefore not be a major drag on either the time or the space performance of the hash table. This also suggests that with the appropriate parameter settings, a given constant $b_{\max}$ will continue to be stable for a *much* larger value of n (possibly quadratically bigger).

In addition, an overflow table can reduce ρ for compact hash ID maps. Keys inserted into the overflow table are given consecutive IDs beginning at $b \cdot b_{\max}$ and stored together with their IDs in an overflow table. If we reduce $b_{\max}$ by a constant factor for a given n, while placing a few keys into the overflow table, ρ will also be reduced by a constant factor.

3 Implementation

We implemented the simple approach and the space-efficient approach described in Sections 2.1 and 2.2, and call the implementations cht and grp, respectively. Each of the two implementations maintains b buckets, where b is a power of two. Following the discussion of different invertible functions given in [9, Sect. 3.2], we select a fixed multiplicative function $f : [u] \to [u]$ for the experiments. We use the last $\lg b$ bits of its return value for the index of the assigned bucket and the other bits for the quotient.

3.1 Maximum Bucket Size and Rehashing

In both cases we choose $b_{\max} = 255$. Our simulations suggest that $b_{\max} = 63$ is almost adequate for n up to 10^9. Intuitively, for stability $b_{\max}$ should be logarithmically dependent on n, so it would appear that $b_{\max} = 255$ should ensure stability for n up to 10^{30}, even without the use of overflow tables. The larger choice of $b_{\max}$ also helps to reduce the per-bucket overhead (at least 17 bytes, see Section 3.2).

When we try to insert an element into a bucket of size $b_{\max}$, we create a new hash table with twice the number of buckets and move the elements from the old table to the new one, bucket by bucket. After a bucket of the old table becomes empty, we can free up its memory, thus significantly reducing the peak memory during resizing. Note that the rebuilding is particularly efficient since the contents of bucket i in the old data structure is distributed between buckets $2i$ and $2i+1$ in the new data structure (the last $1 + \lg b$ bits of a hash value now determine the bucket).

The main additional parameter in grp versus cht is the number of sub-buckets in a group-bucket. We also deviate from theory with respect to the number of sub-buckets assigned to a group-bucket, which we call 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 costs us $m+s+sv+sq$ bits on average, where $q = \lg u - \lg b$ 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 s. 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 $m = 64$.

3.2 Buckets

We represent a bucket (resp. group-bucket for grp) as a quotient and a value bucket. 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.[6]

4 Experiments

We implemented the approaches described in Section 2.1 and Section 2.2 in C++17, and refer to them cht and grp, respectively. We subscript cht with '++' or '50' to indicate whether the hash table resizes a bucket by, respectively, one element or by 50% (cf. Section 3.2). 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++-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`.

4.1 Bonsai Tables

We compare our implementations cht and grp with practical implementations of compact hash tables implemented in the tudocomp project[7]. 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. [16]: layered stores the displacement information in

[6] 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 than merely appending an element to a bucket of cht. We are unsure whether a different resize policy pays off.

[7] `https://github.com/tudocomp/compact_sparse_hash`

two associative array data structures. The first is an array storing 4-bit integers, and the second is an `unordered_map` (the C++ STL hash table implementation) for displacements larger than 4 bits. 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.

4.1.1 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`. Figure 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.

Time. Considering construction time, $\mathsf{layered}_{\mathrm{P}}$ and cht_{50} are the fastest options, while the sparse Bonsai tables $\mathsf{layered}_{\mathrm{S}}$ and $\mathsf{cleary}_{\mathrm{S}}$ are the slowest options. Between them are cht_{++}, $\mathsf{cleary}_{\mathrm{P}}$ and grp. Considering the query time for large instances, the difference to the construction time is that cht and grp are here slower than all variants of cleary and layered, where again $\mathsf{layered}_{\mathrm{P}}$ 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). $\mathsf{layered}_{\mathrm{P}}$ is again generally the fastest (though with somewhat less consistent performance). grp is faster than cht for unsuccessful searches. 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 cht.

Space. grp, followed by cht_{++} and then by cht_{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 cht, which helped grp to delay rehashings. $\mathsf{cleary}_{\mathrm{S}}$ beats on some instances cht_{++} (but not grp) while having significantly slower construction times than cht or grp. However, the relatively large memory reallocation during a rehashing prevents the memory requirement of $\mathsf{cleary}_{\mathrm{S}}$ to stay below cht_{++} for a longer time.

4.1.2 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 phenomena, recall that the Bonsai tables use linear probing with a maximum load factor of 0.95. As the Bonsai tables approach fullness, insertion and query times deteriorate, which is the case just before a peak in the space usage plot (Figure 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.

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 long lists of consecutive elements. Given that we consult an element at the i-th

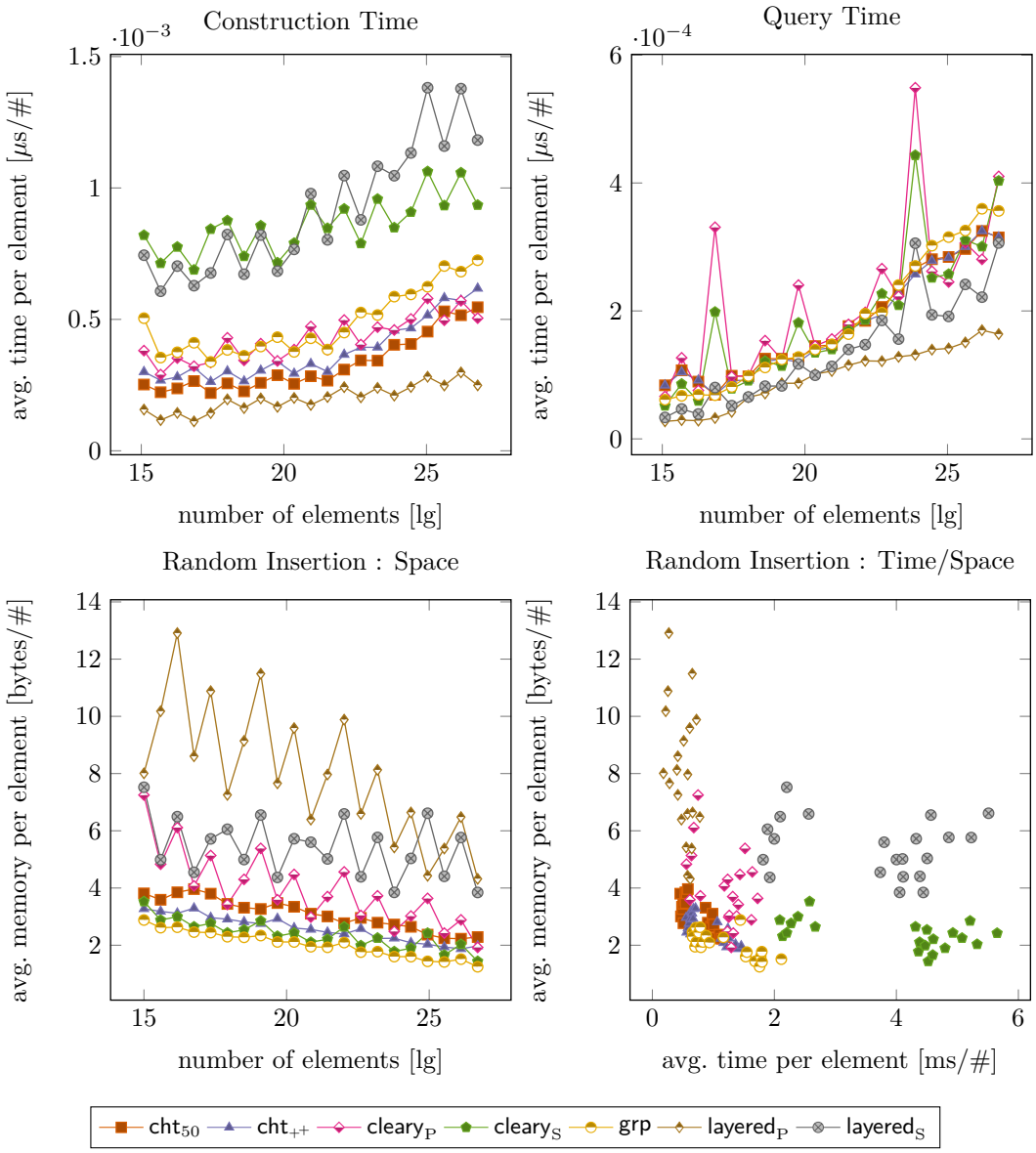

Figure 1 *Top Left*: Time for inserting $2^{10} \cdot (3/2)^n$ randomly generated 1-bit values and 32-bit keys into a compact hash table, for $n \geq 0$ (cf. Section 4.1). *Top Right:* Time for querying all inserted elements. *Bottom Left*: Peak memory needed during construction. *Bottom Right*: Memory and time per stored element.

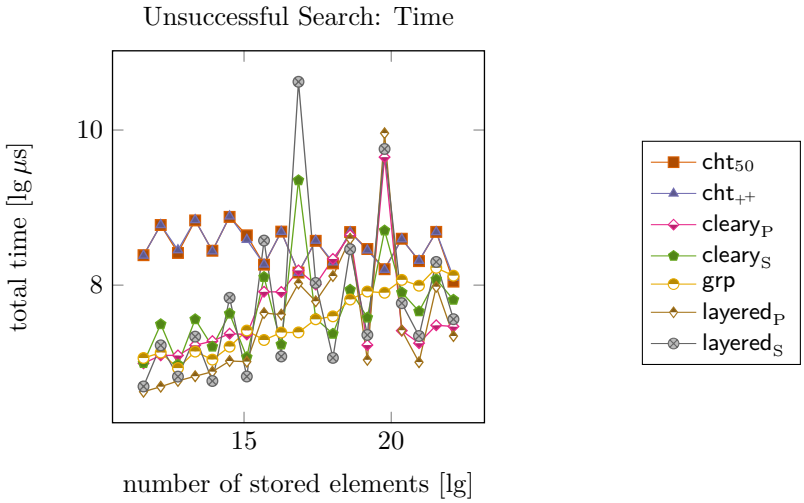

Figure 2 Time for looking up 2^{10} random keys that are not present in the hash tables.

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 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 cht 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.

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. cht 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 cht and grp.

4.2 Non-Compact Hash Tables

Figure 3 and Figure 4 show an additional comparison with 8-bit values and two highly-optimized non-compact hash tables, namely:[8] Google's sparse hash table[4] google and Tessil's sparse map[9] tsl. Both google and tsl are *sparse*, resolve collisions with quadratic probing, use the SplitMix hash function [21], and had maximum load factor set to 0.95.

4.2.1 Insertions and Lookups

In Figure 3, we conducted the same experiment as in Section 4.1.1 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. cht_{50} also shades google during construction.

[8] Unfortunately, we could not evaluate other hash tables mentioned in the introduction. The implementation of [10] lacks resize capabilities (`http://algo2.iti.kit.edu/sanders/programs/cuckoo/`), and the implementation of [12] has a bug (`https://github.com/TooBiased/DySECT/issues/2`).

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

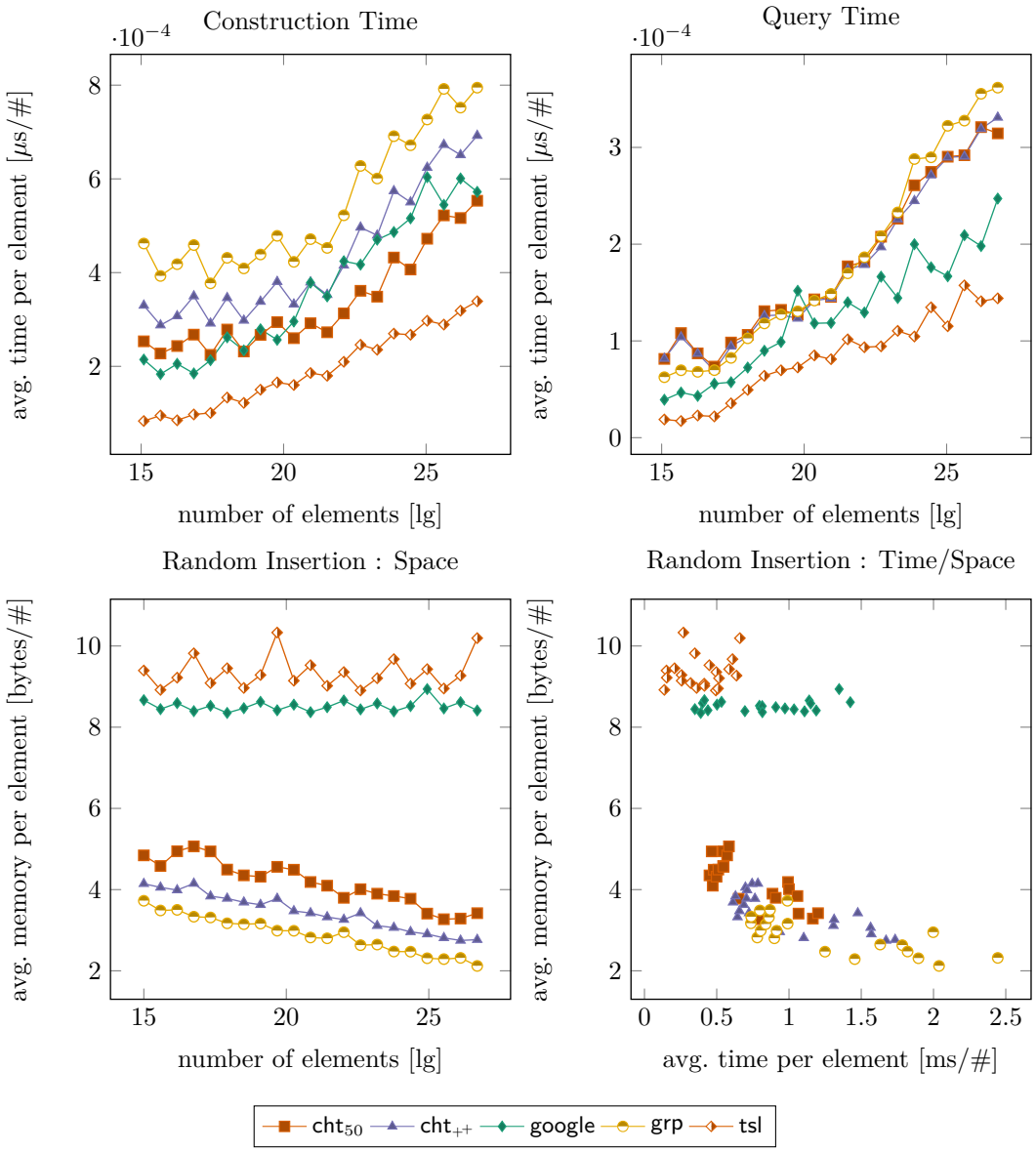

Figure 3 *Top Left*: Time for inserting $2^{10} \cdot (3/2)^n$ randomly generated 8-bit values and 32-bit keys into a not-necessarily compact hash table, for $n \geq 0$ (cf. Section 4.2). *Top Right:* Time for querying all inserted elements. *Bottom Left*: Peak memory needed during construction. *Bottom Right*: Memory and time per stored element.

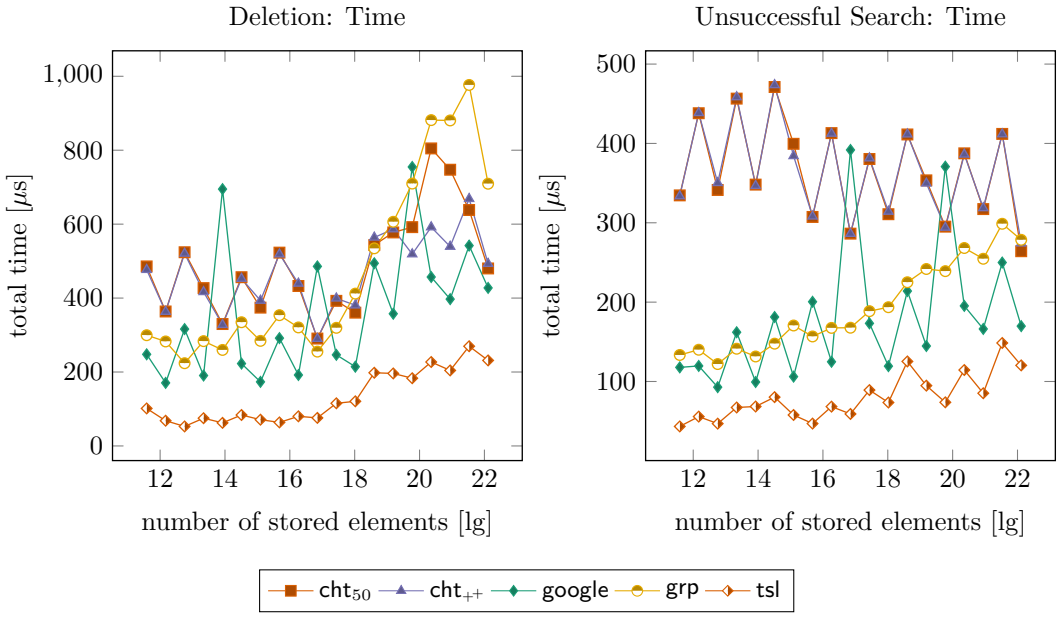

Figure 4 *Left*: Time for erasing 2^{10} random keys that are present in the hash tables. *Right*: Time for looking up 2^{10} random keys that are not present in the hash tables. In both figures, the number of elements (x-axis) is the number of elements a hash table contains (cf. Section 4.2.2).

4.2.2 Removing Elements

Figure 4 left shows the time to remove 2^{10} random elements. We used the hash tables created during the construction benchmark (Figure 3) 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 cht on large instances. Experiments for unsuccessful searches (Figure 4 right) show a similar pattern.

5 Conclusions and Future Work

We have suggested a simple approach for implementing compact hash tables, and our implementations show positive results. The experiments reveal our hash tables grp and cht 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 cht 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 cht and grp as close to $b_{\max}$ as possible, resulting in long scan times.

There are numerous avenues for future work. An analytical treatment of the space usage of the implemented 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 differ significantly to this quantity,

as we allocate many tiny fragments of space. A dedicated memory manager can reduce this space overhead and may also reduce the memory requirement of the bucket pointers by allocating a large contiguous array, pointers into which may require 32 bits, or less.

The AVX2 SIMD instruction set provides a major performance boost over earlier instruction sets like SSE – with benchmarks for comparing strings[10] indicating a speed boost of more than 50% for long strings. We wonder whether we can gain an even steeper acceleration in our hash tables when working with the newer AVX256 instruction set.

References

1 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. FOCS*, pages 787–796, 2010.

2 Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proc. SODA*, pages 11–19, 2004.

3 Daniel K. Blandford and Guy E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Trans. Algorithms*, 4(2):17:1–17:25, 2008.

4 Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

5 Paolo Carlini, Phil Edwards, Doug Gregor, Benjamin Kosnik, Dhruv Matani, Jason Merrill, Mark Mitchell, Nathan Myers, Felix Natter, Stefan Olsson, Silvius Rus, Johannes Singler, Ami Tavory, and Jonathan Wakely. *The GNU C++ Library Manual*. FSF, 2018.

6 John G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

7 John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper.*, 23(3):277–291, 1993.

8 Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Patrascu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *Proc. LATIN*, volume 3887 of *LNCS*, pages 349–361, 2006.

9 Johannes Fischer and Dominik Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017.

10 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005.

11 Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. Dynamic path-decomposed tries. *arXiv 1906.06015*, 2019.

12 Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019.

13 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

14 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

15 Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. Compact dynamic rewritable (CDRW) arrays. In *Proc. ALENEX*, pages 109–119, 2017.

16 Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. m-Bonsai: A practical compact dynamic trie. *Int. J. Found. Comput. Sci.*, 29(8):1257–1278, 2018.

17 Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *Proc. RANDOM*, volume 1518 of *LNCS*, pages 159–170, 1998.

18 Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 357–368, 2003.

[10] `https://github.com/koeppl/packed_string`

19 John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. SIGMETRICS*, pages 134–142, 1990.

20 Kenneth A. Ross. Efficient hash probes on modern processors. In *Proc. ICDE*, pages 1297–1301, 2007.

21 Guy L. Steele Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proc. OOPSLA*, pages 453–472, 2014.