Hash Tables with Pseudorandom Global Order

WOLFGANG BREHM

wolfgang.brehm@cfel.de
Coherent Imaging Division, CFEL,
DESY, NotkestraÃe 85 22607 Hamburg

Abstract. Given a sorting of the keys stored in a hash table one can guarantee a worst case time complexity for associations of O(log(n)) and an average complexity of O(log(log(n))) , thereby improving upon the guarantees usually encountered for hash tables using open addressing. The idea is to use the numerical order given by a hashing function and resolve collisions upholding said order by using insertion sort on the small patches that inevitably form. The name `patchmap` has been devised for the implementation of this data-structure and the source code is freely available.

Keywords: hash table, complexity analysis, c++

(Received February 20th, 2019 / Accepted June 22th, 2019)

1 Introduction

Hashmaps are an essential part of many algorithms as they allow associative retrieval of n elements in average constant time and have a space complexity of O(n). The key is associated to the value by computing a hash function of the key and storing the key value pair in an array at the position indicated by the hash. This position is called bucket. As long as there are no collisions, that is two different keys that produce the same position, different hash map implementations are conceptually identical. This can be the case in the limit for an infinitely large array or in the case of a perfect hashing function. As memory is constrained and there can be no one predetermined perfect hash function for arbitrary keys the key step where different hash map implementations differ is the resolution of hash collisions. The better the hash collisions are handeled, the more of them can be allowed to occur and the smaller the internal array can be. Different hash table implementations are therefore to be judged by the tradeoff between memory required and the time to find, insert or delete a key they offer.

Collision resolution strategies are commonly classified into open addressing and chaining. Chaining means that each position in the array stores a pointer to another data structure that stores all the entries with the same hash value. Using a data structure with O($\log(n)$) worst case lookup, insertion and deletion like a balanced tree will yield strong worst case guarantees for all operations but low practical performance due to the high degree of indirection and therefore likely cache misses. Open addressing means that the address in the hash table is not directly determined by the hash of the key but “open”. The position to store an entry is found by starting from the position indicated by the hash value and then searching for a free bucket with some probing strategy. For hash tables using open addressing the load factor α, the ratio between elements in the table and the total number of buckets, is the main determinant for memory efficiency.

The idea to revisit ordered hash tables was inspired by a scientific use case. Many large histograms needed to be computed and the standard `c++` hash table was not able to hold all the elements it would have needed to within the given memory constraints. The interim solution was to use sorted arrays, which provided the minimum possible memory footprint, but a higher time complexity. The desire arose to interpolate between the two solutions. A data structure that would act like a sorted array when memory is tight and gradually more like a hash map the more memory becomes available would be the ideal solution.

The central idea of this publication is to use an ordering of the keys consistent with the hash value of the key to resolve the collisions that occur in a hash table. The hash function is chosen in a way that its application to the keys will yield uniformly distributed numbers even if the keys themselves are not evenly distributed. Using sorting to resolve hash collisions has been suggested before, most notably by Amble and Knuth [5]. The difference in the proposed algorithm is not using the sorting given by the value of the key but the hash of the key. This leads to a global order of the keys stored in the hash table and therefore allows for efficient lookup using interpolation search and binary search. Interpolation search has the best average case and a binary search provides the best worst case complexity for lookups in a sorted array. This means that the lookup, successfull or not, of a key will have a worst case time complexity of O($\log(n)$), when done right, and an average complexity of O($\log(\log((1-\alpha)^{-1}))$). These guarantees are as good as for optimally ordered hash tables [4][1] and stronger than the guarantees that most other probing strategies can give for lookups [8]. And just like for binary tree hashing, which approximates optimal ordering, the faster lookups come at higher cost for insertions of up to O(n) for a full table, but are constant for tables that have a load factor less than 1. Note that the *probable* worst case for advanced probing strategies equally is O($\log(n)$) as it also approaches optimal ordering, and that robin hood hashing [2] with dynamic resizing achieves the same worst case complexity with exceedingly high probability, as the hash table can be resized as soon as the maximum displacement is larger than O(log(n)), but the iterative resizing is not guaranteed to terminate with O(n) space. Robin hood hashing with dynamic resizing is sucessfully implemented in the `flat_hash_map` and `sherwood_map` [9] of Malte Skarupke for example.

2 The Algorithm

The key-value pairs are stored in an array of size m. A binary mask of the same size is used to indicate whether a bucket in the array is free or set.

The proposed algorithm requires a hashing function, preferably bijective, an order on the keys to be stored if the hashing function is not bijective and a mapping function that compresses the range of the hash value onto the range of the size of the array while preserving the order of the hash.

[1]The authors cauteously claim that the average complexity of successfull lookups in an optimally ordered hash table are bounded by a constant but give no proof or convincing simulations.

The hash function can be any function that maps the keys onto positive integers less than some integer, but as the resolution of collisions is essentially a variant of linear probing it is crucial for good performance that the hash values are evenly distributed. A hash function that works well can be composed from a binary rotation and two multiplications with uneven random integer constants. The map function used is the multiplication of the hash value with m and then dividing the result by the maximum hash value plus one:

```
map(h) :=
return (h*m)
       /(maximum hash value plus 1)
```

This can efficiently be computed for fixed width integer types using long multiplication with two fixed width integers of the same length or a simple multiplication with a fixed width integer with double the size. Note that the division for hash values that fill the full range given by the digits is a simple shift of the digits and does therefore not need to be computed explicitly, this technique is known as `fastrange` [6].

Two types of comparisons need to be considered, comparing keys with keys and keys with free bucket positions. When comparing two keys a key is considered to be less if its hash value is less than the other or in case the hash values are equal if the key itself is less than the other key. When comparing a key to an empty position it is considered to compare less if the application of the map function to the result of the hash function of the key is less than the position: position < map(hash(key)) .

As long as there have been no collisions the keys stored in the hash table are already in the order given by the hash value. The lookup depends on this order. The operations for insertion and deletion should therefore be implemented in a way so they uphold this order. This is achieved by finding the closest free bucket starting from the position indicated by map(order(key)). The key value pair is inserted into the free bucket and then swapped with neighbouring entries until the order of the hash table is restored. This procedure behaves like linear probing in a complexity analysis and therefore needs the same number of probes, O($1+(1-\alpha)^{-2}$) The term $\frac{1}{4}\left(\frac{\alpha}{(1-\alpha)^2}+\alpha\right)$ has been found to describe the required number of swaps wells. For a numerical simulation see figure 3. Deletion works analogous to insertion, but in reverse.

The procedure for inserting is conceptually identical to the insertion step in Algorithm B in [5] with linear probing, but instead of comparing the keys directly, the hash value of the keys is being compared.

```
insert(key, value) :=
1. find the free position p
   closest to map(hash(key))
2. mark p as set
3. insert the key value pair at p
4. while there is a key to the right
   that is less
  4.1 swap the keys
  4.2 swap the values
  4.3 p := p-1
5. while there is a key to the left
   to which the key is less
  5.1 swap the keys
  5.2 swap the values
  5.3 p := p+1
```

```
erase(key) :=
1. p := find(key)
2. while there is a key right of p
   where map(hash(key))>position
  2.1 move the key value pair left
  2.2 advance p to the right
3. while there is a key left of p
   where map(hash(key))<position
  3.1 move the key value pair right
  3.2 advance p to the left
4. mark p as free
```

Given a hash map where the keys are globally ordered an interpolation search can be employed to retrieve keys. Starting with the first and last element as `lower_limits` and `upper_limit` respectively of the array, the range is iteratively subdivided by linear interpolation until the key at midpoint equals the key that needs to be found or the range is empty or the maximum number of iterations has been reached. If the number of iterations exceeds O($\log(n)$) the search switches to a binary search. Almost always the interpolation search will terminate before that, but switching to a binary search guarantees that the search will terminate in O($\log(n)$).

For a table that has been filled completely the average time complexity for key retrieval will be O($\log(\log(n))$) and the worst case time complexity will be O($\log(n)$) as this case reduces to the search in a sorted list of uniformly distributed integers. The algorithm for inserting new keys behaves like linear probing

```
find_binary(key,
        lower_limit,
        upper_limit) :=
1. midpoint :=
   lower_limit
  +(upper_limit-lower_limit+1)/2
2. if midpoint is set
   and (key at midpoint = key)
   then return midpoint
3. if (lower_limit = upper_limit)
   then return not_found
4. if key
     is less than
     midpoint
   then upper_limit := midpoint
5. if midpoint
     is less than
     key to be found
   then lower_limit := midpoint
6. goto 1
```

operation	average	worst case
insertion	O($1 + (1 - \alpha)^{-2}$)	O(n)
deletion	O($1 + (1 - \alpha)^{-2}$)	O(n)
lookup	O($1 + \log_2(1 + \log_2(\frac{2-\alpha}{2-2\alpha}))$)	O($\log_2(n)$)

Table 1: Computational complexities for hash table operations in the `patchmap`. $\alpha = \frac{n}{m}$ is the the load factor, m the number of buckets and n the number of elements stored in the table.

in the way that primary clustering occurs. The complexity for lookup with a linear search would therefore be $\frac{1}{2}(1 + (1 - \alpha)^{-1})$ for a successfull lookup and at most one probe more for an unsuccessfull search, as just like in [5] for algorithm B the search can be concluded as soon as a key outside of the search range is encountered.

However the proposed hash table is in a global order, made up of small patches that are ordered internally *and* with respect to each other. This means probe positions can be interpolated from the keys at the upper and lower limit starting from the beginning and end of the table reducing the number of probes required for lookups to $1 + \log_2(1 + \log_2(\frac{2-\alpha}{2-2\alpha}))$. The exact term depends on implementation details like the way the position is interpolated. For a numerical simulation see figure 3.

Simulations seem to indicate that constructing a full `patchmap` by successively inserting new keys scales like $O(n^{\frac{3}{2}})$. But a full `patchmap` can be trivially constructed by sorting the keys directly in hash order, ensuring a complexity of O($n\log(n)$).

```
find_interpol(key) :=
 1. lower_limit := 0
 2. upper_limit := m - 1
 3. midpoint := map(hash(key))
 4. i := 0
 5. if midpoint is set
      if (key at midpoint = key)
      then return midpoint
 6. if (lower_limit = upper_limit)
    then return not_found
 7. if ( key to be found )
       is less than
       ( midpoint )
    then upper_limit := midpoint
 8. if ( midpoint )
       is less than
       ( key to be found )
    then lower_limit := midpoint
 9. i := i+1;
10. if ( i > 2 log2(n+1) )
    then return find_binary(
                key,
                lower_limit,
                upper_limit)
11. interpolate midpoint linearly
12. goto 5
```

operation	average	worst case
insertion	$O(1+(1-\alpha)^{-1})$	$O(n)$
deletion	$O(1+(1-\alpha)^{-1})$	$O(n)$
lookup	$O(1+(1-\alpha)^{-1})$	$O(n)$

Table 2: Computational complexities for hash tables using double hashing. $\alpha = \frac{n}{m}$ is the load factor, m the number of buckets, n the number of elements stored in the hash table.

3 Results

Using the proposed algorithm a hash table can be implemented that has a good tradeoff between memory requirement and time used for lookup, insertion and deletion. The average time for inserting into, deleting from and searching for a key that has 50% chance of being in the table has been computed for up to 2^{27} keys, for hash tables mapping from 32 bit integers to 32 bit integers. The tests were carried out on a 64 bit linux machine with an Intel Core i5-3320M CPU @ 2.60GHz and a CAS latency of 13.5 ns.

The patchmap was compared to `khash` from attractivechaos klib [1], Malte Skarupke's `bytell` [10] and his `flat_hash_map` [11] using robin hood hashing, `google::sparsehash` from the public github release, `sparsepp` [7], a sparsehash fork and `std::unordered_map`. All executables were compiled with the highest optimization level and using `clang` version `7.0.1`.

Step 0, the patch before insertion:

key	x	6	2	5	7	9	3	6	x	x	...
hash	x	1	2	3	3	4	5	6	x	x	...
mask	0	1	1	1	1	1	1	1	0	0	...

Step 1, inserting key 1 with hash 5 into a patch:

key	x	6	2	5	7	9	3	6	1	x	...
hash	x	1	2	3	3	4	5	6	5	x	...
mask	0	1	1	1	1	1	1	1	1	0	...

Step 2, restoring the order in the patch:

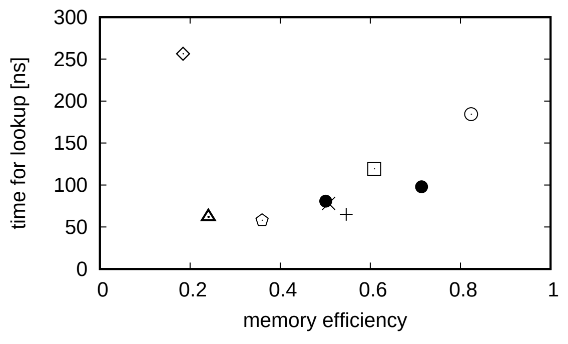
key	x	6	2	5	7	9	1	3	6	x	...
hash	x	1	2	3	3	4	5	5	6	x	...
mask	0	1	1	1	1	1	1	1	1	0	...

Figure 1: Three step scheme of inserting the key 1 with the hash value 5 into a patch in the `patchmap`. The hash function is non bijective. The map function in this example is the identity function. The key 1 is inserted at the position closest to map(hash(key))=5, which is position 8, then it is swapped to the left until the order of the patch is restored.

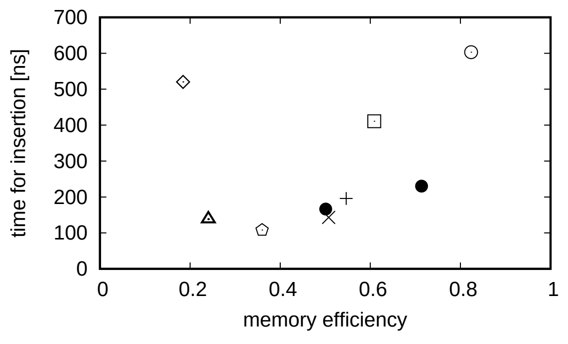
From figure 2 it can be seen that the patchmap sits right at the apparent invisible barrier outlined by other good hash map implementations like `khash`, `bytell` and `google::sparsehash` on a map of memory usage and time efficiency. This broad comparison cannot capture all aspects of hash table performance and it is unfair in several regards. The comparison to the `std::unordered_map` is unfair, as it has to adhere to the `c++` standard requiring that references to elements stay constant for all hash table operations. The hash tables `sparsepp` and `google::sparsemap` are designed to minimise the peak, not the average, memory consumption instead. Deleting elements from the table is done by marking elements as deleted instead of actually removing them from the table in all implementations except the `patchmap`, `bytell` and `std::unordered_map`. This saves time in the short run, but having deleted elements still in the table can lead to a degradation of performance if these elemets are allowed to accumulate. Such behavior can be triggered by repeatedly inserting and erasing many keys, see table 3, but it is considered rare in real world applications.

4 Availability

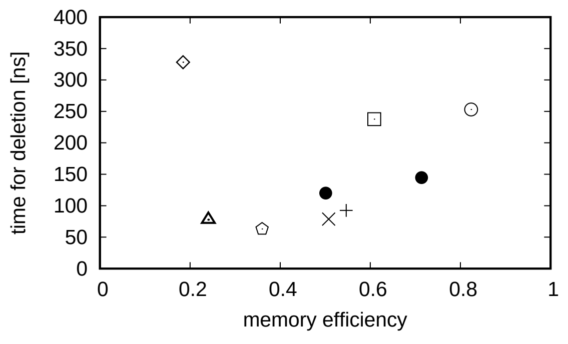
An implementation of the patchmap as outlined in this paper is available on github under an unmodified MIT license:
`https://github.com/1ykos/ordered_patch_map`



((a))



((b))



((c))

Figure 2: Selected hash table implementations and their performance given in average memory efficiency on the horizontal axis and average time for lookup, insertion and deletion of a random key in nanoseconds on the vertical axis, computed for up to 2^{27} keys. Insertion contains the time needed for dynamic expansions.
`patchmap`: ●, `khash`: ×, `bytell`: +, `google::sparse_hash_map`: ○, `google::dense_hash_map` ⬠, `flat_hash_map`: △, `std::unordered_map`: ◇, `sparsepp`: □
There are two points for the `patchmap` - one optimized for memory efficiency and one for speed.

hash table	slowdown	rehasing
`patchmap`	no	none
`bytell`	no	none
`std::unordered_map`	no	none
`flat_hash_map`	no	none
`sparsepp`	minor	frequent
`google::sparse_hash_map`	minor	frequent
`google::dense_hash_map`	severe	rare
`khash`	severe	occasional

Table 3: Performance degradation of hash table implementations when keys from a fixed set are repeatedly chosen at random and then alternatingly inserted or deleted from the hash table to show the effect of many dead keys in the table.

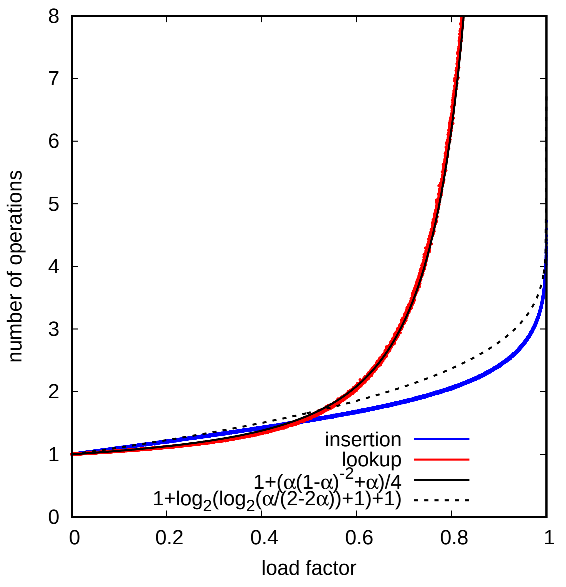


Figure 3: Average cost for insertion and lookup in the hash map as it is being filled up with 2^{28} pseudorandom keys. Insertion (in red) shows a strongly quadratic behavior and the function $1 + \frac{1}{4}\left(\frac{\alpha}{(1-\alpha)^2} + \alpha\right)$ is indicated with a solid black line. Lookup (in blue) is almost linear in α and shows a steep increase only for load factors close to 1. The theoretically derived complexity bound for lookup $1 + \log_2(\log_2(\frac{2-\alpha}{2-2\alpha}) + 1)$ is indicated with dashed black line.

5 Discussion

While chaining hash tables can achieve better or at least equal bounds on the complexity of all hash table operations it comes at the cost of a higher degree of indirection and an additional overhead to storing the pointers to the secondary data structure containing the elements with the same hash value. This is why in practice chaining usually performs worse in terms of memory efficiency and speed. Of the hash tables using open addressing the `patchmap` offers the tightest worst case bounds for lookup but double hashing and other advanced probing strategies achieve better run time complexities for insertion and deletion. Probing strategies that approximate an optimal arrangement at a higher cost of insertion like Robin hood hashing in conjunction with double hashing [2] [3], binary tree hashing [4] and the algorithm presented here seem to have the same average case complexity for lookups, equally asymptotically approaching optimality. In practice however the mask structure and the cache-friendly linear access pattern of the `patchmap` allows to effectively lower the constants associated with insertion and deletion and seems to be a good compromise for medium high load factors of around 83%. The fact that the `patchmap` can perform lookups quickly even when it is very full could make it suitable for use cases where memory is constrained and the hash table itself is virtually static. Other than that the performance characteristics of the patchmap mean that there are some cases for which it is the optimal choice in general. The quadratically increasing costs for inserting and deleting elements when the `patchmap` fills up limit its utility where memory is very constrained and insertion and deletion are frequent.

Sorting based on the hash is almost identical to robin hood hashing with bidirectional linear probing. When deciding which key to displace and which to keep at a given position, the larger key will also be displaced more than the smaller key. The key difference is that when ensuring an absolute order a guarantee can be given for the worst time complexity because a binary search can be employed.

6 Acknowledgements

This research was undertaken alongside my research activities in crystallographic data processing under Professor Henry Chapman. While the requirement to generate very large histograms promted the inception of this algorithm, I am very grateful for the freedoms I am enjoying in the Coherent Imaging research group and that I was able to pursue the developement of the algorithm as well.

References

[1] attractivechaos. klib, 2018.

[2] Celis, P. *Robin Hood Hashing*. PhD thesis, Waterloo, Ont., Canada, Canada, 1986.

[3] Devroye, L., Morin, P., and Viola, A. On worst case robin-hood hashing, 2004.

[4] Gonnet, G. H. and Munro, J. I. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8(3):463–478, 1979.

[5] Knuth, D. E. and Amble, O. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 01 1974.

[6] Lemire, D. Fast random integer generation in an interval. *CoRR*, abs/1805.10941, 2018.

[7] Popovitch, G. `sparsepp`, 2019.

[8] Ramakrishna, M. An exact probability model for finite hash tables. In *Proceedings. Fourth International Conference on Data Engineering*, pages 362–368. IEEE, 1988.

[9] Skarupke, M. I wrote the fastest hashtable, 2017.

[10] Skarupke, M. A new fast hash table in response to googleâs new fast hash table, 2018.

[11] Skarupke, M. `flat_hash_map`, 2018.