

Worm Hashing

Bruno Roustant
2015-09-02

Overview	1
Worm hashing principle.....	1
Comparison with other strategies	2
Table structure	2
Entry addition	3
Bucket at index h is free	3
Bucket at index h contains a head-of-chain	4
Bucket at index h contains a tail-of-chain.....	6
Load factor	7
Entry lookup	7
Entry removal.....	7
Higher load factor with recursive tail-of-chain displacement.....	8

Overview

Worm hashing is a strategy of collision resolution in hash table, a hybrid technique between separate chaining and open addressing, inspired by Cuckoo and Hopscotch Hashing.

It is designed for three goals:

- Reduce the space usage (memory footprint).
- Keep all hash table operations efficient.
- Adapt easily to primitive keys and values for compactness and performance.

Worm hashing principle

The algorithm creates chains of entries with the same hash value (like separate chaining), stored directly in the bucket array (like open addressing).

An entry-chain is composed of one head-of-chain entry, and multiple tail-of-chain entries to store hash collisions.

The head-of-chain entry is unmovable, always at the bucket at index equal to the hash value h.

Each tail-of-chain entry has the same hash value h, but occupies a bucket at a different index i. It can be moved to make room for a new item having a hash value i. The new item is stored in bucket i as a new head-of-chain entry (hash = i). The tail-of-chain entry is moved to another free bucket j, but it is still chained to the entry-chain for hash value h.

This principle of chained entries with an unmovable head-entry and multiple movable tail-entries looks like a worm: its head is attached to a table bucket, with a moving elastic tail.

Comparison with other strategies

- Like Open Addressing, it achieves reduced space usage and cache advantages over separate chaining. All entry items are stored in the bucket array itself. It is memory efficient, especially for primitive types. It cannot have more elements than table buckets.
- Like Coalesced Hashing, entries with the same hash value are chained, but Worm Hashing does not allow the chains to coalesce.
- Like Separate Chaining, it does not exhibit much clustering effects; in fact, the table can be efficiently filled to a high density (0.8 to 0.9 load factor) without performance loss.
- Like Separate Chaining and Hopscotch Hashing, it guarantees an item lookup of exactly the number of hash collisions at the item hash bucket. No linear probing like in open addressing.
- Like Cuckoo and Hopscotch Hashing, an item already stored in a bucket may be pushed to make room for a new item added to the table. Like Hopscotch Hashing, some linear probing is done when a new item is added to find free bucket and make room.

At the end, Worm Hashing is as time efficient as Separate Chaining or Hopscotch Hashing, while being more space efficient. It is more compact than Open Addressing because the table can be filled at a higher load factor.

Table structure

The structure is composed of an array of entry buckets (called `entries[]`), plus an additional forward-chaining data stored in a second array (called `chaining[]`). Both arrays are circular (the bucket after the last bucket is the first bucket).

Note that in real implementation, the `entries[]` array is usually implemented with two distinct arrays `keys[]` and `values[]`. But it is simpler here to describe it as a single array, and it does not change the algorithm.

`chaining[]` is an integer array, byte array for space compactness. For each bucket, `chaining[i]` represents the offset to the next entry in the chain, in a single byte, with some additional encoding.

- `chaining[i] = 0` for an empty free bucket.
- The sign bit is used to encode whether the bucket contains a head or tail entry.
 - `chaining[i] > 0` for head-of-chain entries.
 - `chaining[i] < 0` for tail-of-chain entries.
- `absolute(chaining[i])` is the forward offset to the next entry in the chain (different item having the same hash value). The arrays are circular, meaning that an entry at the end of the array may point to an entry at the beginning of the array with a forward offset.
- A special value marks the end of the chain. It is noted **EOC** below. It can be on either the head-entry (127 or **EOC**), or on the final tail-entry (-127 or **-EOC**).
- So this lets a maximum chaining offset of 126, always forward.
- `chaining[i] != -128` (unused value).

Entry addition

When a new entry is added, with h its key hash value, there are 3 cases:

1. The bucket at index h is free ($\text{chaining}[h] = 0$).
2. The bucket at index h contains a head-of-chain entry ($\text{chaining}[h] > 0$).
3. The bucket at index h contains a tail-of-chain entry ($\text{chaining}[h] < 0$).

We illustrate the algorithm with an example.

Initially the map is empty. Let's consider the 2 map arrays `entries[]` and `chaining[]`:

entries[]									
chaining[]									
index	0	1	2	3	4	5	6	7	8

Let's enumerate the addition algorithm for the 3 cases.

Bucket at index h is free

The map is empty. We add a new entry A, which key hash value is 0.

1. Look at the `chaining[]` offset at index 0. `chaining[0] == 0`, this means the bucket at index 0 is free.
2. Simply set `entries[0] = A`, and `chaining[0] = EOC`.
EOC is the end-of-chain value (127) marking the end of an entry-chain.
`chaining[0]` is positive to indicate that the bucket contains a head-of-chain entry.

Then we add two other new entries B and C, respectively with key hash values 2 and 8. There is no hash collision, so the entries are at their respective hash indexes in the `entries[]` array. All the entries are head-of-chain.

For the clarity of the example, we remind the hash value for each entry. We note the entry letter followed by a dot and then its hash value. In practical implementation this value is not stored to reduce memory usage.

entries[]	A.0		B.2						C.8
chaining[]	EOC		EOC						EOC
index	0	1	2	3	4	5	6	7	8

Bucket at index h contains a head-of-chain

On the same map, we add a new entry D which key hash value is 0.

1. Look at the chaining[] offset at index 0. chaining[0] > 0, this means the bucket at index 0 contains a head-of-chain.
2. Follow the entry chain up to the last entry of the chain.
Since absolute(chaining[0]) == EOC, this means this entry is the last of the entry chain.
3. From the last entry of the chain, find the next free bucket by linear probing to the right (array is circular). If no free bucket can be found in the range of the maximum offset (126), then this map needs to be enlarged, re-hashed, and the new entry is added again.
In the example, we start by looking at the index 1. This bucket is free.
4. Put the new entry in the free bucket. Link the new entry to the entry chain. Remind that chaining[x] is the offset from index x to the next entry in the chain. If the next entry is at index y, chaining[x] = y-x.
Set entries[1] = D.
Set chaining[0] = {offset from 0 to 1} = 1, with positive sign since the entry at index 0 is head-of-chain.
Set chaining[1] = -EOC, because it is the end of the entry chain, and it is a tail-of-chain, so the offset has to be negative.

Green arrow indicates a chain of entries (a "worm").

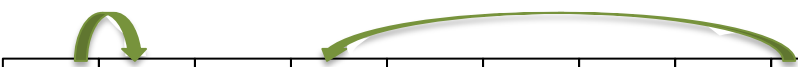
entries[]	A.0	D.0	B.2						C.8
chaining[]	1	-EOC	EOC						EOC
index	0	1	2	3	4	5	6	7	8

Then we add a new entry E which key hash value is 8.

1. Look at the chaining[] array at index 8. chaining[8] > 0, this means the bucket at index 0 contains a head-of-chain.
2. Follow the entry chain up to the last entry of the chain.
Since absolute(chaining[8]) == EOC, this means this entry is the last of the entry chain.
3. From the last entry of the chain, find the next free bucket by linear probing to the right (array is circular).

In the example, we start by looking at the next circular index 0. This bucket is occupied. We continue the linear probing until index 3, with a free bucket.


4. Put the new entry in the free bucket. Link the new entry to the entry chain.
 Set $\text{entries}[3] = E$, and $\text{chaining}[3] = \text{-EOC}$.
 Set $\text{chaining}[8] = \{\text{circular offset from 8 to 3}\} = 4$, with positive sign since the entry at index 8 is head-of-chain.



entries[]	A.0	D.0	B.2	E.8					C.8
chaining[]	1	-EOC	EOC	-EOC					4
index	0	1	2	3	4	5	6	7	8

Then we add a new entry F which key hash value is 0.

1. Look at the chaining[] offset at index 0. $\text{chaining}[0] > 0$, this means the bucket at index 0 contains a head-of-chain.
2. Follow the entry chain up to the last entry of the chain.
 We start at index 0, $\text{chaining}[0] == 1$, so next index = current index (0) + offset (1) = 1. So we look at index 1, there $\text{chaining}[1] == \text{-EOC}$, which means we are at the last entry of the chain.
3. From the last entry of the chain, find the next free bucket by linear probing to the right (array is circular).
 In the example, we start by looking at the next circular index 2. This bucket is occupied. We continue the linear probing until index 4, with a free bucket.
4. Put the new entry in the free bucket. Link the new entry to the entry chain.
 Set $\text{entries}[4] = F$, and $\text{chaining}[4] = \text{-EOC}$.
 Set $\text{chaining}[1] = -\{\text{circular offset from 4 to 1}\} = -3$, with negative sign since the entry at index 1 is tail-of-chain.



entries[]	A.0	D.0	B.2	E.8	F.0				C.8
chaining[]	1	-3	EOC	-EOC	-EOC				4
index	0	1	2	3	4	5	6	7	8

Bucket at index h contains a tail-of-chain

On the same map, we add a new entry G which key hash value is 1.

1. Look at the chaining[] offset at index 0. chaining[1] < 0, this means the bucket at index 1 contains a movable tail-of-chain.
2. Move the tail-of-chain entry to a free bucket, while keeping it linked to its entry chain. To maintain the forward chaining, it is allowed to move the entry up to either the next entry in the chain, or the maximum range for the offset (126, from the previous entry in the chain), whichever is the first.
 - a. If a free bucket is found before the next entry in the chain:
The current tail-of-chain entry is moved there and the chain links are updated (1).
 - b. If the next entry is within the max offset range of the previous entry in the chain, and there is a free bucket found by linear probing after the last entry of the chain:
The current tail-of-chain entry is moved to the free bucket at the new end of chain, the previous entry link is modified to point to the next entry in the chain, and the chain links are updated (1).
 - c. Otherwise the map needs to be enlarged, re-hashed, and the new entry is added again.

(1) To update chain links, we need to find the entry before the current entry in the chain. There are two options:


- Backward linear probing (circular) to find the bucket which chaining[] offset points to the current entry index.
- Re-compute the hash value h of the current entry, then go to index h, follow the entry chain up to the current entry and keep the previous entry.

In our example, there is no free bucket between entry D at index 1 and the next entry in the chain, F, at index 4. The next entry F is within the max offset (126) of the previous entry A at index 0.

We search a free bucket by linear probing after the last entry in the chain F. We find it at index 5.

So D is moved to index 5, A is linked to F, and F is linked to D.

3. Put the new entry in the freed bucket, as head-of-chain.
Set entries[1] = G, and chaining[1] = EOC.



entries[]	A.0	G.1	B.2	E.8	F.0	D.0			C.8
chaining[]	4	EOC	EOC	-EOC	-1	-EOC			4

index

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

Load factor

With this simple tail-of-chain displacement, and rehashing when no free bucket is found by linear probing, Worm Hashing achieves above 0.75 load factor. It is possible to further increase the load factor, see section “Higher load factor with recursive tail-of-chain displacement” below.

Entry lookup

To lookup for an entry which key hash value is h , look at the `chaining[]` offset at index h :

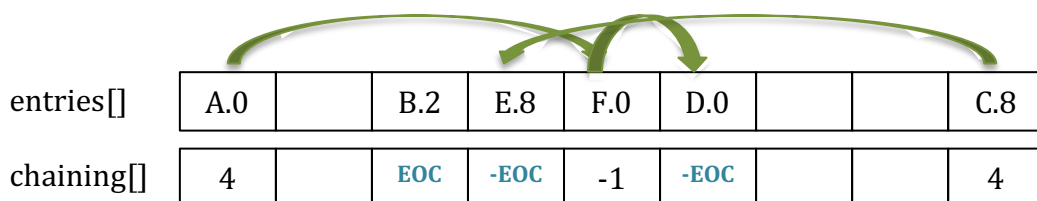
- If `chaining[h] ≤ 0`, no entry.
- If `chaining[h] > 0`, follow the entry chain and compare the lookup key with each entry.

So the entry lookup is straightforward. Its performance is not impacted by the map load factor. There is no linear probing, only entries in the chain (hash collisions) are iterated.

Entry removal

Let’s take the same map example. Now we remove entry G which key hash value is 1.

1. Look at the `chaining[]` offset at index 1.
If `chaining[1] ≤ 0`, there would be no match and no entry to remove, we would return immediately.
In the example, `chaining[1] == EOC`, which is positive, this means the bucket at index 1 contains a head-of-chain.
So we iterate over all the entries in the chain. If one matches, we have to remove it in the next step, otherwise there is no removal and we return.
`entries[1]` matches G, so we continue with next step to remove it.
2. Remove the matching entry and replace it by the last entry in the chain.
We start from entry G at index 1. Since `absolute(chaining[1]) == EOC`, it indicates this is the last entry of the chain. So entry G is simply removed and the bucket is freed.




index

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

Then we remove entry F which key hash value is 0.

1. Look at the chaining[] offset at index 0.
chaining[0] == 4 is positive, this means the bucket at index 0 contains a head-of-chain.
So we iterate over all the entries in the chain. entries[4] matches F, hence we continue with next step to remove it.
2. Remove the matching entry and replace it by the last entry in the chain.
We start from entry F at index 4. chaining[4] == -{offset to the next} = -1, so we iterate to index = 4 + abs(-1) = 5. chaining[5] == -EOC, it is the last entry of the chain. Finally entry F is replaced by entry D.



entries[]	A.0		B.2	E.8	D.0			C.8	
chaining[]	4		EOC	-EOC	-EOC			4	
index	0	1	2	3	4	5	6	7	8

Higher load factor with recursive tail-of-chain displacement

In the first approach described above, the algorithm may decide to rehash the whole map if it cannot find a free bucket. Rehashing the whole map is a costly operation, so it is more efficient to do it less often, and it is more compact if we can store more entries without rehashing.

The algorithm can be enhanced with recursive tail-of-chain displacement: At any time when the algorithm does not find any free bucket (by linear probing) to displace a tail-of-chain, it can attempt to recursively displace some other tail-of-chain entries in the offset range. This is the same idea as in Cuckoo Hashing, where one entry may push another, although here we first try to displace another tail-of-chain, and on success we then displace the initial tail-of-chain to make room.

In practice the number of these recursive attempts must be limited not to impact the performance. In our tests we used 10 attempts at recursive level 1, and 1 attempt at recursive level 2, with no more recursive level. This increased the load factor from above 0.75 (no recursive displacement) to above 0.85 (recursive displacement) on average.

With these limits, the overall performance of the map is not affected.

Tests showed that increasing the number of recursive levels, or increasing the number of attempts per level do not increase much further the load factor.