Allocation-conscious chained hash-table

NEIL A. EDELMAN

2022-02-22

Abstract

We define an inline-chained hash-table as a bucket scheme where each entry overlaps with an index to the next entry. Thus, the front entry in the bucket is closed; all others are open, taken from unoccupied slots. We show that this hash-table design is feasible, and can often be less expensive, performance-wise and to maintain. Having a chained hash-table with the simplicity of allocation of open-addressing is especially attractive for simple data.

1 INTRODUCTION

Performance is a critical issue, but we are also concerned with usability. Specifically, without higher-level language support for native hash-tables or automatic garbage-collection, we want to easily understand and maintain a general hash-table, with minimal surprise on behalf of the users.

Overlapping each entry with an index to the next entry creates a hash-table that is self-contained in memory, yet behaves as a chained hash in the regime where the load factor is less-then one.[1] We call this inline-chaining to differentiate it from separate-chaining, where the buckets are objects with links between them.

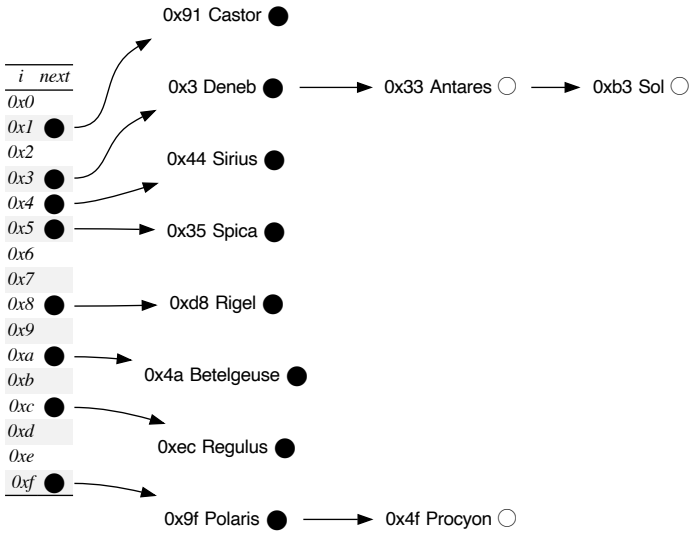
Thus, the *data* part of the *data structure* is similar to that of coalesced-hashing[2]. However, coalescing cannot occur. The closed entries that form the heads of buckets and the open entries that form a stack are orthogonal.

2 EXAMPLE COMPARISON

Figure 1 shows a comparison of some standard hash-table types. It uses D.J. Bernstein's *djb2* to hash a string to 8-bit unsigned integer. All tables use a most-recently-used heuristic as probe-order; experimentally, this was found to make little difference in the run-time, and is advantageous when the access pattern is non-uniform[3, 4].

Separate-chaining is seen in Figure 1a; this is more like T.D. Hanson's *uthash*: only being in one hash-table at a time. A similar hash to *C++*'s `std::unordered_map`, *Lua*'s `table`, and many others, would have another dereference between the linked-list and the entry. This style of hash-table allows unconstrained load factors. With ordered data, keeping a self-balancing tree cuts down the worst case to $\mathcal{O}(\log n)$[1], as in *Java*. The expected number of dereferences is a constant added to the number of queries. The nodes are allocated separately from hash-table.

Open-addressing[5] as seen in Figure 1b, is another, more compact, and generally more cache-coherent table design. Robin Hood hashing[6] has been used to keep the variation in the query length to a minimum; here, with the condition on whether to



(a) Separate-chaining.

i	*disp.*	*hash*	*key*
0x0	*1*	0x4f	Procyon
0x1	*0*	0x91	Castor
0x2			
0x3	*0*	0x3	Deneb
0x4	*1*	0x33	Antares
0x5	*2*	0xb3	Sol
0x6	*2*	0x44	Sirius
0x7	*2*	0x35	Spica
0x8	*0*	0xd8	Rigel
0x9			
0xa	*0*	0x4a	Betelgeuse
0xb			
0xc	*0*	0xec	Regulus
0xd			
0xe			
0xf	*0*	0x9f	Polaris

(b) Open-addressing.

i	*hash*	*key*	*next*
0x0			
0x1	0x91	Castor	●
0x2			
0x3	0x3	Deneb	● → 0xd → 0xe
0x4	0x44	Sirius	●
0x5	0x35	Spica	●
0x6			
0x7			
0x8	0xd8	Rigel	●
0x9			
0xa	0x4a	Betelgeuse	●
0xb	0x4f	Procyon	○
0xc	0xec	Regulus	●
0xd	0x33	Antares	○ → 0xe
0xe	0xb3	Sol	○
0xf	0x9f	Polaris	● → 0xb

(c) Inline-chaining.

Figure 1: A set of star names using different collision-resolution schemes. Load factor $^{11}/_{16}$ = 0.69. Expected value: chained number of queries, 1.4(7); open probe-length, 1.6(9), (standard deviation.) Order of insertion: Sol, Sirius, Rigel, Procyon, Betelgeuse, Antares, Spica, Deneb, Regulus, Castor, Polaris.

evict strengthened because of the most-recently-used heuristic. It has lower maximum load-factor, because clustering decreases performance as the load-factor reaches saturation.[7] Although they have less data *per* entry, on average they have more entries. Practically, 0.69 is high; *Python*'s `dict` [1] uses a maximum of ⅔. One can calculate the displacement from the hash, but we have to have a general way of telling if it's null. The lack of symmetry presents a difficulty removing entries.

Inline-chaining, as seen in Figure 1c, is, in many ways, a hybrid between the two. The dark circles represent the closed heads, and the outline the open stack, with a highlight to indicate the stack position. The expected probe-length is number of chained queries. Because of the `next` field, the space taken is one index *per* entry more then open-addressing. Being chained offers a higher load-factor, but it is not possible to exceed one: like open-addressing, the hash-table is contained in one block of memory.

3 PERFORMANCE

Because the closed and open entries are orthogonal for inline-chaining, the limiting factor in the worst-case is the same as for separate-chaining, and it will have identical behaviour as long as the load-factor doesn't exceed one. In the average case, we do at least as much work by a constant factor; in addition to the steps required for chaining, we also have to sometimes also have to manage the stack. Moving the top of the stack involves finding the closed head of the top entry and iterating until the top. However, modification requires only copying one entry; we aren't concerned with the order of the stack, only the order of the next indices. However, inline-chaining does have the advantage of cache-locality.

Figure 2 benchmarks straight insertion on a closed separately-chained set like Figure 1a, an inline-chained set like Figure 1c, and a `std::unordered_set`. The data is a pointer to a randomly generated set of non-sense names, formed out of syllables, Poisson-distributed up to 15-letters in length; a `char[16]` with a null-terminator. Hashed by *djb2* to a 64-bit `size_t`.

The same data is used for each replica across different methods, thus the same number of duplicates were ignored. The hash-tables were then destroyed for the next replica. The data is in a memory pool. Pre-allocation of the nodes of the separately-chained is done in an array; this is not counted towards the run-time; for the `unordered_set`, however, this is transparent, and is timed. For the inline-chaining, it's behaviour with respect to allocation is like open-addressing: it is all contained it in one array.

4 IMPLEMENTATION

This section talks about the specific implementation of inline-chained hash-table whose results are shown in Figure 2 as a map from star name strings to distance double-precision floating point.

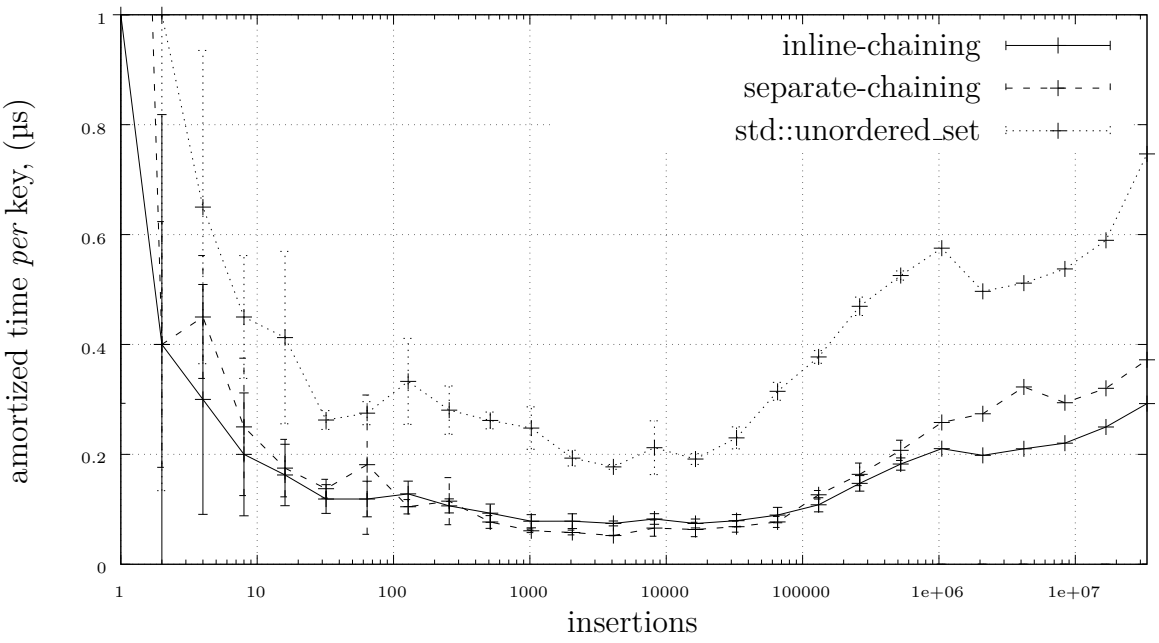


Figure 2: A comparison of chained techniques in a C^{++} benchmark with a log-x scale.

4.1 Next Entry

The `next` field in offers a convenient place to store out-of-band information without imposing restrictions on the key; specifically, we do not assume that it is non-zero. There are two special values that must be differentiated from the indices: there is no closed value associated with this address, called `NULL`, and there is no next value in the bucket, called `END`.

Since the `next` field must store the range of addressable buckets, minus one for itself, this is one short of representing the whole range. Since the implementation uses power-of-two resizes, it causes the addressable space to be one-bit less; we waste nearly a bit, half the size, or the equivalent of a signed integer. The default is a `size_t`, but in Figure 1, for illustration, the `hash` is 8 bits. Therefore the addressable space by `next` is $[0, 127]$, and `NULL`, `END` are 128, 129.

4.2 Load Factor

It is possible to get rid of load factor calculations by making the maximum load factor identically one. Only if the hash-table is full does one resize. In an inline-chained hash-table, the entries and the next index are overlapped; this means that the capacity for each goes up in the same allocation. This simplifies the design.

4.3 Power-of-Two

During the rehashing phase, closed entries have an expected value of,

$$E[\text{will move}] = \frac{\text{old capacity}}{\text{new capacity}}$$

that they will move. This can be seen as a form of consistent hashing[8], and was a consideration when designing the hash-table with power-of-two resizes. Instead of swapping moved entries with open entries, something that could take $\mathcal{O}(n^2)$, we rehash all the open entries.

4.4 Stack and Maintaining Orthogonality

We can tell whether an element is of the stack by it being open; if the address and the address given by the hash function does not match, it is not the head of the bucket. To keep track of the open entries, we place them on top of a stack formed from unused buckets. In this way, only one parameter needs to be added for the table: the `top` of the stack. The stack has to yield to a new entry and replace a deleted entry that is in it's range. This involves moving the stack pointer up and down. For that, we need to take the front closed element in the bucket with the `top`'s address and iterate until one before the `top`.

It is convenient to grow the stack from the back. That way, when rehashing the stack, a stack entry never conflicts with another. The alternative would allow $\mathcal{O}(1)$ low-numbered stack items to keep their place, but at a much more complex copying procedure.

The stack jumps over occupied entries, but it is bounded by $\mathcal{O}(n)$ in n inserts. A slight subtlety is that this amortization is not valid if inserting *and* deleting. Despite $\mathcal{O}(n)$ worst-case performance anyway, it is useful to minimize this as much as possible.

Since the `top` is an address, and the addresses only go to half the available space, we have a bit to spare. This has been used for a lazy stack. In this way, repeatedly adding and deleting to the open stack just sets the lazy bit. Only when one deletes twice does a move get forced.

4.5 Inverse Hash Function

If the hash function forms a bijection between the range in the space where elements live and the image in hash-space, the keys are not stored in the hash-table at all. They are generated from the hashes using an inverse-mapping. This can be the case when the items being hashed are discrete, like an `enum`, or a discrete integer set.

4.6 Iteration

Inline-chaining is generally good for iteration. For implementations that do not provide a special iteration mechanism, iteration on separate-chaining is $\mathcal{O}(\text{capacity} + \text{size})$. For open-addressing and inline-chaining, because we store collisions in the hash-table itself, it's $\mathcal{O}(\text{capacity})$. However, a practical capacity in open-addressing will be smaller than inline-chaining because the decrease in performance as the load-factor reaches saturation.

5 CONCLUSION

This specific data for the average use-case shows the difference between separate-chaining and inline-chaining is not great enough to matter to performance, and even helps in some cases. For situations where one needs a new hash-table, the simplicity of inline-chaining is appealing.

REFERENCES

[1] D. Knuth, *The Art of Computer Programming, Sorting and Searching*, vol. 3. Addison-Wesley, Reading, Massachusetts, 2 ed., 1998.

[2] F. A. Williams, "Handling identifies as internal symbols in language processors," *Communications of the ACM*, vol. 2, no. 6, pp. 21–24, 1959.

[3] R. P. Brent, "Reducing the retrieval time of scatter storage techniques," *Communications of the ACM*, vol. 16, pp. 105–109, Feb 1973.

[4] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[5] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, pp. 130–146, April 1957.

[6] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pp. 281–288, IEEE, 1985.

[7] S. S. Skiena, *The algorithm design manual*, vol. 2. Springer, 2008.

[8] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, (New York, NY, USA), p. 654–663, Association for Computing Machinery, 1997.