Representing Sets with Minimal Space: Jamie's Trit

James Dow Allen

August 16, 2020

Abstract

The theoretical minimum storage needed to represent a set of size N drawn from a universe of size M is about $N \cdot (\log_2(M/N) + 1.4472)$ bits (assuming neither N nor M/N is very small). I review the technique of 'quotienting' which is used to approach this minimum, and look at the actual memory costs achieved by practical designs. Instead of somehow implementing and exploiting 1.4472 bits of steering information, most practical schemes use two bits (or more). In the conclusion I mention a scheme to reduce the overhead cost from two bits to a single trit.

1 Common Nomenclature of Hash Tables

This paper focuses on a single specific topic in combinatorics, but to motivate the discussion we review the storage costs of memory-efficient hash tables. To discuss very different hash table methods using a common terminology, we will say that a hash table has K_C cabinets; that each cabinet has B drawers; and that each drawer has K_B bins. The total number of bins is $(B \cdot K_B \cdot K_C)$, but in the example systems we discuss either $K_B = 1$ or $K_C = 1$ or both, so we will define $K_T = K_B \cdot K_C$ and use K_T for a multiplicity factor throughout. To keep the discussion focused we ignore some issues (e.g. dynamic resizing) that arise in practical hash table design.

A hash table will store some subset S of a universe U. We will suppose the sizes of these sets are $N = |S|$ and $M = |U|$ and that these sizes are known in advance. Each element u of the universe U is considered a 'key'; the hash

table is presented with a 'key' and answers the question *Is u (the key under consideration) in the set S and if so, what is the associated 'payload' datum?* Although keys in many applications will be variable-length character strings, we will assume that $U = \{0, 1, 2, 3, 4, ..., M-1\}$; if $M = 2^m$ is a power of two, the keys will then be fixed-length bit strings of length m.

Thus for our purpose, a key u is a whole number. We will assume it is transformed into a 'quotient' q and 'remainder' r via the arithmetic

$$u' = h(u) = q \cdot B + r \tag{1}$$

where B is the number of drawers and $h(u)$ is a reversible hash function, e.g. $h(u) = (u \cdot 11669)\%65537$ which might be chosen when keys are 16-bit integers. (Here '% 65537' denotes the remainder after division by 65537. Some results $h(u)$ won't quite fit into 16 bits but this hardly matters, due to the division by B which immediately follows.) There will be a way to reconstruct the original key

$$u = h^{-1}(q \cdot B + r) \tag{2}$$

where $h^{-1}(u') = (u' \cdot 40505)\%65537$ for the example hash above. Such reversible hash functions are plentiful and easily found, as shown in the cited source code. While the 'quotienting' (partition of u into q and r) is essential to the entire paper, the hashing is inessential to our discussion, and the reader is welcome to assume use of the trivial $h(u) = u$.

The total number of bins (possible table entries) in a hash table will be $(K_T \cdot B)$ where K_T might be either the number (perhaps 20 to 50 or so) of bins per drawer in a method like Tight Chained Hash Tables, or the number of cabinets (perhaps 3 or 4) each with their own drawers as in a Cuckoo Hash Table, or $K_T = 4 \cdot 2 = 8$ in a Cuckoo Hash design with 4 bins per drawer and 2 cabinets. $K_T = 1$ for Open-Address Hash Tables.

Since each stored element uses one bin, the number of stored elements in the hash table is $(N = L \cdot K_T \cdot B)$ where L is the 'occupancy rate' or 'load factor'. $L = 0.80$ is a typical value, but designers might prefer $L < 0.75$ to maximize speed, or $L > 0.85$ to minimize storage cost.

We will find that the hash table will require a total of $(N/L) \cdot (\log_2(M/N) + \log_2(L \cdot K_T) + p + s)$ bits to store the subset and its payloads. Here p is the individual payload size in bits, and s is any other required overhead.

When tuning a method's parameters you will need to estimate the occupancy rate L, but for our purpose in this paper we assume different methods

have similar occupancy rates L and we can compare their memory performances most simply with the assumption $L = 1$. Note that the open-address method will work with occupancy as high as $L = 1$: it will just be extremely clumsy and slow. We can also ignore the payload cost (p bits) since this will be the same in different methods. With these simplifications, the cost formula becomes quite simple: The net cost to store a quotient in a hash table is $(\log_2(M/N) + \log_2(K_T) + s)$ bits *per stored element*. In the Cuckoo Hash, K_T is smallish but never less than 2, and $s = 0$. In Allen's Tight Hash, $K_T \approx 30$ and $s \approx 0.4$ are typical parameters. In Cleary's Compact Hash, $K_T = 1$ and $s = 2$. The two bits of overhead in Cleary's scheme are called the 'virgin' and 'change' bits and will be described below.

In Section 2 we derive the theoretical minimum storage cost. In Section 3 we mention five types of table, and show the memory cost in each case. In Section 4 we mention some applications for hash tables and mention ways to reduce the storage cost. Section 5 takes a look at efficient storage when we don't need fast look-up. Section 6 shows a way to reduce the steering overhead from the two bits used in Section 5 to a single trit.

2 Theoretical Minimum Storage Cost

There are $C_N^M = \frac{M!}{N!(M-N)!}$ ways to choose N elements from a universe of size M, so the informational cost to depict an arbitrary N-sized subset (S) is given in bits by $\log_2(M!) - \log_2(N!) - \log_2((M-N)!)$. We will assume that there is no payload to be stored.

Starting with Stirling's approximation, $\ln(K!) \approx (K + 0.5) \cdot \ln(K) - K + 0.919$; we note that some terms will cancel and some will be too small to matter; so $\log_2(K!) \approx K \cdot \log_2(K)$ is good enough for our purpose. With this we find that the storage cost *per element* is given in bits by

$$\frac{StorageCost}{N} \approx \log_2(\frac{M}{N}) - (\frac{M}{N} - 1) \cdot \log_2(1 - \frac{N}{M}) \tag{3}$$

When $M \approx 2N$ this shows a cost of approximately 2 bits per stored element (i.e. 1 bit per element in the universe), which can be achieved with a simple bit-map: allocate M bits and set the u'th bit to 1 if and only if $u \in S$. When $M < 2N$ it will be better to store the elements of U which are *not* in S. (This won't work if there is payload data that needs to be stored.) But when M is larger than $2N$ the cost shown above rapidly approaches a

simple asymptote. Using the asymptote $\log_2(1-\epsilon) \rightarrow -1.4427 \cdot \epsilon$ (where $1.4427 \approx \log_2(2.71828)$) the equation above reduces to

$$\frac{StorageCost}{N} \approx \log_2(\frac{M}{N}) + 1.4427 \cdot (\frac{M-N}{M}) < \log_2(\frac{M}{N}) + 1.4427 \quad (4)$$

This upper bound is also an excellent approximation for typical values of M and N. $\log_2(M/N)$ is simply the cost to store the quotient q in an Open Address scheme. 1.4427 bits (1 nat) can be called a 'steering overhead': it can be viewed as information which steers an algorithm to recover the remainder r. We've mentioned above that the elemental storage cost in Cleary's Compact Hash is $\log_2(M/N)+2$; and we now see that this is nearly optimal. In Cuckoo Hash the cost is $\log_2(M/N)+\log_2(K_T)$; the use of $K_T \geq 3$ eliminates need for any steering overhead.

3 Types of Hash Tables

Simple List

When access speed is not important, a simple list of keys will suffice to store a subset. To minimize memory we will want to store just the quotients instead of the entire keys. This will require a side channel of 'steering information' which is discussed in detail in Section 5.

Open-Address Tables: Cleary's Compact Hash

'Open-addressing' is the most popular and simplest hash table method: storage for one entry is provided in each drawer. When inserting a new entry for which the corresponding drawer is already otherwise occupied, nearby drawers are probed in a prescribed sequence until a matching entry or an empty drawer is found.

Normally 'quotienting' to reduce the cost of storing the key is unavailable with this method, but Cleary was able to achieve quotienting in an open-address scheme (Cleary Compact Hash tables) by providing two 'steering bits,' in each drawer: a 'virgin' bit (set initially and cleared when some inserted element has remainder r equal to that drawer's index) and a 'change' bit (set when the element physically in a drawer has a different index from its left-adjacent element).

This method requires that the code search a neighborhood about the indexed entry in order to reconstruct the indexes; and sometimes already-placed elements must be moved. This makes it difficult to construct a stable tag, but Darragh-Cleary have a workaround (see below), using 4 or 5 extra bits to construct such a tag.

All hash tables also require an 'empty bit' to distinguish drawers which are physically empty, but this seldom actually costs memory. For example, if the stored quotient can range from $0 \leq q < 4090$, 12 bits will be allocated for quotients, and a left-over value ($q = 4095$) can be used as an empty indicator. The designer would have to be very unlucky for *both* the number of possible payloads and the number of possible quotients to be exact powers of two. (In practice, the coder would probably store $q + 1$ as the quotient instead of q, so that all zero-bits becomes the empty indicator.)

Chained-Address Tables: Allen's Tight Hash

In chained-address schemes, no reprobing is done. Instead each drawer has an associated list of bins; and that list will grow as needed to accommodate all insertions whose keys map to that drawer. 'Quotienting' is automatic: the drawer index doesn't change. There are various implementations of chained-address hash tables; we focus on one due to the author, very similar to a method due to Valmari.

In Tight Hash, there are a fixed number of bins allocated to each drawer, e.g. 24 bins in a main bin group, and 4 bins in each of two secondary bin groups. When the bins in a drawer are exhausted, a bin group from a neighboring drawer is used. Because of severe restrictions on the bin-group linkages, only about 4 bits are spent on a 'quasi-pointer' for each bin-group, for a net cost of less than 0.4 bits per bin. A typical K_T might be $K_T = 32$ in Allen's approach. The size in bits for (quotient + quasi-pointers) is $\log_2(M/N) + \log_2(K_T) + 0.4$.

Alternate-Address Tables (Cuckoo)

Cuckoo hash tables are sometimes classed as 'Open Address' but their principles are too different to condone that terminology. In the cuckoo system, two or more cabinets are provided and different hashing functions $h(u') \rightarrow (q, r)$ are provided for the different cabinets. (In my implementation I achieve this

by simply using different but nearly equal divisors: $B_1, B_2, B_3, ...$ I ensure that these are all prime numbers but this is non-essential.)

Searching for a given key may need to examine each cabinet. A new entry is inserted into whichever cabinet has an empty drawer at the appropriate position. When no empty drawer is found, the new entry is inserted anyway, dislodging whichever entry was already there. (Much like a cuckoo bird commandeering another bird's nest! For improved speed, 2 or more bins may be provided in each drawer, but our focus is on memory cost.) The dislodged entry will then be inserted elsewhere, perhaps doing another dislodgement, and so on. There is no overhead. The size in bits for the quotient is $\log_2(M/N) + \log_2(K_T)$.

Indexed Tables

If the key is m bits and you don't want to bother with quotienting you can still save, say, 8 bits in each table entry by using the 8 high-order bits of the key to select one of 256 smallish hash tables (of any type). Only $(m - 8)$ key bits need be stored now. (This scheme breaks down if extended to give individual tables with very low populations. In the extreme case you will expend bits just to denote that some tables are completely empty!) Key storage is reduced by $\log_2(K_T)$ bits; this is $\log_2(256) = 8$ in the example. This technique is a "poor man's quotienting."

Hash Tree

In a digital search tree ('trie'), a node may contain a list of pointers to its children, along with any other needed payload. The trie can be organized as a hash table; instead of storing pointers, we can use an implicit pointer as a key to access another entry in the table. If that entry exists (i.e. it is occupied and its quotient matches the quotient of the implicit pointer) then the node is discovered with no bits wasted on a pointer! The only cost here is the stored key, or rather its quotient. The key will comprise a tag to denote the parent node uniquely, and a child index of $\log_2 J$ bits where J is the number of child pointers that can arise conceptually from the parent node.

If nodes are never moved after initial insertion then the bin number, denoted with $\log_2(B + K_T)$ bits, can be used for the tag. (This suffices with Allen's Tight Hash: nodes are sometimes relocated slightly but this is

detected on each search.) Nodes are relocated in Cleary's Compact Hash; the Darragh-Cleary Compact Hash Tree circumvents the problem by using $\log_2(B \cdot W)$ bits for the tag where B is the number of drawers and W is a hoped-for upper bound on the number of elements that can be mapped to the same drawer. (These elements are maintained in insertion order, so the index $0 \leq w < W$ is reconstructed in each search. Darragh-Cleary suggest $W = 16$ but mention $W = 32$ as an alternative to make the chance of failure almost infinitesimal.

Neither W nor J need be powers of two: The code will do arithmetic like $h(((w \cdot J) + j) \cdot B + r_{parent}) \rightarrow q' \cdot B + r'$; treating these fields as using a fixed whole number of bits is just for ease of exposition. Thus, $W \approx 25$ is available as a compromise between $W = 16$ and $W = 32$. Similarly, if $J = 45$ you don't really need to waste six bits to represent a child index: it will be the final product $L \cdot N \cdot W \cdot J$ that matters.

For the Darragh-Cleary method, the net cost of a quotient in bits is $\log_2 J + 4 + 2$ bits, where $4 = \log_2 W$ and 2 is for the Virgin and Change bits already mentioned. For Allen's Tight Hash Tree, the cost is $\log_2 J + \log_2(K_T) + 0.4$ which will be identical to the Darragh-Cleary cost when $K_T = 48$; or one bit less when $K_T = 24$. A compact hash tree cannot be implemented with Cuckoo Hash – there is no way to construct a stable tag.

4 Some Simple Applications of Hash Tables

Peg Solitaire

There are $M = 2^{33}$ possible positions in Peg Solitaire but, with a reversal trick, only some 13 million (N) positions need be stored to depict all solutions. This example is developed in the cited source code. 14 bits are used for each quotient in the Tight Chained Hash (where $K_T = 37$) and 11 bits for Cuckoo Hash (where $K_T = 4$). These are in agreement with the $\log_2(\frac{M}{N}) + \log_2(K_T)$ quotient cost shown above.

No payload bits need be allocated at all in this application. That a position was encountered and placed in the table at all denotes that a successful path to an end position exists. A brief loop through possible moves will discover the successful move.

Checkers-position Caching

Suppose a checkers-playing program uses 75 bits to represent an arbitrary checkers position, and wishes to store N (say, 500 million) positions in a cache. The designer may want that cache to store as many positions as possible; we may assume that the size (500 million) was chosen to use up almost all available memory. Obviously this designer wants the cache to use as few bits as possible for each position. Two bits may suffice for the 'payload' since it is probably enough to remember whether the position is Won, Drawn or Lost. (In fact, with only three possibilities to distinguish, the two bits of payload might be replaced with a single trit.)

Supposing $K_T \approx L \approx 1$, then $B \approx N$. Since 500 million is about 2^{29}, the 75-bit position code will be decomposed into a 29-bit remainder and a 46-bit quotient. The total storage for each entry might be 50 bits: 46 bits for the quotient, 2 bits for a payload, and 2 'steering' bits needed, in effect, to help us recover the key's remainder (and hence the key).

In this example 92% (46 bits out of 50) of the table is spent on quotient storage. A method first published by Zobrist will reduce this substantially. Simply discard all but 21 bits of the quotient! This produces a false match with probability 2^{-21}, or 0.0000477%, when a second key maps to a drawer which is already occupied, but reduces the total bits needed per element from 50 to 25. The designer must decide whether he'd rather store 500 million positions with about 100 of them erroneous, or store only 250 million positions, but with perfect reliability. (The 100 errors is calculated for $L = 0.90$, and would be even smaller for a more typical $L = 0.80$. If you can tolerate 3300 errors, get 625 million positions in the same space by storing just 16 quotient bits.)

Rubik's Cube Solutions

Here's another example where a very small payload suffices. To record the solution to Rubik's Cube from any starting position, a single trit per position suffices: Record the remainder after division by 3 of the number of moves needed to win. If the remainder is, for example, 1 (so that total number of moves might be 31) then search the results of possible moves until arriving at one with remainder 0. This move must be correct (it yields a remaining path to solution with only 30). The 0 remainder cannot correspond to a 33-move position because Rubik's Cube moves are reversible. No position moved to

from the win-in-31 position can require more than 32 moves: a single-move reversion to the win-in-31 is always available.

5 Storage as a simple list

Hash tables feature very fast look-up, but we sacrifice speed when pursuing the absolute minimum storage cost. To record a set of size N for later transmission, we can simply set $B = N$, reduce each element to its quotient and drawer index and sort the quotients by their drawer index. If we are lucky and each drawer has exactly one element, this yields an elemental cost of $\log_2(M/N)$ bits – no overhead. But in practice, some drawers will be empty while some have two or more items, so this simple method doesn't quite work.

We can correct this defect by sending the list of drawer populations on a separate channel. If, for example, the populations are (0, 2, 4, 0, 0, 1, 1, 0), the receiver can use this information to reconstruct that the first two items whose quotients are transmitted were in the 2nd drawer, the next four items were in the 3rd drawer, and so on. A simple way to represent a small whole number (k) with few bits is a 'Stone Age tally' of k consecutive 1-bits followed by a 0-bit as delimiter. For example (0, 2, 4, 0, 0, 1, 1, 0) becomes (0 110 11110 0 0 10 10 0). In this encoding of drawer sizes there is one 1-bit for each item and one 0-bit for each drawer. Since we've assumed number of items and number of drawers are equal, this yields a total of two bits per item. This is the same steering overhead we saw with Cleary's Compact Hash Table and indeed, although details are different, these bits can be mapped to Cleary's virgin and change bits.

There is no need to have $N = B$ exactly in this approach. If N/B is allowed to vary, the elemental cost will be $Cost = \log_2(M/N) + \log_2(N/B) + 1 + B/N$. By varying N/B we will be trading off new bits in the required quotient with the 0-bits used for each drawer in the size code, or vice versa. $Cost$ is minimized (at $\log_2(M/N) + 1.91393$ bits), when $N/B = \ln(2) = 0.69315$. We approach this minimum closely, with $Cost = \log_2(M/N) + 2$, when $N/B = 1$ or $N/B = 0.5$.

Of course the 1.91393 bits for steering overhead is not a *theoretical* minimum – it depends on the 'Stone Age tally' representation of populations which, albeit elegant, is not quite optimal theoretically. But assuming $N/B = 1$, and that the resultant population codes follow a Poisson distribution, it

might seem we can do no better than 1.8825 bits: the Shannon entropy of a Poisson variate with mean 1. Yet we *know* we should be able to approach 1.4427 bits. Before reading ahead, see if you can figure out what we're missing. (Hint: If there are multiple ways to represent the same list, then that redundancy denotes wasted information. An intuitive way to see this is that the choice of which to use among multiple representations can be used as a channel of extra information.)

6 Jamie's Trit

I have reviewed several methods to provide steering overhead of about 2 bits. Cuckoo hash with four cabinets is a straightforward example. Cleary Compact Hash tables use 2 bits of steering (but need another 4 bits to build a stable tag). My own Tight Chained Hash needs, in effect, about 5 or 6 bits for steering equivalent (but needs no extra bits for stable tagging). And the simple list of elements with a side-channel of drawer populations in 'Stone Age tally' format uses 2 bits of steering per element (or 1.91393 bits when $N/B = 0.69315$).

However the information-theoretical optimum is known to be 1.4427 bits ($\log_2(e)$). I have no idea how to manipulate a 'nat,' but manipulating a 'trit' isn't too difficult. Trits are frequently used implicitly in computer programming. For example, the checkers-playing example needs only a trit ({ Won, Drawn, Lost }) as its payload, and solitaire puzzles like Rubik's cube can also get by with just a trit as payload. Storing trits efficiently as bits is easy: Pack five trits $(t_0, t_1, t_2, t_3, t_4)$ into an octet (b) of bits with the arithmetic $b = 3^4t_4 + 3^3t_3 + 3^2t_2 + 3t_1 + t_0$. (This spends 1.6000 bits per trit compared with the theoretical 1.5850 bits.)

But how do we get the list representation overhead down from two bits to a single trit? I found myself pondering this question from time to time. One day it struck me that I was overlooking an important insight from information theory. Test yourself before reading on!

In the set listing system just presented with $N = B$, 18.4% of drawers will have a population of two, and another 8.0% will have a population of three or more. A pair has two different orderings, and if we don't take advantage of the bit specifying the chosen ordering, *we've just wasted a bit of information.* Recovering that bit from each bucket with a plural population will net us 0.264 bits on average. not quite enough it might seem to reduce the cost from

2 trits to a single trit, but recall that the 'Stone Age tally' was chosen for simplicity, not optimality. In fact, coding t as a trit on { Zero, One, Plurality } (and then, if needed, recoding $t-1$ recursively) conserves bits compared with the 'Stone Age tally.'

Once I recalled the insight that otherwise wasted information should be exploited, solution immediately struck me. Provide each drawer with a trit denoting the number of elements which map to that drawer: { 0, 1, 2+ }. Sort all the elements mapping to that drawer, except for the final one, into ascending sequence of their quotients, always placing the largest quotient into the penultimate position. For example, if there are five elements and their quotients are (13, 17, 19, 23, 29) a valid ordering would be (13, 17, 23, 29, 19). When the 'steering trit' tells us there are 2 or more elements in the list, a simple scan looking for the first order-reversal will locate the final element in that list.

The method will seldom be practical, and the savings too minuscule to bother with anyway, but it did seem neat to me since it ties directly to an important bound from information theory $Cost \approx \log_2(M/N) + \log_2(2.71828)$. I decided to use my boyhood nickname and to call this "Jamie's Trit."

7 References

- Methods for Memory-Efficient Hash Tables
  Allen, J.D. Source code for memory-efficient hash tables.
  http://james.fabpedigree.com/jdas_htab.tar.gz; 2020.

- Cleary Compact Hash
  Cleary, J.G. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers* C-33(9): 828-834; 1984.

- Compact Hash Tree
  Darragh J.J., Cleary J.G., Witten, I.H. Bonsai: a Compact Representation of Trees. *Software Practice and Experience* 23(3): 277-291; March 1993.

- Very Tight Hash Table
  Geldenhuys, J., Valmari, A. Nearly Memory-optimal Data Structure itshape ACM Digital Library.

- Zobrist Hashing
  Zobrist, A.L. A New Hashing Method with Application for Game Playing
  itshape Tech. Rep. 88, Comp.Sci. Dept., University of Wisconsin 1969.