6.897: Advanced Data Structures Spring 2012

Lecture 10 — March 20, 2012

Prof. Erik Demaine

1 Overview

In the last lecture, we finished up talking about memory hierarchies and linked cache-oblivious data structures with geometric data structures. In this lecture we talk about different approaches to hashing. First, we talk about different hash functions and their properties, from basic universality to k-wise independence to a simple but effective hash function called simple tabulation. Then, we talk about different approaches to using these hash functions in a data structure. The approaches we cover are basic chaining, perfect hashing, linear probing, and cuckoo hashing.

The goal of hashing is to provide a solution that is faster than binary trees. We want to be able to store our information in less than $O(u \lg u)$ space and perform operations in less than $O(\lg u)$ time.

In particular, **FKS hashing** achieves $O(1)$ worst-case query time with $O(n)$ expected space and takes $O(n)$ construction time for the static dictionary problem. **Cuckoo Hashing** achieves $O(n)$ space, $O(1)$ worst-case query and deletion time, and $O(1)$ amortized insertion time for the dynamic dictionary problem.

2 Hash Function

In order to hash, we need a *hash function.* The hash function allows us to map a universe $\mathcal{U}$ of u keys to a slot in a table of size m. We define three different four different hash functions: totally random, universal, k-wise independent, and simple tabulation hashing.

Definition 1. *A* hash function *is a map h such that*

$$h : \{0, 1, \ldots, u-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$

2.1 Totally Random Hash Functions

Definition 2. *A hash function h is* totally random *if for all $x \in \mathcal{U}$, independent of all y for all $y \neq x \in \mathcal{U}$,*

$$\Pr_h\{h(x) = t\} = \frac{1}{m}$$

Totally random hash functions are the same thing as the simple uniform hashing of CLRS [1]. However, with the given defintion, a hash function must take $\Theta(\lg m)$ to store the hash of one key

$x \in \mathcal{U}$ in order for it to be totally random. There are u keys, which mean in total, it requires $\Theta(u \lg m)$ bits of information to store a totally random hash function.

Given that it takes $\Theta(u \lg u)$ to store all the keys in a binary tree, $\Theta(u \lg m)$ essentially gives us no benefits. As a result, we consider some hash functions with weaker guarantees.

2.2 Universal Hash Functions

The first such hash function worth considering is the universal families and the strong unversal families of hash functions.

Definition 3. *A family of hash functions $\mathcal{H}$ is* universal *if for every $h \in \mathcal{H}$, and for all $x \neq y \in \mathcal{U}$,*

$$\Pr_h\{h(x) = h(y)\} = O(\frac{1}{m}).$$

Definition 4. *A set $\mathcal{H}$ of hash functions is said to be a* strong universal family *if for all $x, y \in \mathcal{U}$ such that $x \neq y$,*

$$\Pr_h\{h(x) = h(y)\} \leq \frac{1}{m}$$

There are two relatively simple universal families of hash functions.

Example: $h(x) = [(ax) \mod p] \mod m$ for $0 < a < p$

In this family of hash functions, p is a prime with $p \geq u$. And ax can be done by multiplication of by vector dot product. The idea here is to multiple the key x by some number a, take it modulo a prime p and then slot it into the table of size m.

The downside of this method is that the higher slots of the table may be unused if $p < m$ or more generally, if $ax \mod p$ is evenly distributed, than table slots greater than $p \mod m$ will have fewer entries mapped to them.

The upside is that this a hash function belonging to this universal family only needs to store a and p, which takes $O(\lg a + \lg p)$ bits.

Example: $h(x) = (a \cdot x) >> (\lg u - \lg m)$

This hash function works if m and u are powers of 2. If we assume a computer is doing these computations, than m and u being powers of 2 is reasonable. Here, the idea is to multiply a by x and then rightshift the resulting word. By doing this, the hash function uses the $\lg m$ high bits of $a \cdot x$. These results come from Dietzfelbinger, Hagerup, Katajainen, and Penttonen [2].

2.3 k-Wise Independent

Definition 5. *A family $\mathcal{H}$ of hash functions is* k-wise independent *if for every $h \in \mathcal{H}$, and for all distinct $x_1, x_2, \ldots, x_k \in \mathcal{U}$,*

$$\Pr\{h(x_1) = t_1 \& \cdots \& h(x_k) = t_k\} = O(\frac{1}{m^k}).$$

Even pairwise independent ($k = 2$) is already stronger than universal. A simple example of a pairwise independent hash is $h(x) = [(ax + b) \mod p] \mod m$ for $0 < a < p$ and for $0 \leq b < p$. Here, again, p is a prime greater than u.

There are other interesting k-wise independent hash functions if we allow $O(n^\epsilon)$ space. One such hash function presented by Thorup and Zhang has query time as a function of k [4]. Another hash function that takes up $O(n^\epsilon)$ space is presnted by Siegel [5]. These hash functions have $O(1)$ query when $k = \Theta(\lg n)$.

Example: Another example of a k-wise independent hash function presented by Wegman and Carter [3] is

$$h(x) = [(\sum_{i=0}^{k-1} a_i x^i) \mod p] \mod m.$$

In this hash function, the a_is satisfy $0 \leq a_i < p$ and $0 < a_{k-1} < p$. p is still a prime greater than u.

2.4 Simple Tabulation Hashing [3]

The last hash function presented is simple tabulation hasing. If we view a key x as a vector $x_1, x_2, \ldots, x_c$ of characters, we can create a totally random hash table T_i on each character. This takes $O(cu^{1/c})$ words of space to store and takes $O(c)$ time to compute. In addition, simple tabulation hashing is 3-independent. It is defined as

$$h(x) = T_1(x_1) \oplus T_2(x_2) \oplus \cdots \oplus T_c(x_c).$$

3 Basic Chaining

Hashing with chaining is the first implementation of hashing we usually see. We start with a hash table, with a hash function h which maps keys into slots. In the event that multiple keys would be hashed to the same slot, we store the keys in a linked list on that slot instead.

For slot t, let c_t denote the length of the linked list chain corresponding to slot t. We can prove results concerning expected chain length, assuming universality of the hash function.

Claim 6. *The expected chain length $E[C_t]$ is constant, since $E[C_t] = \sum_i [Pr[h(x_i) = t]] = \sum_i [O(1/m)] = O(n/m)$.*

where $\frac{n}{m}$ is frequently called the load factor. The load factor is constant when we assume that $m = \Theta(n)$. This assumption can be kept satisfied by doubling the hash table size as needed.

However, even though we have a constant expected chain length, it turns out that this is not a very strong bound, and soon we will look at chain length bounds w.h.p. (with high probability). We can look at the variance of chain length, analyzed here for totally random hash functions, but in general we just need a bit of symmetric in the hash function:

Claim 7. *The expected chain length variance is constant, since we know $E[C_t^2] = \frac{1}{m} \sum_s E[C_s^2] = \frac{1}{m} \sum_{i \neq j} Pr[h(x_i) = h(x_j)] = \frac{1}{m} m^2 O(\frac{1}{m}) = O(1)$.*

Therefore, $Var[C_t] = E[C_t^2] - E[C_t]^2 = O(1)$.

where again we have assumed our usual hash function properties.

3.1 High Probability Bounds

We start by defining what high probability (w.h.p.) implies:

Definition 8. *An event* E *occurs with high probability if* $Pr[E] \geq 1 - 1/n^c$ *for any constant* c.

We can now prove our first high probability result, for totally random hash functions, with the help of Chernoff bounds:

Theorem 9 (Expected chain length with Chernoff bounds)**.** $Pr[C_t > c\mu] \leq \frac{\exp{(c-1)\mu}}{(c\mu)^{c\mu}}$*, where* μ *is the mean.*

We now get our expected high probability chain length, when the constant $c = \frac{\lg n}{\lg \lg n}$

Claim 10 (Expected chain length $C_t = O(\frac{\lg n}{\lg \lg n})$)**.** *For the chosen value of* c*,* $Pr[C_t > c\mu]$ *is dominated by the term in the denominator, becoming* $1/(\frac{\lg n}{\lg \lg n})^{\frac{\lg n}{\lg \lg n}} = \frac{1}{2^{\frac{\lg n}{\lg \lg n}} \lg \lg n} \approx 1/n^c$

so for chains up to this length we are satisfied, but unfortunately chains can become longer! This bound even stays the same when we replace the totally random hash function assumption with either $\Theta(\frac{\lg n}{\lg \lg n})$-wise independent hash functions (which is a lot of required independence!), as found by Schmidt, Siegel and Srinivasan (1995) [11], or simple tabulation hashing [10]. Thus, the bound serves as the motivation for moving onto perfect hashing, but in the meantime the outlook for basic chaining is not as bad as it first seems.

The major problems of accessing a long chain can be eased by supposing a cache of the most recent $\Omega(\log n)$ searches, a recent result posted on Pătraşcu's blog (2011) [12]. Thus, the idea behind the cache is that if you are unlucky enough to hash into a big chain, then caching it for later will amortize the huge cost associated with the chain.

Claim 11 (Constant amortized access with cache, amortized over $\Theta(\lg n)$ searches)**.** *For these* $\Theta(\lg n)$ *searches, the number of keys that collide with these searches is* $\Theta(\lg n)$ *w.h.p. Applying Chernoff again, for* $\mu = \lg n$ *and* $c = 2$*, we get* $Pr[C_t \geq c\mu] > 1/n^\epsilon$ *for some* ϵ.

So by caching, we can see that the expected chain length bounds of basic chaining is still decent, to some extent.

4 FKS Perfect Hashing – Fredman, Komlós, Szemerédi (1984) [17]

Perfect hashing changes the idea of chaining by turning the linked list of collisions into a separate collision-free hash table. *FKS hashing* is a two-layered hashing solution to the static dictionary problem that achieves $O(1)$ worst-case query time in $O(n)$ expected space, and takes $O(n)$ time to build.

The main idea is to hash to a small table T with collisions, and have every cell T_t of T be a collision-free hash table on the elements that map to T_t. Using perfect hashing, we can find a collision-free hash function h_i from C_t to a table of size $O(C_t^2)$ in constant time. To make a query then we compute $h(x) = t$ and then $h_t(x)$.

Claim 12 (Expected linear time and space for FKS perfect hashing)**.** $E[space] = E\left(\sum C_t^2\right) = n^2 * O(1/m) = O\left(\frac{n^2}{m}\right)$.

If we let $m = O(n)$, we have expected space $O(n)$ as desired, and since the creation of each T_t takes constant time, the total construction time is $O(n)$.

Claim 13 (Expected # of collisions in C_t)**.** $E[\#collisions] = C_t^2 * O(1/C_t^2) = O(1) \leq \frac{1}{2}$

where the inequality can be satisfied by setting constants. Then for perfect hashing, $Pr[\#Collisions = 0] \geq \frac{1}{2}$. If on the first try we do get a collision, we can try another hash function and do it again, just like flipping a coin until you get heads.

The perfect hashing query is $O(1)$ deterministic and expected linear construction time and space, as we can see from the above construction. Updates, which would make the structure dynamic, are randomized.

4.1 Dynamic FKS – Dietzfelbinger, Karlin, Mehlhorn, Heide, Rohnert, and Tarjan (1994) [13]

The translation to dynamic perfect hashing is smooth and obvious. To insert a key is essentially two-level hashing, unless we get a collision in the C_t hash table, in which case we need to rebuilt the table. Fortunately, the probability of collision is small, but to absorb this, if the chain length C_t grows by a factor of two, then we can rebuild the C_t hash table, but with a size multiplied by a factor of 4 larger, due to the C_t^2 size of the second hash table. Thus, we will still have $O(1)$ deterministic query, but additionally we will have $O(1)$ expected update. A result due to Dietzfelbinger and Heide in [14] allows Dynamic FKS to be performed w.h.p. with $O(1)$ expected update.

5 Linear probing

Linear probing is perhaps one of the first algorithms for handling hash collisions that you learn when initially learning hash tables. It is very simple: given hash function h and table size m, an insertion for x places it in the first available slot $h(x) + i \mod m$. If $h(x)$ is full, we try $h(x) + 1$, and $h(x) + 2$, and so forth. It is also well known that linear probing is a terrible idea, because "the rich get richer, and the poorer get poorer"; that is to say, when long runs of adjacent elements develop, they are even more likely to result in collisions which increase their size.

However, there are a lot of reasons to like linear probing in practice. When the runs are not too large, it takes good advantage of cache locality on real machines (the loaded cache line will contain the other elements we are likely to probe). There is some experimental evidence that linear probing imposes only a 10% overhead compared to normal memory access. If linear probing has really bad performance with a universal hash function, perhaps we can do better with a hash function with better independence guarantees.

In fact, it is an old result that with a totally random hash function h, we only pay $O(1/\epsilon^2)$ expected time per operation while using $O((1+\epsilon)n)$ space [6]. If $\epsilon = 1$, this is $O(1)$ expected time with only double the space (a luxury in Knuth's time, but reasonable now!) In 1990, it was shown that $O(\lg n)$-wise independent hash functions also resulted in constant expected time per operation [7].

The breakthrough result in 2007 was that we in fact only needed 5-independence to get constant expected time, in [8] (updated in 2009). This was a heavily practical paper, emphasizing machine implementation, and it resulted in a large focus on k-independence in the case that $k = 5$.

At this time it was also shown that 2-independent hash functions could only achieve a really bad lower bound of $\Omega(\lg n)$ expected time per operation; this bound was improved in 2010 by [9] showing that there existed some 4-independent hash functions that also had $\Omega(\lg n)$ expected time (thus making the 5-independence bound tight!)

The most recent result is [10] showing that simple tabulation hashing achieves $O(1/\epsilon^2)$ expected time per operation; this is just as good as totally random hash functions.

OPEN: In practical settings such as dictionaries like Python, does linear probing with simple tabulation hashing beat the current implementation of quadratic probing?

5.1 Linear probing and totally random hashing

It turns out the proof that given a totally random hash function h, we can do queries in $O(1)$ expected time, is reasonably simple, so we will cover it here [8]. The main difficulty for carrying out this proof is the fact that the location some key x is mapped to in the table, $h(x)$, does not necessarily correspond to where the key eventually is mapped to due to linear probing. In general, it's easier to reason about the distribution of $h(x)$ (which is very simple in the case of a totally random hash function) and the distribution of where the keys actually reside on the table (which has a complex dependence on what keys were stored previously). We'll work around this difficulty by defining a notion of a "dangerous" interval, which will let us relate hash values and where the keys actually land.

Theorem 14. *Given a totally random hash function h, a hash table implementing linear probing will perform queries in $O(1)$ expected time*

Proof. Assume a totally random hash function h over a domain of size n, and furthermore assume that the size of the table $m = 3n$ (although this proof generalizes for arbitrary $m = (1+\epsilon)n$; we do this simplification in order to simplify the proof). For our analysis, we will refer to an imaginary perfect binary tree where the leaves correspond to slots in our hash table (similar to the analysis we did for ordered file maintenance.) Nodes correspond to ranges of slots in the table.

Now, define a node of height h (i.e. interval of size 2^h) to be "dangerous" if the number of keys in the table which hash to this node is greater than $\frac{2}{3}2^h$. A dangerous node is one for which the "density" of filled slots is high enough for us to be worried about super-clustering. Note that "dangerous" only talks about the hash function, and not where a key ends up living; a key which maps to a node may end up living outside of the node. (However, also note that you need at most 2^{h+1} keys mapping to a node in order to fill up a node; the saturated node will either be this node, or the adjacent one.)

Consider the probability that a node is dangerous. By assumption that $m = 3n$, so the expected number of keys which hash to a single slot is 1/3, and thus the expected number of keys which hash to slots within a node at height h, denoted as X_h, is $E[X_h] = 2^h/3$. Denote this value by μ, and note that the threshold for "dangerous“ is 2μ. Using a Chernoff bound we can see $\Pr[X_h > 2\mu] \leq e^\mu/2^{2\mu} = (e/4)^{2^h/3}$. The key property about this probability is that it is double exponential.

At last, we now relate the presence of run in tables (clustering) to the existence of dangerous nodes. Consider a run in table of length $\in [2^l, 2^{l+1})$ for arbitrary l. Look at the nodes of height $h = l - 3$ spanning the run; there are at least 8 and at most 17. (It is 17 rather than 16 because we may need an extra node to finish off the range.) Consider the first four nodes: they span $> 3 \cdot 2^h$ slots of the run (only the first node could be partially filled.) Furthermore, the keys occupying the slots in these nodes must have hashed within the nodes as well (they could not have landed in the left, since this would contradict our assumption that these are the first four nodes of the run.) We now see that at least one node must be dangerous, as if all the nodes were not dangerous, there would be less than $< 4 \cdot \frac{2}{3} \cdot 2^h = \frac{8}{3} \cdot 2^h$ occupied slots, which is less than the number of slots of the run we cover ($\frac{9}{3} \cdot 2^h$).

Using this fact, we can now calculate an upper bound on the probability that given x, a run containing x has length $\in [2^l, 2^{l+1}]$. For any such run, there exists at least one dangerous node. By the union bound over the maximum number of nodes in the run, this probability is $\leq 17\Pr[\text{node of height } l - 3 \text{ is dangerous}] \leq 17 \cdot (e/4)^{2^h/3}$ So the expected length of the run containing x is $\Theta(\quad _{l} 2^l \Pr[\text{length is } \in [2^l, 2^{l+1})]) = \Theta(1)$, as desired (taking advantage of the fact that the inner probability is one over a doubly exponential quantity). □

If we add a cache of $\lg^{n+1} n$ size, we can achieve $O(1)$ amortized with high probability [10]; the proof is a simple generalization of the argument we gave, except that now we check per batch whether or not something is in a run.

6 Cuckoo Hashing – Pagh and Rodler (2004) [15]

Cuckoo hashing is similar to double hashing and perfect hashing. *Cuckoo hashing* is inspired by the Cuckoo bird, which lays its eggs in other birds' nests, bumping out the eggs that are originally there. Cuckoo hashing solves the dynamic dictionary problem, achieving $O(1)$ worst-case time for queries and deletes, and $O(1)$ expected time for inserts.

Let f and g be $(c, 6 \log n)$-universal hash functions. As usual, f and g map to a table T with m rows. But now, we will state that f and g hash to two separate hash tables. So $T[f(x)]$ and $T[g(x)]$ refer to hash entries in two adjacent hash tables. The cuckoo part of Cuckoo hashing thus refers to bumping out a keys of one table in the event of collision, and hashing them into the other table, repeatedly until the collision is resolved.

We implement the functions as follows:

- *Query(x)* – Check $T[f(x)]$ and $T[g(x)]$ for x.
- *Delete(x)* – Query x and delete if found.
- *Insert(x)* – If $T[f(x)]$ is empty, we put x in $T[f(x)]$ and are done.

Otherwise say y is originally in $T[f(x)]$. We put x in $T[f(x)]$ as before, and bump y to whichever of $T[f(y)]$ and $T[g(y)]$ it didn't just get bumped from. If that new location is empty, we are done. Otherwise, we place y there anyway and repeat the process, moving the newly bumped element z to whichever of $T[f(z)]$ and $T[g(z)]$ doesn't now contain y.

We continue in this manner until we're either done or reach a hard cap of bumping $6 \log n$ elements. Once we've bumped $6 \log n$ elements we pick a new pair of hash functions f and g and rehash every element in the table.

Note that at all times we maintain the invariant that each element x is either at $T[f(x)]$ or $T[g(x)]$, which makes it easy to show correctness. The time analysis is harder.

It is clear that query and delete are $O(1)$ operations. The reason Insert(x) is not horribly slow is that the number of items that get bumped is generally very small, and we rehash the entire table very rarely when m is large enough. We take $m = 4n$.

Since we only ever look at at most $6 \log n$ elements, we can treat f and g as random functions. Let $x = x_1$ be the inserted element, and $x_2, x_3, \ldots$ be the sequence of bumped elements in order. It is convenient to visualize the process on the *cuckoo graph*, which has verticies $1, 2, \ldots, m$ and edges $(f(x), g(x))$ for all $x \in S$. Inserting a new element can then be visualized as a walk on this graph. There are 3 patterns in which the elements can be bumped.

- *Case 1* Items $x_1, x_2, \ldots, x_k$ are all distinct. The bump pattern looks something like[1]

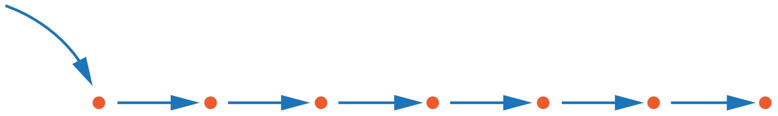
  

  Image by MIT OpenCourseWare.

  The probability that at least one item (ie. x_2) gets bumped is

  $$Pr(T[f(x)]\text{is occupied}) = Pr(\exists\, y : f(x) = g(y) \vee f(x) = f(y)) < \frac{2n}{m} = \frac{1}{2}.$$

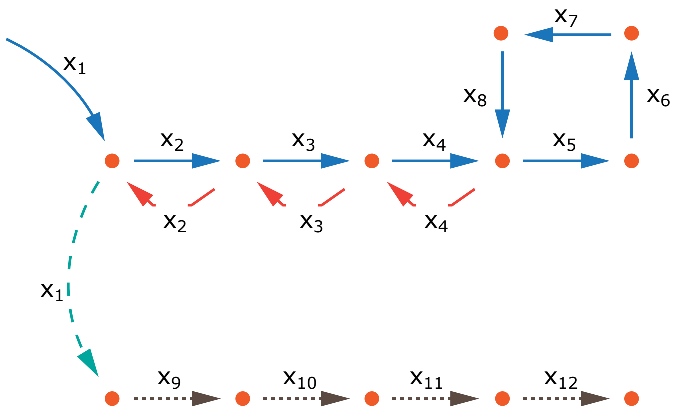
  The probability that at least 2 items get bumped is the probability the first item gets bumped ($< 1/2$, from above) times the probability the second item gets bumped (also $< 1/2$, by the same logic). By induction, we can show that the probability that at least t elements get bumped is $< 2^{-t}$, so the expected running time ignoring rehashing is $< \quad {}_t\, t2^{-t} = O(1)$. The probability of a full rehash in this case is $< 2^{-6 \log n} = O(n^{-6})$.

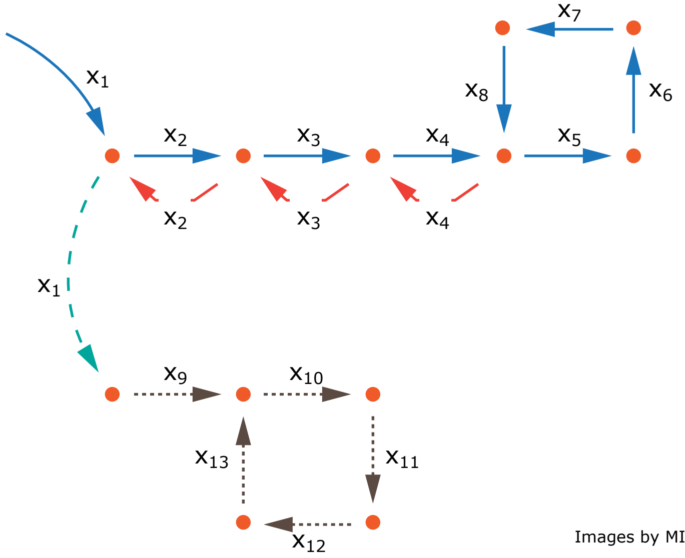
- *Case 2* The sequence $x_1, x_2, \ldots, x_k$ at some point bumps an element x_i that has already been bumped, and $x_i, x_{i-1}, \ldots, x_1$ get bumped in turn after which the sequence of bumping continues as in the diagram below. In this case we assume that after x_1 gets bumped all the bumped elements are new and distinct.

  The length of the sequence (k) is at most 3 times $\max\{\#\text{solid arrows}, \#\text{dashed arrows}\}$ in the diagram above, which is expected to be $O(1)$ by Case 1. Similarly, the probability of a full rehash is $O(2^{\frac{-6 \log n}{3}}) = O(n^{-2})$.

- *Case 3* Same as Case 2, except that the dotted lines again bump something that has been bumped before (diagram on next page).

[1] Diagrams courtesy of Pramook Khungurn, Lec 1 scribe notes from when the class was taught (as 6.897) in 2005





Images by MIT OpenCourseWare.

In this case, the cost is $O(\log n)$ bumps plus the cost of a rehash. We compute the probability Case 3 happens via a counting argument. The number of Case 3 configurations involving t distinct x_i given some x_1 is ($\leq n^{t-1}$ choices for the other x_i) $\cdot$ ($< t^3$ choices for the index of the first loop, where the first loop hits the existing path, and where the second loop hits the existing path) $\cdot$ (m^{t-1} choices for the hash values to associate with the x_i) $= O(n^{t-1}t^3m^{t-1})$.

The total number of configurations we are choosing from is $\genfrac{}{}{0pt}{}{m}{2}^{t} = O(2^{-t}m^{2t})$, since each x_i corresponds to a possible edge in the cuckoo graph. So the total probability of a Case 3 configuration (after plugging in $m = 4n$) is

$$\sum_t \frac{O(n^{t-1}t^3m^{t-1})}{O(2^{-t}m^{2t})} = O(n^{-2})\sum_t \frac{t^3}{2^{3t}} = O(n^{-2}).$$

If there is no rehash, the cost of insertion is $O(1)$ from Cases 1 and 2. The probability of a rehash is $O(n^{-2})$. So, we have an insertion taking time $Pr(\text{Rehash})\cdot(O(\log n)+n\cdot\text{Insert})+(1-Pr(\text{Rehash}))\cdot O(1) = O(1/n)\cdot\text{Insert}+O(1)$, so overall the cost of an insertion is $O(1)$ in expectation as desired. Thus, with Cuckoo hashing we have $(2+\epsilon)n$ space, and 2 deterministic probes per query.

6.1 Claims

With either totally random or $O(\lg n)$-wise independence, we get $O(1)$ amortized expected update, and $O(1/n)$ build failure probability, which is the chance that your keyset will be completely unsustainable with the current Cuckoo hash table, at which point you would have to start over and rebuild [15].

6-wise independence is insufficient for Cuckoo hashing to get $O(1)$ expected update, with a build failure probability of $1 - 1/n$, which is quite bad. This result is shown by Cohen and Kane (2009) in [16].

With simple tabulation hashing, the build failure probability becomes $\Theta(1/n^(1/3))$, which can be found in [10].

Theorem 15 (Constant expected update, for totally random hash functions)**.** *Pr[Insert follows bump path of length k]* $\leq 1/2^k$

For two hash functions g *and* h*, where each has* n *values, and each of these* n *values has* $\lg m$ *bits. Thus we need* $2n \lg n$ *bits to encode* g *and* h*.*

Claim 16 (Encoding hash functions in $2n \lg(n) - k$ time)**.**

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2009.

[2] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19-51, 1997.

[3] Mark N.Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265-279, 1981.

[4] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. *SODA*, pages 615-624, 2004.

[5] Alan Siegel. On universal classes of extremely random constant-time high functions. *SIAM-COMP*, 33(3):505-543, 2004.

[6] Don Knuth. Notes on "open" addressing, 1963.

[7] Jeanette P. Schmidt and Alan Siegel. The Spatial Complexity of Oblivious k-Probe Hash Functions. *SIAM Journal on Computing*, 19(5):775-786, 1990.

[8] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. In *In STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318-327. ACM Press, 2007.

[9] Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *Proc. 37th International Colloquium on Automata Languages and Programming (ICALP)*, pages 715-726, 2010.

[10] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. In *Proc. 43rd ACM Symposium on Theory of Computing (STOC)*, pages 1-10, 2011. See also arXiv:1011.5200.

[11] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223-250,1995.

MIT OpenCourseWare
http://ocw.mit.edu

6.851 Advanced Data Structures
Spring 2012

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

[12] Mihai Pătraşcu. Blog, 2011.

[13] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738-761, 1994.

[14] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP*, pages 6-19, 1990.

[15] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122-144, 2004.

[16] Jeffrey S. Cohen and Daniel M. Kane. Bounds on the independence required for cuckoo hashing. *ACM Transactions on Algorithms*, 2009.

[17] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538-544, 1984.