Lecture 11: Perfect Hashing

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Hashing Review

The point of hashing is

- to get fast ($O(1)$ on average) insert and search
- to use storage space much smaller than the space of all possible key values, closer to the optimum of just enough space for the data.

A hash table is essentially an array.

Put the key k into position $h(k)$ if that cell is empty.

If cell $h(k)$ is not empty (a collision) either put it

- somewhere else ("open addressing"), or
- into a linked list at position $h(k)$ ("chaining").

Subtleties

For open addressing:

- need a collision resolution strategy
- Deletion: need lazy deletion to not disrupt search path (but table fills up)
- Rehashing required for full tables, which is costly, so suboptimal for data sets that grow and shrink.

For chaining:

- Insertion: (-) overhead of maintaining list, (+) still $O(1)$ to insert.
- Retrieval: (-) $O(n)$ search of list, (+) lists are short for a good hash function.
- Deletion: (-) $O(n)$ search of list, plus list management overhead, (+) retrieval speed not impeded by real deletes.

Perfect Hashing

If the set of keys is *static* (i.e. there will never be any insertions or deletions), we can design the hash function to get a worst-case search $= O(1)$. This technique is called *perfect hashing*.

Example of static data: consider the set of file names on a CD-ROM. Or: the set of reserved words in a programming language.

Ideally with perfect hashing there are *no* collisions. If we can randomly generate a hash function that gives a collision infrequently, then we can generate new hash functions until there are no collisions.

Universal hashing with big table

Universal hashing, on average, will produce a collision between two random keys $1/m$ of the time, for table size m.

What's the probability of at least one collision between n keys?

There are:

$$\begin{aligned}\binom{n}{2} &= \frac{n!}{(n-2)!2!} \\ &= \frac{n(n-1)(n-2)(n-3)\cdots 1}{(n-2)(n-3)\cdots 1 \cdot 2 \cdot 1} \\ &= \frac{n(n-1)}{2}\end{aligned}$$

possible pairs of keys.

Universal hashing with big table

Each pair has a probability of collision of $1/m$. The probability of at least one collision for a random universal hash function is:

$$\begin{aligned} Pr(\#\text{collisions} \geq 1) &= 1 - Pr(\#\text{collisions} = 0) \\ &= 1 - \left(1 - \frac{1}{m}\right)^{n(n-1)/2} \\ &= 1 - \left(\frac{m-1}{m}\right)^{n(n-1)/2} \end{aligned}$$

How big should we make m if we want the probability of a collision to be low?

Probability of a collision

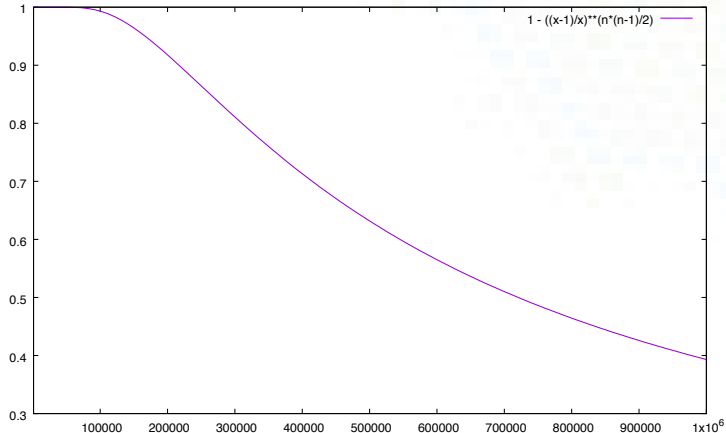


Figure: Probability of a collision when n=1000. m is along x-axis

Choosing m

If we have $n = 1000$ and choose $m = 100000$, then $Pr((\#\text{collisions} \geq 1) = 0.9932$.

If we choose 100 random functions, what's the probability that *all* of them have a collision?

$$0.9932^{100} = 0.51$$

So if we choose 100 random functions, then there is a 50% chance that one of them has no collisions. We would need to choose 340 before the probability dropped below 10%.

And that's for an occupancy rate of 1% - not a great way of choosing a perfect hashing function.

Hash of hashes

Build a hash table of small "secondary" hash tables.

By trying several hash functions from a universal class, we can easily achieve the goal of having no collisions in the secondary tables. A similar "tryout" principle on the primary hash function will allow us to choose a primary hash function that doesn't have lots of collisions, so that our tablesize for the primary hash table can be $m = O(n)$.

Recall that a member of a universal class of hash functions is fully specified by p, m, a, b where p is a prime greater than all possible keys, m is the size of the hash table, and a and b are chosen randomly from the range 1 up to $p - 1$.

Hash of hashes

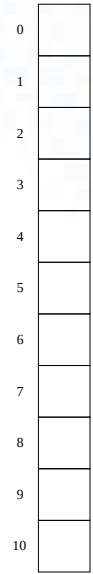
Main hash table: choose size $m = n$ where n is the number of data items, choose $p >$ the biggest key value. Choose a and b randomly to get primary hash function $h(k) = ((ak + b)\%p)\%m$.

Test h as follows:
1. For each key k, work out the home cell $h(k)$. Keep a count n_i for each cell i of how many keys hash to i.
2. Check whether space required is too big: is the sum of all the n_i^2 giving a total $> 2n$? Then h is not good enough so repeat the process with new a, b for a new primary hash function.

If the primary hash function h is good enough:
For each slot i, get the secondary hash function h_i by setting $p_i = p$ (i.e. p doesn't change), setting $m_i = n_i^2$, and choosing a_i and b_i randomly. Check to make sure that the resulting h_i doesn't cause any collisions within the secondary table.

Perfect hashing example

Data is: 8, 22, 36, 75, 61, 13, 84, 58



Cuckoo hashing

Use two hash functions h and g, so every key k gets two addresses, $h(k)$ and $g(k)$.

Insert k at $h(k)$ to start. If this address is occupied, make room for k by moving the current occupant to its second address.

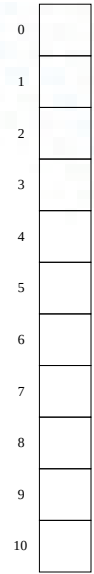
When you move a key to its other address, that cell may also be occupied. Repeat the behaviour, just as a cuckoo kicks the eggs of another bird out of the nest to make room for its own.

Either the process stops when an empty cell is found, or it loops, so we rehash with new hash functions. This is expensive but theoretically rare.

The plus is that search is $O(1)$. Why? And we can delete! What is the downside of cuckoo hashing?

Cuckoo hashing example

Use $h(k) = k\%m$ and $g(k) = 1 + k\%(m-1)$ to insert the keys 8, 22, 36, 75, 61, 13, 84, 58 by cuckoo hashing.



Relevant parts of textbook

Perfect hashing is discussed in Section 11.5. Theorem 11.9 uses a different analysis of the probability of a collision to show that you need $m = n^2$ for a probability of collision less than 0.5 (the analsyis in the textbook looks simpler, but there are some deeper concepts used).

Cuckoo hashing is not discussed in the textbook. But this pdf `https://web.stanford.edu/class/cs166/lectures/13/Small13.pdf` contains quite a good deal of information.