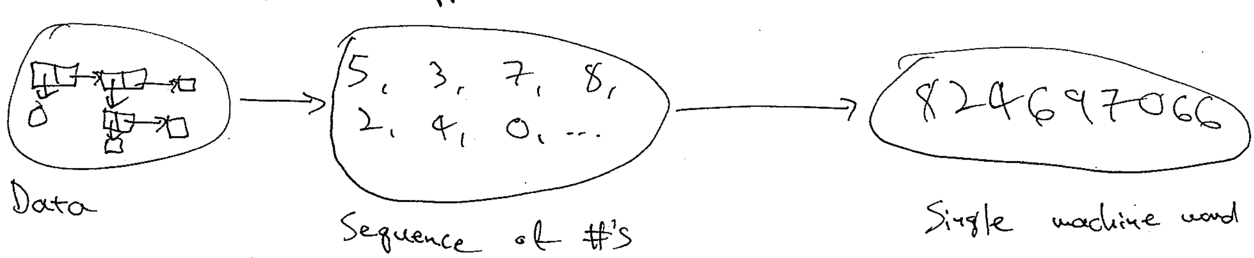
Hashing

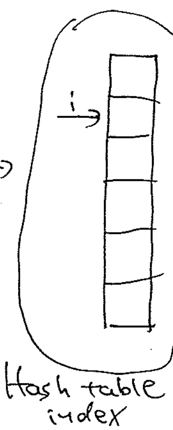
6.006 Rec. 9/25/09

* Hash functions
  - Mod hash
  - Multiplication hash
  - How to create a good hash function
* Hash tables
  - How big? Uniform hashing assumption.
  - Collision resolution
  - Table doubling
* Python dictionaries
* Document distance with hash functions.

Hash functions

We have: some large, complicated data structure
We want: a uniformly distributed table index
Here's the general approach:





Data → Sequence of #'s

Some data are just #'s. Others might be an array. If you have a more complicated data structure, like a tree or a graph, serialize it in some way.

~~For the other~~

Sequence of #'s → machine word

Let $\{a_n\}$ be the sequence of #'s. Let b be some base. Then, write the sequence as a polynomial in b:

$$a_0 + a_1 b + a_2 b^2 + a_3 b^3 + \cdots + a_n b^n$$

Evaluate this for some particular b.

Machine word to table index

Take $x \bmod m$, where x is the # above and m is the table size.

Good hash function examples

Two common ways:

Multiplication hash

Let the tablesize m be a power of 2 and let the base b be odd. Then just compute

$$a_0 + a_1 b + a_2 b^2 + \cdots + a_n b^n$$

allowing overflow of the machine word. At the end, take the last $\log_2 m$ bits to get the table index.

Mod hash

Let the tablesize be a prime m. Then compute

$$\beta (a_0 + a_1 b + a_2 b^2 + \cdots + a_n b^n) \bmod m$$

by taking the remainder mod m at every step. The base b can be anything not divisible by the prime m which is larger than all the a_i's. The number β should not be too close to a power of 2.

Remember: Primes are your friends: multiply by them, divide by them.

Good practice: Pick a prime which is not close to a power of 10 or a power of 2. (For the mod hash, $m \approx \phi b^k$, for $\phi = \frac{1+\sqrt{5}}{2}$ and some k, is good.)

Hash tables

Uniform hashing: with a good hash function, keys should be uniformly distributed in the table, with each key equally likely to go in each bucket.

Suppose you have n balls and you throw them at random into m bins. Then, on average, each bin has $\frac{n}{m}$ balls. This value, $\alpha = \frac{n}{m}$, is called the <u>load factor</u> of the hashtable. Then, the amount of time to search a random bin is $O(1+\alpha)$. (1 cost upfront plus $\approx \alpha$ balls found there)

Collision resolution

Two common methods of dealing with collisions:

Chaining — discussed last lecture
Open addressing — discussed next lecture

Table doubling

If we know the number of keys n in advance, then we can pick a table size m such that the load factor $\alpha = \frac{n}{m} \lesssim 1$. If we don't know the # of keys in advance, then α may increase above 1 as we add more keys. To fix this, when α gets too large, we can pick a new table size $m' \approx 2m$ and rebuild a bigger hashtable with half the load factor. This is similar to the array length doubling used in, e.g., Python lists.

Python dictionaries

Items are pairs of the form (key, value)

Create dictionary: ~~dict = {'algorithms': 5}~~
dict = {'algorithms': 5, 'cool': 42 }

d.items() → [('algorithms', 5), ('cool', 42)]
d['cool'] → 42
d[42] → KeyError
'cool' in d → True
42 in d → False

Python set is really dict where the items are just keys (not pairs).