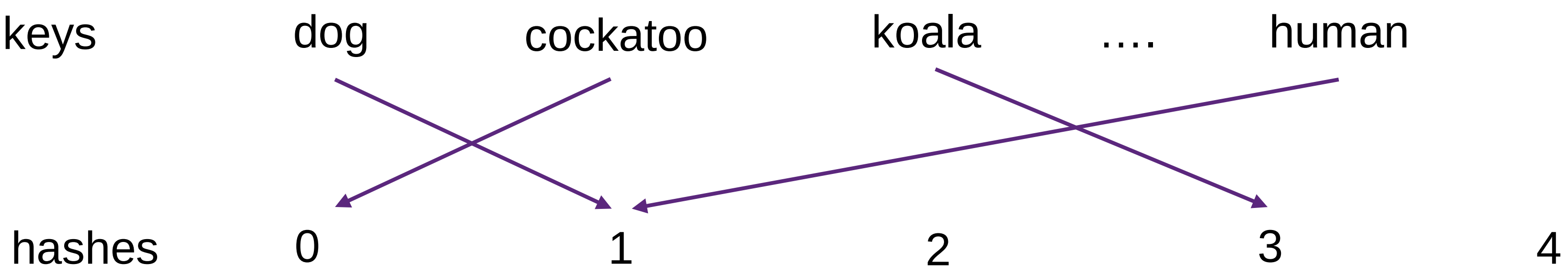
C04 Hash Functions

Hash functions
Choosing a good hash function

Hash functions

- A hash function is a function *f* that maps a key *k*, to a value *f(k)*, within a prescribed range. It maps arbitrary sized keys to fixed-sized *hashes*.
- A hash function is deterministic:
  - for a given key, *k*, *f(k)* will always be the same;
  - if *k* == *l*, then *f(k)* == *f(l).*



Uses of hashing

- Hash table (implement a set or map)
- Checksums
  - Error detection and/or correction
- Compression
  - A hash is typically much more compact than the key
- As a fast inequality test
- Cryptographic

Choosing a good hash function

- A **good hash** for a given population, P, of keys, $k \in P$, will distribute $f(k)$ evenly within the prescribed range for the hash.
- A ***perfect* hash** will give a unique $f(k)$ for each $k \in P$.

(Perfect hash is rarely possible –

Pigeon hole principle.)

https://upload.wikimedia.org/wikipedia/commons/thumb/5/5c/TooManyPigeons.jpg/220px-TooManyPigeons.jpg

Why value determinism and even distribution?

- Lets reword how we stated determinism a bit:
  - Given x, y, if x == y, then h(x) == h(y).
  - It follows that (by contraposition):
    - If h(x) != h(y), then x != y
- Even though we cannot give positive result (x is y) confidently,
  - We can for the negative result (x is **not** y)

Why value determinism and even distribution?

- Now lets suppose h(x) gives an integer in range [0, 9]
- And suppose input is uniformly random
- With 10 values (or buckets), given inputs x and y, we have 90% chance of deciding x != y in O(1)
- There is still a 10% chance of collision, but we have cut down our average workload of later stage by 90%
  - HashSet vs ArrayList

Why so many different hashes?

- We outlined the basic properties we look for in a hash
  - Deterministic
    - This is fundamental, and by definition of a mathematical function
    - No exception to this requirement
  - Even/uniform distribution of output
    - This is not as indisputable – we don’t know what the distribution of input is like
    - But we try to obtain this by guessing what the “usual” input looks like, e.g. statistical analysis of past usage
- The second point is roughly where the divergence begins

Why so many different hashes?

- For each input distribution, we would need a different hash function to get an even distribution.

Evenly distributed output if input is normally distributed	Evenly distributed output if input is evenly distributed	Evenly distributed output if input is bimodal
Deterministic	Deterministic	Deterministic

Why so many different hashes?

- Even more variations if we want additional properties

Fast on CPU
Evenly distributed output if input is normally distributed
Deterministic

Low memory usage
Evenly distributed output if input is evenly distributed
Deterministic

Secure
Low memory usage
Evenly distributed output
Deterministic

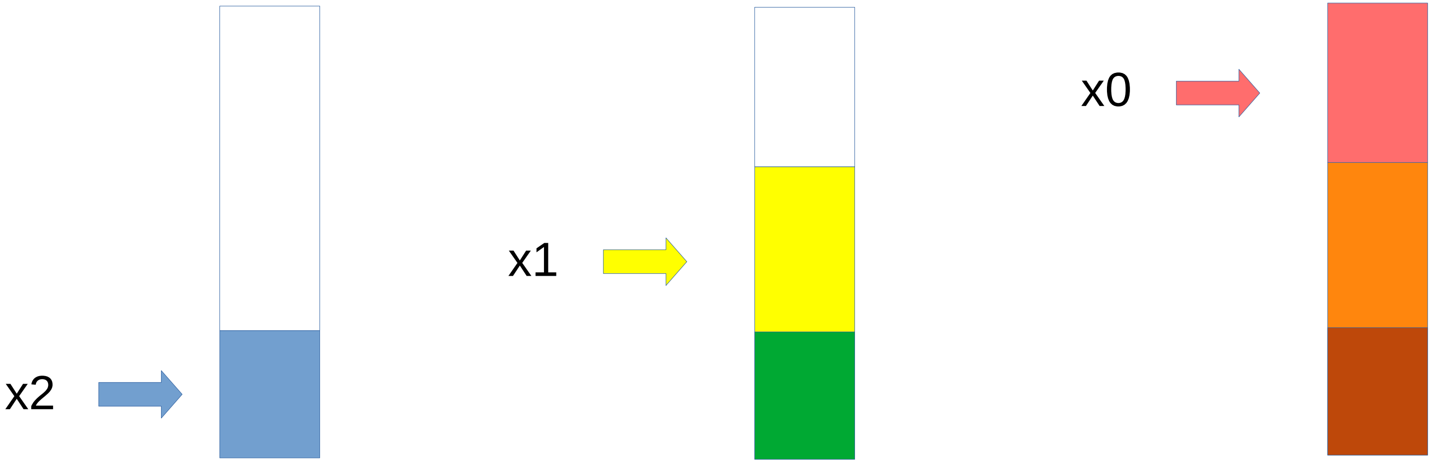
Very high memory usage
Very slow
Evenly distributed output
Deterministic

Assume whatever distribution, pick a recipe

- From “Effective Java”, Josh Bloch
- (An approximate translation below in pseudo code)
- Assume you have fields (or more generally values) field0, field1, field2, …
- ```
  int result = 0; // accumulator
  for (var field : fields) {
      var x = convertToInt(field); // recursively call this hash if needed
      result = 31 * result + x;
  }
  ```
- How does this work? Suppose we have fields: x0, x1, x2
- After loop 0, result = x0
- After loop 1, result = 31 * x0 + x1
- After loop 2, result = 31 * (31 * x0 + x1) + x2 = 961 * x0 + 31 * x1 + x2
  ```

Intuition behind this pattern

- Why 961 * x0 + 31 * x1 + x2? (or similar)
- Each factor is used to disperse the field to a different band/partition of the output range, so it is sensitive to change in any field.



- Note: For hashing, overflow (wrap-around) is not a bad thing.

Why 31?

- From the book, multiplication with 31 is very efficient:
  - 31 * x = (x << 5) - x
- We don’t use odd primes very often. Suppose we use 100 instead of 31:
  - 10000 * x0 + 100 * x1 + x2
- If we then reduce the range of hash by taking it % 10, the above becomes
  - x2
- (Of course if we modulo 31, we run into the same problem.)

Converting things into int

- Again mostly based on the recipe from Effective Java book
- Any numeric primitive type: multiply by prime, hashCode(), Float.floatToIntBits(x)
- Recursive: 31 * node.left.hashCode() + node.right.hashCode()
- Linear/array: treat each element as a field in previous recipe

More complex hash

- We can always mix and match, and use the recipe as the base skeleton
- Suppose we parameterise the recipe as
  - hash(int prime, List<int> fieldHashes)
- Examples:
  - hash(31, fields in some order) // original reciple
  - hash(31, fields in some order) + hash(7, fields in reverse order)
  - Use a mix of primes: 67 * 31 * x0 + 31 * x1 + x2