Hashing

DAT470/DIT065 Computational techniques for large-scale data

2024-04-24

Matti Karppa

Bitwise operations

- There are 6 basic bitwise operations (in Python)
  - Bitwise AND or &
  - Bitwise OR or |
  - Bitwise XOR or ^
  - Bitwise NOT or ~
  - Left shift <<
  - Right shift >>
- The hardware supports more operations that don’t have a standard operator in Python, including left and right rotations
- We will go through each of these operations and see how they operate

Bitwise AND &

- Bitwise AND is a **binary** operator: it takes two inputs and yields one output
- Bitwise AND compares the corresponding bit on both input operands, and sets the corresponding bit to 1 in the output if and only if it is 1 in **both** inputs
- This is useful for **bitmasking**: if the other operand is a literal whose certain bits are set, we can **select** only those bits from the other operand (like when we enforced the length of integers)
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, then `(x & y) = 81 = 01010001`
  because
  `253 = 11111101`
  `083 = 01010011`
  `081 = 01010001`

Bitwise OR |

- Bitwise OR is a **binary** operator: it takes two inputs and yields one output
- Bitwise OR compares the corresponding bit on both input operands, and sets the corresponding bit to 1 in the output if and only if it is 1 in **either or both** inputs
- This is useful for setting bits: if the other operand is a literal whose certain bits are set, we can **set** those bits to be true
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, then `(x | y) = 255 = 11111111`
  because
  `253 = 11111101`
  `083 = 01010011`
  `255 = 11111111`

Bitwise XOR ^

- Bitwise XOR is a **binary** operator: it takes two inputs and yields one output
- Bitwise XOR compares the corresponding bit on both input operands, and sets the corresponding bit to 1 in the output if and only if it is 1 in **either but not both** inputs
- This corresponds to flipping bits, and also addition over GF(2)
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, then `(x ^ y) = 174 = 10101110` because

```
253 = 11111101
083 = 01010011
174 = 10101110
```

Bitwise NOT ~

- Bitwise NOT is a **unary** operator: it takes one input and yields one output
- Bitwise NOT simply flips all bits of its operand
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, then `~x = 2` and `~y = 172` because

```
253 = 11111101
002 = 00000010
083 = 01010011
172 = 10101100
```

- Note that if you operate on Python `ints`, you will need to enforce the bit length with bitmasking because otherwise the negation of a positive number will be a negative integer, and the implicit leading ones are negated as well

Left shift <<

- Left shift is a **binary** operator: it takes two inputs and yields one output
- The left-hand input is shifted to the left by padding it to the right with a number of zeros as specified by the right-hand operand
- For fixed-precision numbers, the bits that overflow to the left are discarded
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, and assuming unsigned 8-bit arithmetic, then
  `(x << 3) = 232` and `(y << 5) = 96` because
  `253 = 11111101`
  `232 = 11101000`
  `083 = 01010011`
  `096 = 01100000`
- Note that if you operate on Python `ints`, you will need to enforce the bit length with bitmasking because otherwise shifting to the left will just grow the integer; no leftmost bits are discarded at any point
- Thus, to get the examples above in Python, you'd have to use `(x << 3) & 0xff`, and `(y << 5) & 0xff`, respectively
- Note that left-shifting corresponds to multiplying by a power of two (and is faster than multiplication so compilers usually optimize such multiplications as a shift)

Right shift >>

- Left shift is a **binary** operator: it takes two inputs and yields one output
- The left-hand input is shifted to the right by discarding the rightmost bits as specified by the right-hand operand
- How the left end of the number is padded depends on programming language; however, for unsigned numbers, zeros are usually inserted to the left
- Python `ints` are **sign-extended**: if the number is negative, then ones are (implicitly) inserted to the left-end; if this is not desired, then bitmasking must be used
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, and assuming unsigned 8-bit arithmetic, then
  `(x >> 3) = 31` and `(y >> 5) = 2` because
  `253 = 11111101`
  `031 = 00011111`
  `083 = 01010011`
  `002 = 00000010`
- Note that sign-extending right-shift corresponds to division by a power of two, and in fact, is more efficient than a division instruction, so compilers usually emit a right-shift instead of a division if they can determine that the division is by a power of two

Rotation

- Left and right rotations are variations of the left and right shift
- Instead of discarding the bits, the bits are put to the other side of the word (they “flow over”)
- Left rotation by i bits can be implemented as `rol(x,i) = (x << i) | (x >> (b-i))` assuming unsigned b-bit arithmetic
- Similarly, right rotation would be `rol(x,i) = (x >> i) | (x << (b-i))`
- Example: suppose `x = 253 = 11111101` and `y = 83 = 01010011`, and assuming unsigned 8-bit arithmetic, then `rol(x,3) = 239` and `ror(y,5) = 154` because

```
253 = 11111101
191 = 11101111
083 = 01010011
154 = 10011010
```

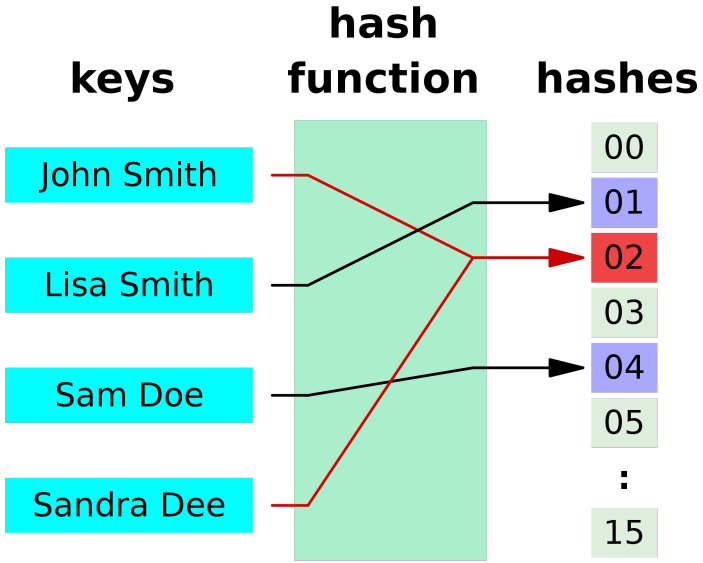
- Rotations are a special instruction on many CPUs, so using the standard way to implement the rotation is advised, as compilers can often recognize it and emit the correct instruction
- Note that on Python you may need to use bitmasks at the appropriate places to prevent overflows and sign extensions

Idioms and caveats

- **Idiom:** select the i^{th} bit: `(x >> i) & 1`
- **Idiom:** set the i^{th} bit: `x | (1 << i)`
- **Idiom:** clear the i^{th} bit: `x & (~(1 << i))`
- Beware: bitwise operations have **surprising** precedence, so you should **always** use parentheses around them
- For example, `1 << 2 * 3` evaluates to 64 because * has higher precedence than <<
- There are differences in the precedence depending on language, and in languages like C, bitwise operators have very low precedence (lower than ==) which easily leads to weird bugs if parentheses are not used appropriately

Hash function

- A **hash function** $h : U \to R$ maps the elements of a universe U to some fixed number of values R
- In most interesting cases, $|R| \ll |U|$
- The universe can be an arbitrary set of strings, or numbers, or any kind of objects
- The codomain of the hash function is usually a set of integers, such as the set of 32-bit or 64-bit unsigned integers, or sometimes (in theoretical analysis) segments of the real line
- Hash functions often come as **families** of hash functions $\mathcal{H}$, from which one particular function can be drawn according to some distribution
- Whenever two **distinct keys** $x, y \in U$, $x \neq y$, map to the same **hash value**, that is $h(x) = h(y)$, we say that there is a **hash collision**



https://commons.wikimedia.org/wiki/File:Hash_table_4_1_1_0_0_1_0_LL.svg

Randomization vs. hashing

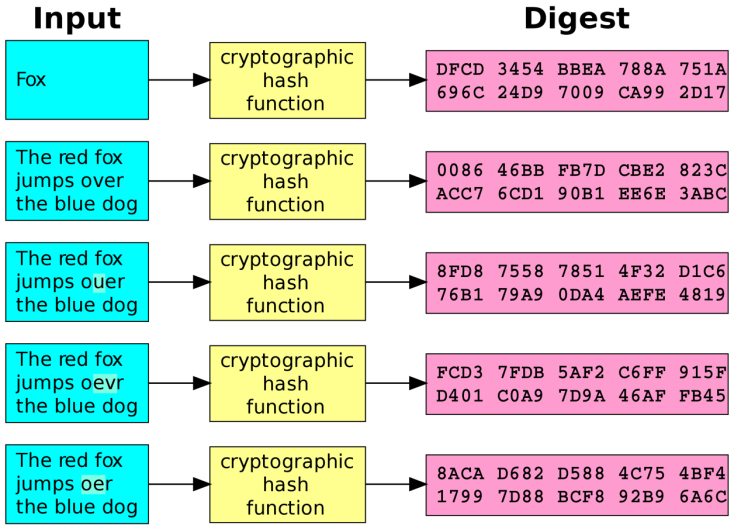
- Hash functions are often used to **implement** randomized algorithms
- Randomness is then **in the choice of the hash function**
  - Randomness is over the distribution of the hash function family
- Once the hash function has been drawn from the distribution **it is deterministic**
  - No more randomness at this step
- In a lot of practical applications, we fix the hash function beforehand
  - This may be required to ensure that all others also see the same hash values

Properties of good hash functions

- A good hash function should be **fast to compute** (we often need to compute the hash values for all elements in a dataset)
- We often want the hash function to **minimize** collisions
- Ideally, we would often like the hash function to behave as if it was uniformly random
  - Denote $[m] = \{1,2,\dots,m\}$
  - Suppose h is drawn from a family $\mathcal{H} = \{h : U \to [m]\}$ of random hash functions mapping elements to m **bins**
  - The probability that any element $x \in U$ falls to a certain bin $j \in [m]$ is $\Pr_{h\sim\mathcal{H}}[h(x) = j] = \frac{1}{m}$
  - The probability of any event occurring, that is, for all $j_1, j_2 \in [m]$, we have
$$\Pr_{h\sim\mathcal{H}}[h(x) = j_1 \wedge h(y) = j_2] = \Pr_{h\sim\mathcal{H}}[h(x) = j_1]\Pr_{h\sim\mathcal{H}}[h(y) = j_2] = \frac{1}{m^2}$$
  - The probability that any two distinct elements collide is thus
$$\Pr_{h\sim\mathcal{H}}[h(x) = h(y)] = \sum_{j=1}^{m} \Pr_{h\sim\mathcal{H}}[h(x) = j \wedge h(y) = j] = \sum_{j=1}^{m} \Pr_{h\sim\mathcal{H}}[h(x) = j]\Pr_{h\sim\mathcal{H}}[h(y) = j] = \frac{1}{m}$$
- Unfortunately, truly random hash functions require exponential number of bits in the length of the element to **describe** (and truly random bits are difficult to obtain)
- Weaker hash functions often perform very well in practice anyway

Cryptographic hash functions

- **Cryptographic hash functions** satisfy certain properties
  - Hash values are (approximately) **uniformly distributed**, that is, any n-bit hash value occurs with probability 2^{-n}
  - Given y, finding x such that $y = h(x)$ is **unfeasible**; that is, it is very difficult to find a **pre-image** that hashes to a given value
  - Given x and y such that $y = h(x)$, it is unfeasible to find $x' \neq x$ such that $y = h(x')$; that is, it is very difficult to find a **second pre-image** that hashes to a given value as another key
  - Finding a pair of keys x, x' such that $h(x) = h(x')$ is unfeasible; that is, it is very difficult to find a **hash collision**
- Examples of cryptographic hash functions include SHA-1, SHA-2, SHA-3, BLAKE
- These often come with significant computation cost, so unless security requirements are involved, they are not necessarily a good choice
- We're happy with simpler but faster functions



https://commons.wikimedia.org/wiki/File:Cryptographic_Hash_Function.svg

Trivial hashing of integers

- Let us assume we want to hash b-bit integers into a domain of size m
- What is the simplest possible hash function $h : [2^b] \to [m]$?
- Perhaps $h(x) \equiv x \bmod m$
- While adequate for a lot of purposes, this is generally a very poor hash function if $m < 2^b$
- If m is a power of 2, then $h(x)$ simply selects the low bits of x
- Even if m were a prime, then for all a, $h(x + am) = h(x)$, so collisions can be guaranteed
- This is very bad especially in an adversarial setting where an adversary can introduce an arbitrary number of collisions, guaranteeing worst-case behavior

Universal hashing

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- We say that a hash function (family) $h : U \to [m]$ is **universal** if it satisfies for all distinct keys $x, y \in U$, $x \neq y$

$$\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

- We say the family is c**-approximately universal** if the family satisfies for some $c > 0$

$$\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq \frac{c}{m}$$

- Trivial hashing is universal if and only if $m \geq |U|$ (not a very interesting case)
- **Multiply-mod-prime**
  - Suppose $U = [u]$ and pick **prime** $p \geq u$
  - Choose uniformly random $a \in \{1,2, \dots, p-1\}$ and $b \in \{0,1, \dots, p-1\}$, and define

$$h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod m$$

  - Then, $\Pr_{a,b}\big[h_{a,b}(x) = h_{a,b}(y)\big] < \frac{1}{m}$, so it is universal (see Thorup's paper for proof)
  - While theoretically interesting, this is not very practical because it involves computing the modulo operation (which is slow); also the product of two 64-bit integers requires 128 bits, as $u \geq 2^{64}$ when the keys are 64-bit
  - Using Mersenne primes helps a bit because then we get away with bitwise and, right shift, and subtraction

Multiply-shift

- Suppose we want to hash w-bit integers into ℓ-bit integers
- Pick uniformly random **odd** w-bit integer a and define $h_a : [2^w] \to [2^\ell]$ as

$$h_a(x) = \left\lfloor \frac{ax \bmod 2^w}{2^{w-\ell}} \right\rfloor$$

- **Fact:** w-bit multiplication discards overflow, so $ax \bmod 2^w$ is simply ax in unsigned w-bit arithmetic
- **Fact:** Truncated division by a power of two is equivalent to right shift, so floored division by $2^{w-\ell}$ is simply right shift by $w - \ell$ bits
- Multiply-shift can thus be implemented as follows in C for 64-bit keys:

```
uint64_t hash(uint64_t x, uint64_t l, uint64_t a) {
  return (a*x) >> (64-l);
}
```

- Multiply-shift satisfies $\Pr_{a \in \{x \in [2^w] \mid x \text{ is odd}\}}[h_a(x) = h_a(y)] \leq \frac{2}{m}$, so it is 2-approximately universal

Strong universality

- Let $h : [u] \to [m]$
- Consider all pairwise events
  - Let the keys $x, y \in [u]$ be distinct, $x \neq y$
  - Let $q, r \in [m]$ be their (not necessarily distinct) hash values, that is, $h(x) = q$ and $h(y) = r$
- We say h is **strongly universal** if the probability of every pairwise event is $\frac{1}{m^2}$
- That is,

$$\Pr[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$$

- Since $\Pr[h(x) = h(y)] = \sum_{q \in [m]} \Pr[h(x) = q \wedge h(y)] = \frac{m}{m^2} = \frac{1}{m}$, strong universality implies universality
- We say $h : [u] \to [m]$ is ***c*-approximately strongly universal** if
  - h is c-approximately uniform, that is, for every key $x \in [u]$ and every hash value $q \in [m]$

$$\Pr[h(x) = q] \leq \frac{c}{m}$$

  - Every pair of disticnt keys hash independently

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

k-wise independence

- Strong universality can also be characterized as a hash function is strongly universal if each key is hashed uniformly into $[m]$ and every two distinct keys are hashed independently
- Strong universality is also called **pairwise independence**
- The concept can be generalized to k-wise independence: a hash function $m : [u] \to [m]$ is ***k*-wise independent** if, for any k **distinct** keys $x_1, x_2, \ldots, x_k$ and any k (not necessarily distinct) hash values $q_1, q_2, \ldots, q_k$,

$$\Pr[h(x_1) = q_1 \wedge h(x_2) = q_2 \wedge \cdots \wedge h(x_k) = q_k] = m^{-k}$$

- Equivalently, h is k-wise independent if
  - For any fixed $x \in [u]$, $h(x)$ is uniformly distributed in $[m]$ (over the distribution of h)
  - For any fixed distinct k keys $x_1, x_2, \ldots, x_k$, the values $h(x_1)$, $h(x_2)$, …, $h(x_k)$ are independent
- k-wise independence can be used to establish strong theoretical guarantees, even in an adversarial setting

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

Strongly universal multiply-shift

- Suppose we want to hash w bit keys into ℓ bit values, that is, we want a $h : [2^w] \to [2^\ell]$
- Choose $\overline{w} \geq w + \ell - 1$, draw $a, b \in 2^{\overline{w}}$ independently and uniformly at random
- Define $h_{a,b}(x) = (ax + b)|\overline{w}, \overline{w} - \ell)$ where "$|\overline{w}, \overline{w} - \ell)$" means "take the ℓ highest bits" (of the $\overline{w}$-bit word)
- This family of hash functions is strongly universal!
- Furthermore, it is very efficient to implement:
  - We need one multiplication (although of larger size)
  - Choosing the high bits can be done with right shift

Example

- Suppose we want to hash $w = 32$-bit keys into ℓ-bit hash values with $1 \leq \ell \leq 32$
- We can then choose $\overline{w} = 64$
- We then choose 64-bit a, b uniformly and independently at random
- We compute, using 64-bit arithmetic, $ax + b$, and right-shift the result by $64 - \ell$ bits
- In C,

```
uint32_t hash(uint32_t x, uint32_t l, uint64_t a, uint64_t b) {
  return (a*x+b) >> (64-l);
}
```

Generalizations

- Hashing d-dimensional vector keys with w-bit coefficients into ℓ-bit hash values:
  - Choose $\overline{w} \geq w + \ell - 1$
  - Pick integers $a_1, a_2, \ldots, a_d, b \in [2^w]$ independently and uniformly at random
  - Define

$$h_{a_1,a_2,\ldots,a_d,b}(x_1, x_2, \ldots, x_d) = \left(\left(\sum_{i=1}^{d} a_i x_i\right) + b\right) | \overline{w} - \ell, \overline{w})$$

  - This is strongly universal
- For even d, we can reduce the number of multiplications by defining

$$h_{a_1,a_2,\ldots,a_d,b}(x_1, x_2, \ldots, x_d) = \left(\left(\sum_{i=1}^{d/2} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})\right) + b\right) | \overline{w} - \ell, \overline{w})$$

  - An array of 64-bit numbers can be efficiently hashed by treating it as an array of 32-bit numbers ($\overline{w} = 64$, $w = 32$)
- Practical case with $d = 2$: hashing 64-bit keys into $1 \leq \ell \leq 32$-bit values

```
uint32_t hash(uint64_t x, uint32_t l,
              uint64_t a1, uint64_t a2, uint64_t b) {
  return ((a1+x)*(a2+(x>>32))+b) >> (64-l);
}
```

Generalizations

- Suppose we have a c-approximately strongly uniform hash function $h : [u] \to [M]$ (think: typically M is a power of two)
- We want to hash to $[m]$ with $m < M$ (think: m is not power of two but a general range, e.g., a number of hash table bins)
- If $r : [M] \to [m]$ is **most uniform**, that is, for any $z \in [m]$, the number of $y \in [M]$ such that $r(y) = z$ is either $\left\lfloor \frac{M}{m} \right\rfloor$ or $\left\lceil \frac{M}{m} \right\rceil$, then $r \circ h$ is $\left(1 + \frac{M}{m}\right) c$-approximately strongly universal
- One such r is $y \mapsto \left\lfloor \frac{ym}{M} \right\rfloor$
- In C, with $M = 2^{32}$,

```
uint32_t hash(uint32_t x, uint32_t m, uint64_t a, uint64_t b) {
  return (((a*x+b)>>32)*m)>>32;
}
```

Generalizations

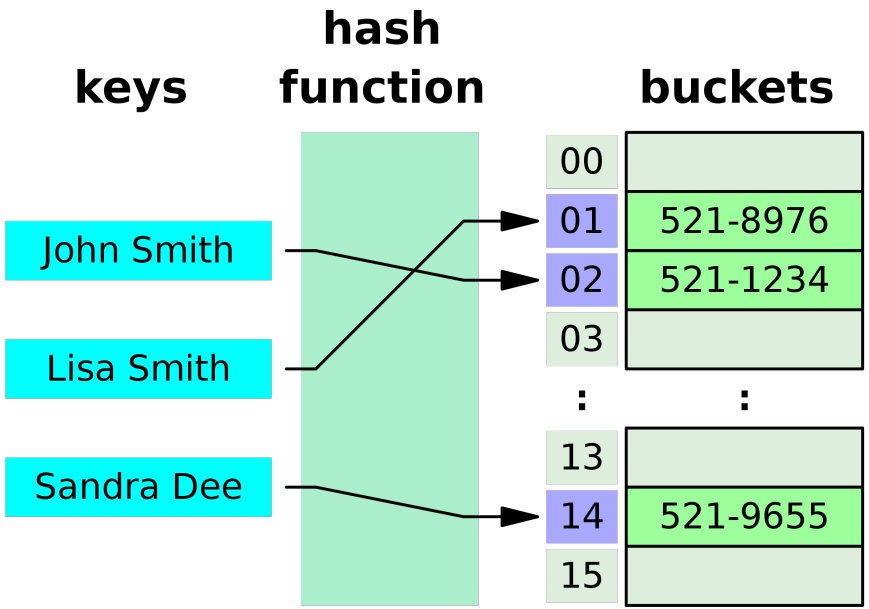
- What if we want to hash to, e.g., 64-bit values?
- Suppose $h_1 : U \to R_1$ is c_1-approximately strongly universal and $h_2 : U \to R_2$ is c_1-approximately strongly universal
- Define $h : U \to R_1 \times R_2$ by $h(x) = (h_1(x), h_2(x))$
- h is $(c_1 c_2)$-approximately strongly universal
- For example, to hash 64-bit keys into 64-bit hash values:
  - Draw six 64-bit integers uniformly at random $a_1, a_2, a_3, a_4, b_1, b_2$
  - Define $h_1 : [2^{64}] \to [2^{32}]$ using a_1, a_2, b_1 as before
  - Define $h_2 : [2^{64}] \to [2^{32}]$ using a_3, a_4, b_2 as before
  - Define $h(x)$ as the concatenation of the two 32-bit words $h_1(x), h_2(x)$

```
uint32_t hash64to32(uint64_t x, uint32_t l,
                    uint64_t a1, uint64_t a2, uint64_t b) {
  return ((a1+x)*(a2+(x>>32))+b) >> (64-l);
}

uint64_t hash64to64(uint64_t x, uint64_t l, uint64_t a1, uint64_t a2,
                    uint64_t a3, uint64_t a4, uint64_t b1, uint64_t b2) {
  uint64_t y = hash64to32(x, 32, a1, a2, b1);
  y <<= 32;
  y |= hash64to32(x, 32, a3, a4, b2);
  return y >> (64-l);
}
```

Hash tables

- Hash functions immediately give rise to **hash tables**: associative arrays where key-value pairs are stored in an array, as indicated by a hash function
- The Python dictionary is an example of a hash table
- Suppose the set of keys is K and of values is V
- The underlying data structure is an array X of m bins, together with a hash function $h : K \to [m]$
- **Insertion:** compute the index of the bin matching the key $j \leftarrow h(k)$, then store (k, v) in $X[j]$
- **Retrieval:** compute the index of the bin matching the key $j \leftarrow h(k)$, then return v from $X[j]$ if the bin is occupied (or otherwise return none)
- **Removal:** compute the index of the bin matching the key $j \leftarrow h(k)$, then mark $X[j]$ empty if it contained the key



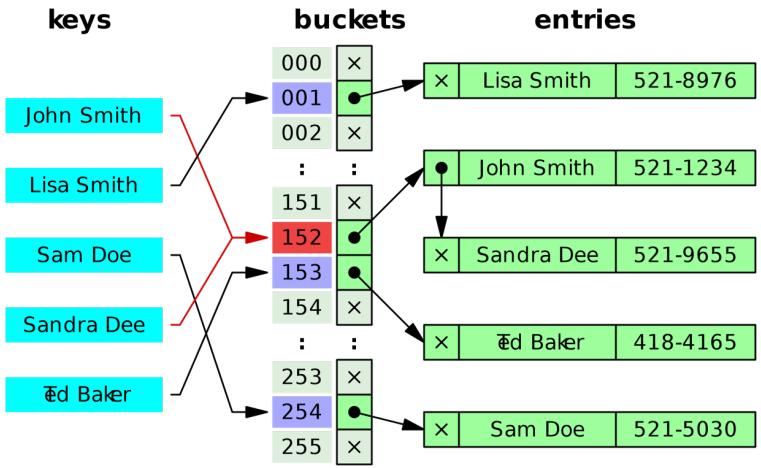
https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg

Hash tables

- Hash tables are **flexible**: they usually are implemented in such a way that they can grow if they become too full and shrink if elements are removed
- Growing the hash table is expensive: it requires one to allocate a new, larger array, **rehashing** of all elements (with a new hash function matching the new size), and copying the elements to their new locations
- If there is already an element in the bin (a hash collision occurs), **collision resolution** is needed
- Load factor $\alpha = \frac{n}{m}$ describes how full the hash table is for n elements and m bins

Separate chaining

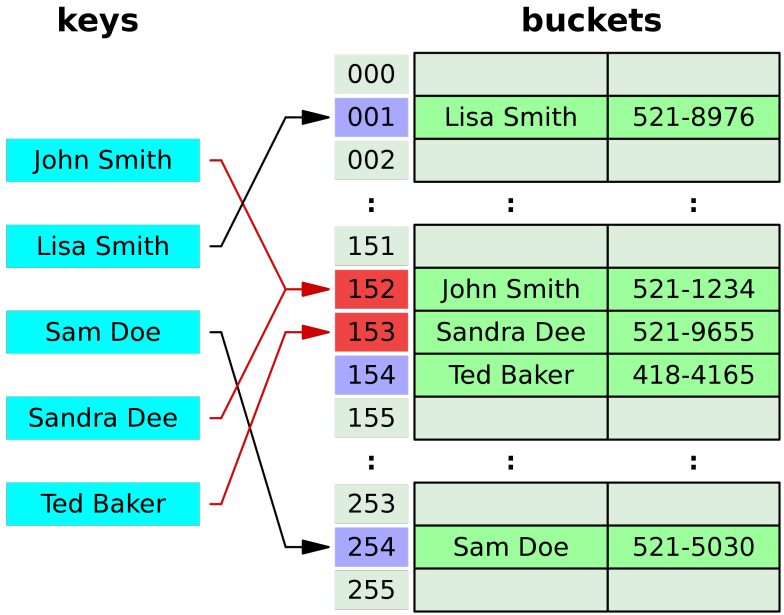
- In separate chaining, all elements of the array are linked lists of elements mapped to that bin
- Advantages:
  - Flexible
  - Easy to implement for arbitrary data
  - Can make use of ordering of data
  - Efficient when α is close to 1
- Disadvantages
  - Linked lists make poor use of cache locality



https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg

Open addressing

- In open addressing, if the initial bin is occupied, the array is **probed** until an empty bin is found and the key is stored there
- Various probing strategies:
  - **Linear probing:** Try elements one at a time
  - **Quadratic probing:** Try index $j + i^2 \bmod m$ for the ith attempt
  - **Double hashing:** Use another hash function for the probing sequence
- Advantages:
  - Cache-friendly
  - Very efficient when α is close to 0
- Disadvantages
  - Inefficient when α is close to 1



https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_SP.svg

Asymptotics of hash tables

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Hash tables provide a tradeoff between running time and space usage, saving potentially a lot of space in comparison to a full array that is mostly empty (as only occupied keys need to be stored), so they need $\Theta(n)$ space
- If the hash table does not need resizing, then insertion/deletion is an $O(1)$ operation (assuming the hash function can be evaluated in $O(1)$ time)
- However, when resizing is needed, the operation becomes $O(n)$
- Hash tables provide **amortized** constant operations: the operations are constant-time **on average** over a large number of accesses
- However, worst-case behavior is $\Omega(n)$ (think about pathological sequences that induce a very large number of collisions)

Balls and bins

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Suppose we throw n balls into m bins uniformly at random
- This is a useful model that can be used to answer questions like
  - How many balls are going to end up in any one bin in expectation?
  - How many empty bins do we expect to see?
  - What is the probability that a particular bin is empty?
  - And so on
- Expected number of balls in a bin:
  - For each $i \in [n]$ and each $j \in [m]$, define the indicator variable $X_{i,j} = \begin{cases} 1 \text{ if the ball } i \text{ ends up in bin } j \\ 0 \text{ otherwise} \end{cases}$
  - Then, the random variable $X_j = \sum_{i=1}^{n} X_{i,j}$ is the number of balls that end up in bin j
  - By assuming uniformity and independence, we have that $X_{i,j} \sim \text{Bernoulli}\left(\frac{1}{m}\right)$, so $\Pr[X_{i,j} = 1] = \frac{1}{m}$ for all i, j
  - Therefore, $\mathrm{E}[X_{i,j}] = \frac{1}{m}$
  - By linearity of expectation $\mathrm{E}[X_j] = \sum_{i=1}^{n} \mathrm{E}[X_{i,j}] = \frac{n}{m}$

Distribution of balls in the bin

- Since the number of balls in a bin $X_j = \sum_{i=1}^{n} X_{i,j}$ is a sum of independent Bernoulli variables, we have that $X_j \sim \text{Binomial}\left(n, \frac{1}{m}\right)$
- So, the probability that there are k balls in the bin j is $\Pr[X_j = k] = \binom{n}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$
- Note that this only holds for **one particular bin**, not for joint bins because the events are dependent, negatively so: if the ball does **not** land in the bin j, we know that it has to land in some other bin, so the probability for the ball landing in some other bin (on the condition that it did not land in j) is greater than not knowing where it landed!
- We can, however, use the probability to determine the probability that bin j is empty:

$$\Pr[X_j = 0] = \left(1-\frac{1}{m}\right)^n \approx e^{-\frac{n}{m}}$$

- If we let Y_j be indicator variable for the bin j being empty, let $Y = \sum_{j=1}^{m} Y_j$ is the random variable for the number of empty bins
- By linearity of expectation, we then get that $\mathrm{E}[Y] = \sum_{j=1}^{m} \mathrm{E}[Y_j] = m\left(1-\frac{1}{m}\right)^n \approx me^{-\frac{n}{m}}$

Balls and bins and chain hashing

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Suppose we want to store objects (strings, integers) into a hash table $\mathcal{H}$ with separate chaining
- If we query whether an object x is in $\mathcal{H}$, we compute $h(x)$ and iterate over the associated linked list to find a match
- Question: How many objects do we need to compare to **on average**?
- We can model this as a balls and bins problem with n balls (objects) thrown into m bins (hash table size)
- If the object x **is not in** $\mathcal{H}$, we need to inspect $\frac{n}{m}$ objects on average
  - Since the number of balls in bin j is $X_j \sim \text{Binomial}\left(n, \frac{1}{m}\right)$, we get that $\text{E}[X_j] = \frac{n}{m}$
- If the object x **is in** $\mathcal{H}$, we know that the object is in the chain and we need to inspect $1 + \frac{n-1}{m}$ objects on average
  - Since the number of **other** balls in bin j is $X_j \sim \text{Binomial}\left(n-1, \frac{1}{m}\right)$, we get that $\text{E}[X_j] = \frac{n-1}{m}$
- If hashing takes a constant time and $m = n$, then lookup takes constant time **on average**

Simple tabulation hashing

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Let $U = [u]$ and $R = [2^w]$
- Treat each $x \in U$ as a string of c characters over an alphabet $\Sigma = [u^{\frac{1}{c}}]$
  - That is, $x = (x_1, x_2, \ldots, x_c)$ and each $x_i \in \Sigma$
- Define $h : U \to R$ by $h(x) = \bigoplus_{i \in [c]} h_i(x_i)$
  - Each $h_i : \Sigma \to R$ has been chosen uniformly at random
  - $\oplus$ stands for bitwise XOR
- To construct the h_is, for each $i \in [c]$, draw a uniformly at random an array A_i of $|\Sigma|$ w-bit words
- That is, for each possible character, there exists a random word, so $h_i(x_i) = A_i[x_i]$ is simply a lookup from a table

Simple tabulation hashing: Example

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Suppose the universe of keys U is the set of 32-bit integers, then $u = 2^{32}$
- Suppose we want to split the keys into 8-bit characters
  - We have $c = 4$ characters
  - Each key is split into $x = (x_1, x_2, x_3, x_4)$ and each $x_i \in \Sigma$
  - $\Sigma = \left[u^{\frac{1}{c}}\right] = \left[2^{\frac{32}{4}}\right] = [2^8]$
  - Basically, we just split the integers into bytes
- Suppose we want to hash the values into 16-bit integers
- We need to draw 4 arrays of 256 16-bit integers uniformly at random (or alternatively one array of 16-bit integers of shape 4×256)
- We could implement the actual hashing as follows:

$y \leftarrow 0$
for $i \leftarrow 0,1, \dots, 3$ **do**
  $y \leftarrow y \oplus A\left[i, \left(x \gg (i * 8)\right) \,\&\, 0\text{x}ff\right]$
done

Simple tabulation hashing: Example

- Suppose we want to hash the value $x = a5e2457f_{16}$
- Suppose this is the array we've drawn

$$A = \overbrace{\begin{bmatrix} d9cf & 1659 & \cdots & 435d & 1414 \\ b222 & 61dd & \cdots & 67e8 & e3cb \\ 306e & 5fbc & \cdots & 9e5c & fe79 \\ 17ab & 5063 & \cdots & bab2 & 5c69 \end{bmatrix}}^{256} \Big\} 4$$

- We have $x = (7f, 45, e2, a5)$ in little-endia order
  - These correspond to $x_1 = x \;\&\; 0xff$, $x_2 = (x \gg 8) \;\&\; 0xff$, $x_3 = (x \gg 16) \;\&\; 0xff$, $x_4 = (x \gg 24) \;\&\; 0xff$
- We thus compute
  - $h = A[1, x_1] \oplus A[2, x_2] = \oplus A[3, x_3] \oplus A[4, x_4] = A[1,7f] \oplus A[2,45] \oplus A[3, e2] \oplus A[4, a5]$
    $= 570b \oplus 2049 \oplus c129 \oplus 5a6d = ec06$

Simple tabulation hashing: Properties

- Tabulation hashing is a very attractive choice because it provides strong theoretical guarantees
- In particular, it provides stronger theoretical guarantees than 3-wise independent hashing
- However, the hash functions take some space to describe (using an alphabet larger than 16 bits is probably not doable or cache efficient)
- The ideas can be extended to hashing variable-length strings

Python __hash__

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Objects that are **hashable**, that is, that can be used as keys for dictionaries, or as elements of sets etc. must implement the method `__hash__` in Python
- `__hash__` is called when `hash()` is called on the object
- `__hash__` must return an integer
- The function `__hash__` must satisfy the requirement that if x and y are equal, then their hashes **must** be equal, that is, `x == y` implies `hash(x) == hash(y)`
- The standard way to implement `__hash__` for user-defined types is to pack the fields that play a role in equality comparisons into a tuple and hash that (`hash` of a tuple is well-defined if all elements of the tuple are hashable); this may be suboptimal for performance, but is easy to do
- Beware: the hash may be truncated on certain implementations of CPython, usually to the word-length of the target architecture, so care must be taken to make the hashes interoperable across different implementations of Python
- By default, sufficiently small integers are hashed with $\texttt{hash}(x) \equiv x \bmod m$ (with $m = 2^{61} - 1$ on x86-64 architecture)
- String hashes are salted by each process, so the hashes are nonpredictable across processes