1. Bitwise operators
   1. Operators
      1. &: AND
         1. Both
      2. |: OR
         1. Either 1
      3. ^: XOR
         1. Exactly 1 or the other
      4. >>: right bit shift
         1. Clip off bits
      5. << left bit shift
         1. Add 0s to the rightmost bit
   2. Examples: x = 13 [1101] & y = 6 [0110]
      1. x & y
         1. Do & for each digit from left to right
         2. x & y = 0100 = 4
      2. x | y = 1111 = 15
      3. x ^ y = 1011 = 11
      4. x << 3 = 1101000 = 104
      5. x >> 2 = 11 = 3
   3. Most hash functions use them.
      1. One-time pad: 2 unbreakable copies of a long random bitstring.
         1. Call XOR with secret key
            1. Encrypt: f(p) = p^k
            2. Decrypt: p = f(p)^k
         2. Example
            1. p = 110111101001
            2. k = 010110111010
            3. XOR = 100001010011
            4. k = 010110111010
            5. XOR = 110111101001
2. Hash table: table indexed 0 to *n* – 1.
   1. Insert record *x*: calculate f(x) = an int [0, *n* – 1], set table[f(x)] = x.
      1. f(“dog”) = 4
      2. f(“cat”) = 1
         1. If f is fast, we can search in O(1) time.
      3. f(“elephant”) = 5
      4. f(“hippo”) = 5
   2. Exam Example: hash function f(x) = (5x+3)%13
      1. Hash table
      2. Use \_\_\_ to insert…
3. Collision: hash functions are inherently many to one functions.
   1. Ways to deal with collisions
      1. Overwrite the data
         1. Not usually an option
         2. OK with cache
            1. Relatively small but fast block of memory.
         3. Functions
            1. Lookup [O(1)]
            2. Insert [O(1)]
            3. Delete [O(1)]
      2. Linear probing: if f(x) is full, go to f(x+1)%TABLE\_SIZE
         1. Advantages
            1. Guarantees to hit any open slot eventually.
         2. Problems
            1. Really bad system if table is near full.

Never fill >50%.

* + - * 1. Clustering: consecutive row of elements that have been filled; chance of hitting cluster is greater than of hitting an individual element.
      1. Functions
         1. Insert [O(1) 🡪 O(*n*)]
    1. Quadratic probing: if f(x) is full, go to f(x + i^2)%TABLE\_SIZE
       1. Advantages
          1. Skips rapidly over clusters.
       2. Problem
          1. Over large tables, this could loop forever even when there are open slots.

Never fill >50%.

Make it a prime number at least 2x the number of elements that will be stored.

* + - * 1. If table size is a large prime P, then the first (P-1)/2 attempts to insert all hit different locations.
      1. Method 1
         1. int slot = f(x);
         2. int tryval = slot, i = 1;
         3. while(array[tryval]!=NULL) {

tryval = (slot + i\*i)%SIZE;

i++;

* + - * 1. }
        2. array[tryval] = x;
      1. Method 2: exploit odds
         1. int slot = f(x);
         2. i = 1;
         3. while(array[slot]!=NULL){

slot = (slot + 2\*i – 1)%SIZE;

i+

* + - * 1. }
        2. array[slot] = x;
      1. Consider quadratic probing with table size P, where P is prime.
         1. After i collisions 🡪 look at p+i^2
         2. After j collisions, look at p+j^2
         3. Assume that there is a collision in the first (P-1)/2 locations.

Let 0≤i,j≤(P-1)/2.

Let location i and j be the same so let k = initial hash value.

k + i^2 = k + j^2 (mod p), i =/= j

(i-j)(i+j) = 0

Since i=/=j, impossible for i-j to be 0.

Contradiction

i+j>0

i+j<p

* + 1. Linear chaining hashing: hash table as array of linked lists.
       1. If hash function is good (equally-distributes items), most linked lists will be short.
          1. Normally insert to front.
       2. Works fairly well.
          1. Utilize all of our linked list code.

1. Using bitwise operators
   1. Programming contractor
      1. Turn integer i into a set of bits.
      2. Each of its bits is used to represent a subset as a Boolean array.