Hash tables + bitwise operators

Bitwise

&	bitwise and (both 1s)	x = 13 = 1101
|	bitwise or (either 1)	y = 6 = 0110
^	bitwise xor (exactly one 1)	x^y = 0100 = 4
>>	right bit shift	x | y = 1111 = 15
<<	left bit shift	x ^ y = 1011 = 11
		x << 3 = 1101000 = 104
		x >> 2 = 11 = 3

clip off bits

add 0s to right most bits

Most actual hash functions use a great deal of bitwise operators.

In crypto, one time pads are unbreakable. 2 copies of a long random bitstring, call it k,

To encrypt $f(p) = p \wedge k$ (xor, secret key)

To decrypt $p = f(p) \wedge k$

p = 110111101001

k = 010110111010

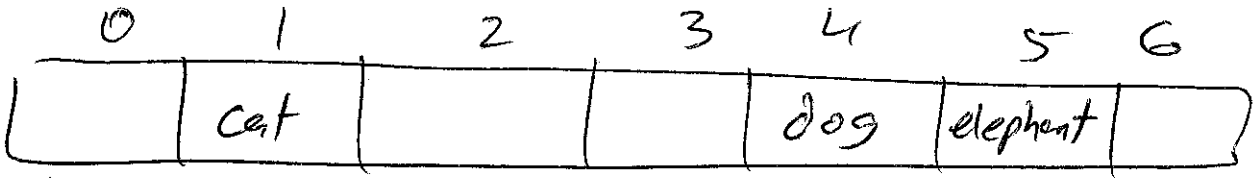
xor = 100001010011

k = 010110111010

xor = 110111101001

Idea of a Hash Table

Imagine a big table indexed 0 to n-1.
To insert a record, x, calculate f(x) = an int [0, n-1], set table[f(x)] = x



f("dog") = 4 → if f is fast to calculate we can search in O(1) time!
f("cat") = 1

f("elephant") = 5
f("hippo") = 5 → where does hippo go? does he fight w/ the elephant?

→ hash functions are inherently many to one functions (more than 1 input can map to the same output)

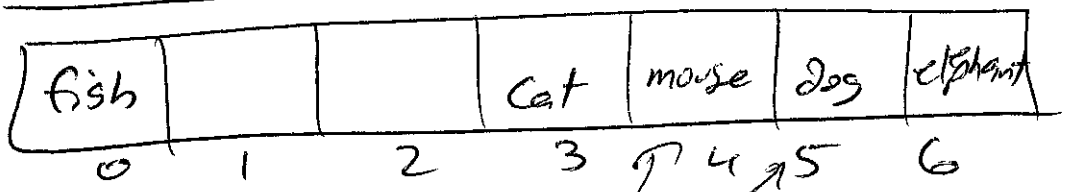
→ THIS IS A COLLISION!

WAYS TO DEAL WITH A COLLISION

① Don't - let the hippo eat the elephant (overwrite old data) → all runtimes are O(1)
② Linear Probing
③ Quadratic Probing
④ Linear Chaining Hashing

Linear Probing

4/11/17 (B)



f("cat") = 3

f("dog") = 5

f("elephant") = 5, → 6

f("mouse") = 4

f("~~s~~fish") = 4 → 5 → 6 → 0

if array[f(x)] is filled
try array[f(x)+1].
(f(x)+1) % tablesize
↑ wrap-around

Search for fish → f("fish") = 4

Search for alligator f("alligator") = 3 → 4 → 5 → 6 → 0 → 1
↳ NOT THERE

In worst case,
insert, search, delete are all O(n) time.

Rule - Never fill more than 50%.

Problem - clustering.

```
int slot = f(x)
while (array[slot] != NULL)
    slot = (slot+1) % size;
array[slot] = x;
```

pseudocode for insert using linear probing

Quadratic Probing

let slot = f(x).

Try inserting in the following positions:

slot, (slot+1) % size, (slot+4) % size,

(slot+9) % size, ... (slot + i*i) % size, ...

size = 19

slot = 7 → 8 → 11 → ~~17~~ 16 → 4

(1) (3) (5) (7)

```
int slot = f(x);
int tryval = slot, i=1;
while (array[tryval] != NULL) {
    tryval = (slot + i*i) %size;
    i++;
}
array[tryval] = x;
```

```
int slot = f(x);
 i = 1;
while (array[slot] != NULL) {
    slot = (slot + 2*i - 1) %size;
    i++;
}
array[slot] = x;
```

linear probing was guaranteed to hit any open slot eventually. But could this loop forever, even when there are open slots?

If the table size is a large prime p, then the first $\frac{p-1}{2}$ attempts to insert all hit different locations.

→ to do this table must be < 50% full

4/11/17 (3)

8, 9, 12, 17, 24, (33), 44

17, 18, 21, 26, (33), 42

Consider quadratic probing w/table size p, p←Prime. After i collisions we look at $p+i^2$
j collisions we look at $p+j^2$.

Assume there's a collision in the 1st $\frac{p-1}{2}$ locations let $0 \le i, j \le \frac{p-1}{2}$. Let location i and j be the same so, let k = initial hash function value.

$$k + i^2 \equiv k + j^2 \pmod p, \quad i \neq j.$$

$$i^2 \equiv j^2 \pmod p$$

$$i^2 - j^2 \equiv 0 \pmod p$$

$$(i-j)(i+j) \equiv 0 \pmod p$$

↳ since $i \neq j$, $i - j \neq 0$.

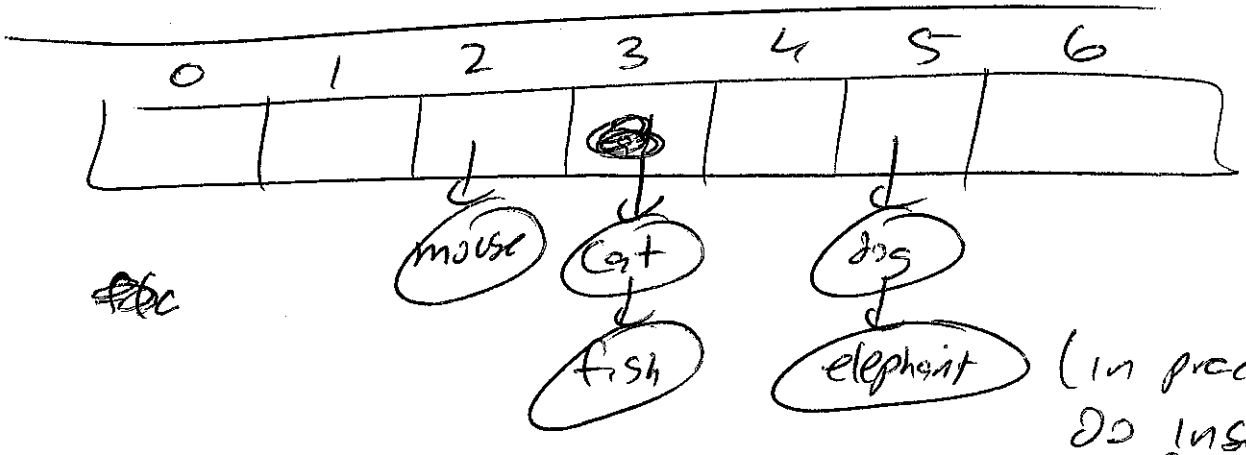
either $p \mid (i-j)$ or $p \mid (i+j)$

$p \mid (i-j)$ false since
$i \neq j$
$0 \le i, j \le \frac{p-1}{2}$

$i+j > 0$
$i+j < p$
↳ IMPOSSIBLE CONTRADICTION.

4/11/17 (16)

Linear Chaining Hashing

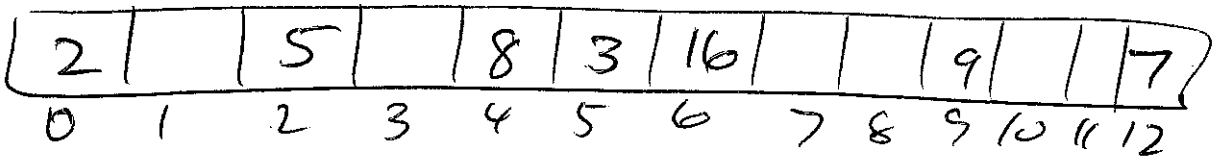


(in practice I do insert front)

$f("cat") = 3$
$f("dog") = 5$
$f("elephant") = 5$
$f("mouse") = 2$
$f("fish") = 3$

If your hash function is good (equally distributes items), most linked lists will be quite short.
Utilize all of our linked list code.

let hash function be $f(x) = (5x+3) \% 13$



Use quadratic probing
insert 3, 8, 2, 9, 16, 5 and 7

$3 \rightarrow 5(3)+3 = 18 \% 13 = 5$
$8 \rightarrow 5(8)+3 = 43 \% 13 = 4$
$2 \rightarrow 5(2)+3 = 13 \% 13 = 0$
$9 \rightarrow 5(9)+3 = 48 \% 13 = 9$

$16 \rightarrow 5(16)+3 = 83 \% 13 = 5$
$5 \rightarrow 5(5)+3 = 28 \% 13 = 2$
$7 \rightarrow 5(7)+3 = 38 \% 13 = 12$

23 = 10111

4/11/17 (9)

linear probing

$f(x) = ((x << 1) \wedge 23) \% 32$

insert 16, 5, 27, 19 table size = 32

16 = 100000
$\wedge$ 10111

110111
= 55 % 32
= **23**

5 << 1 = 1010
10111

11101
29 % 32

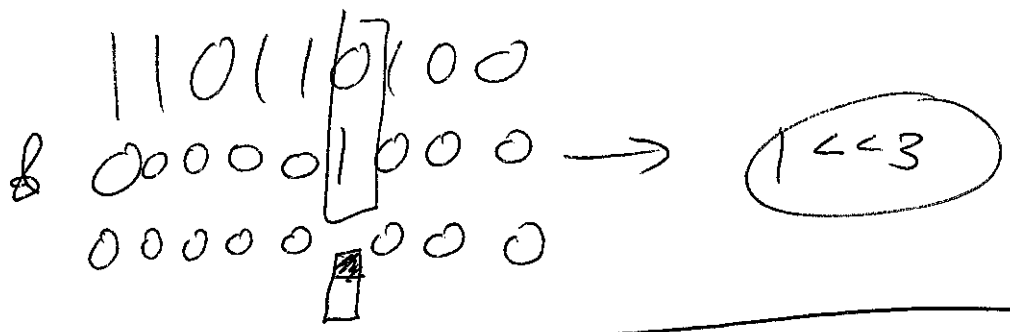
27 = 11011
27 << 1 = 11011 0
23 = 10111

100001
= 33 % 32
= (1)

19 = 10011
19 << 1 = 10011 0
23 = 10111

110001
= 49 % 32
= **17**

Useful Notes About using bitwise ops



---

n & (1 << i)

is 0 if bit i is off in n

is non-zero if bit i is on in n

↓

2^i

---

a & b is 0 if the subsets represented by a and b are disjoint. It's also the intersection of those 2 sets

---

a | b is the union of sets a and b

---

if ((a & (1 << i)) != 0)

then a - (1 << i) is the set a with item i removed.