CS 2100 - Hashing (part 2)

Note Title 4/29/2011

Announcements

Data Storage

Ex:

Locker #	Name
26	Dan
355	Kevin
101	Tracy
53	Nitish
201	David
⋮	⋮

We want to be able to retrieve a name quickly when given a locker number.

(Let n = # of people, &
m = # of lockers)

$m \geq n$

Dictionaries

A data structure which supports the following:

locker#   Name

```
void     insert (keyType &k, dataType &d)
dataType find (keyType &k)
void     remove (keyType &k)
```

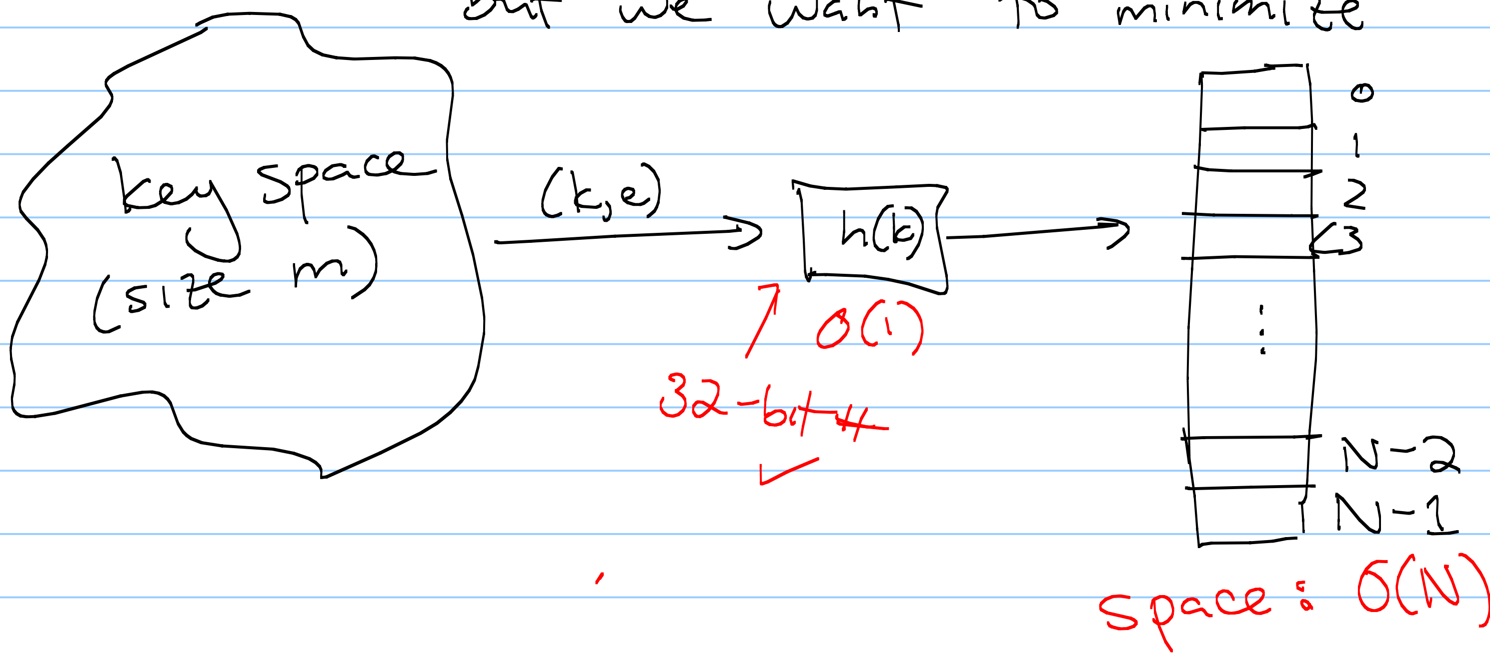
Note: Everything is based on keys!

Don't know keyType - might not correspond to an int.

Good hash functions:

- Are fast   goal: $O(1)$
- Don't have collisions when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$

  these are unavoidable, but we want to minimize



Step 1: Get a number (& avoid collisions)

char (32-bits) $\longrightarrow$ ASCII

float (64-bits) $\underbrace{\boxed{\quad\quad\quad}}_{a}\underbrace{\boxed{\quad\quad\quad}}_{b}$

$a+b = 32\text{-bits}$

Ex:

```
int hashCode(long x) {
    return int(unsigned long(x >> 32)
                  + int(x));
}
```

What about strings?

(Think ASCII.)

Erin

69 + 114 + 105 + 110 = 32-bit single representation

fast

Goal: a single int.

But, in some cases, a strategy like this can backfire:

temp01 and temp10 and pm0te1

collide under simple XOR

We want to avoid collisions between "similar" strings (or other types).

A Better Idea: Polynomial Hash Codes

Pick $a \neq 1$ and split data into k 32-bit parts: $x = (x_0, x_1, x_2, x_3, \ldots, x_{k-1})$

input

Let $p(x) = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$

Ex: Erin with $a = 37$

x_0

$p(\text{"Erin"}) = 69 \cdot 37^3 + 114 \cdot 37^2 + 105 \cdot 37 + 110$

Side Note: How long does this take?

(In terms of $k = \#$ of parts)

$$h(x) = \underbrace{x_0 a^{k-1}}_{\substack{k-1 \\ \text{mult.}}} + \underbrace{x_1 a^{k-2}}_{k-2} + \cdots + \underbrace{x_{k-2} a}_{1} + \underbrace{x_{k-1}}_{0 \text{ mult.}}$$

$+ \; k-1$ additions

Alternate idea:

Horner's rule: $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots))$

$k-1$ mult & $k-1$ additions

Polynomial Hashing

This strategy makes it less likely that similar keys will collide.

(Works for floats, strings, etc.)

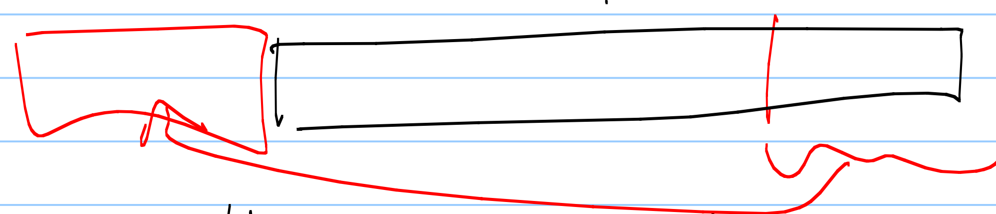
What about overflow?

truncate, XOR, ...

Cyclic shift hash codes

Alternative to polynomial hashing

Instead of multiplying by a^p, shift each 32-bit piece by some # of bits.



Also works well in practice.

Step 2: Compression maps

Now we can assume every key k is an integer.

Need to make it between 0 & $N-1$ (not 0 and 2^{32}).

Goal: Find a "good" map.

"Good": - fast

- minimize collisions

Modular compression maps

Take $h(k) = k \bmod N$

What does mod mean again?

$3 \bmod 10 =$

$50 \bmod 10 =$

$14 \bmod 10 =$

Example: $h(k) = k \bmod 11$

A:

0	1	2	3	4	5	6	7	8	9	10

Insert: (12, E)
(21, R)
(37, I)
(16, N)
(26, C)
(5, H)

Some Comments:

This works best if the size of the table is a prime number.

Why?

Go take number theory & cryptography

Strategy 2: MAD (multiply, add & divide)

First idea: take $h(k) = k \bmod N$

Better: $h(k) = |ak+b| \bmod N$

where a & b are:

- not equal
- less than N
- relatively prime

(Why? Go take number theory!)

Example:

$$h(k) = |ak+b| \bmod 11$$

$$a = 3$$
$$b = 5$$

A:

0	1	2	3	4	5	6	7	8	9	10

Insert:
(12, E)
(21, R)
(37, I)
(16, N)
(26, C)
(5, H)

This is a lot of work!
Why bother?

In practice, drastically reduces collisions.

End Goal: Simple Uniform Hashing Assumption

For any $k \in$ key space,

$$\Pr[h(k) = i] = 1/N$$

(Essentially, elements are "thrown randomly" into buckets.)

Collisions

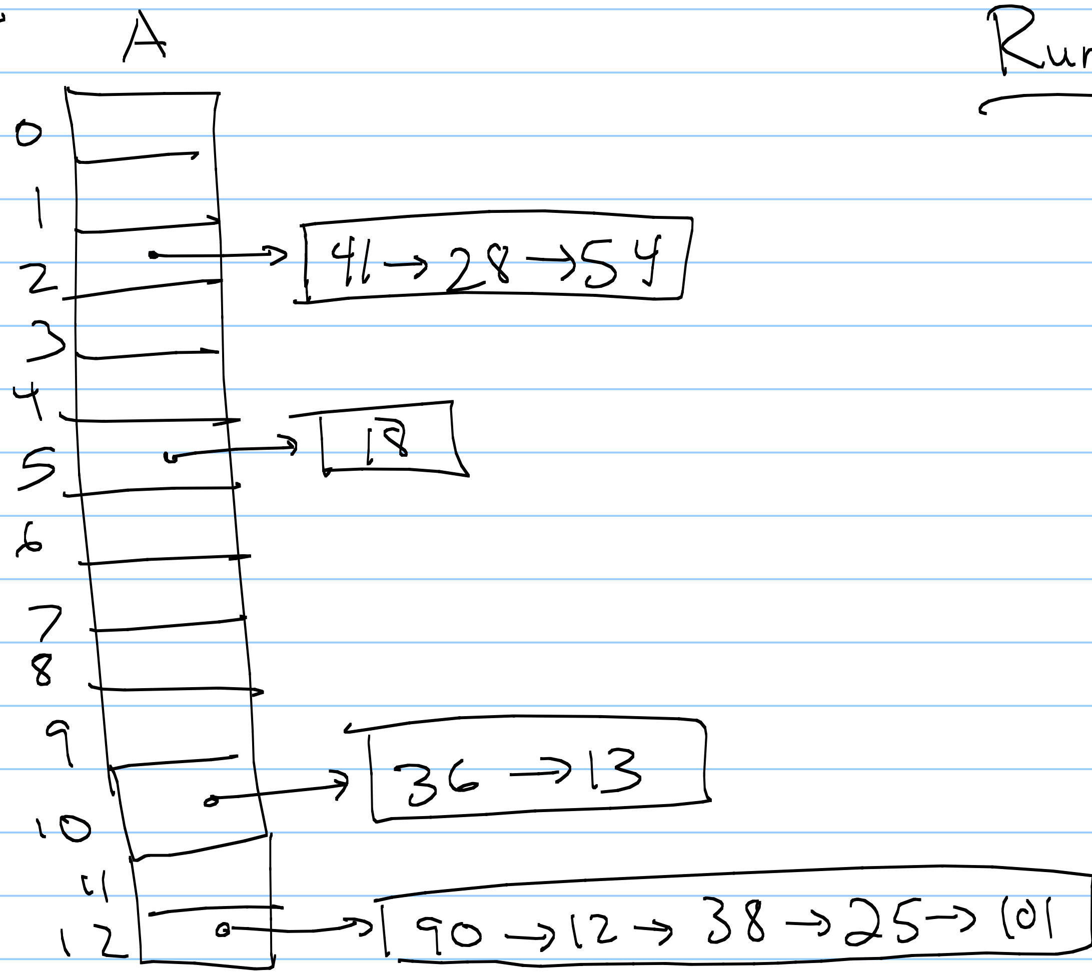
Can we ever totally avoid collisions?

Step 3: Handle collisions (gracefully & quickly)

So how can we handle collisions?

[Hint: Do we have any data structures that can store more than 1 element?]

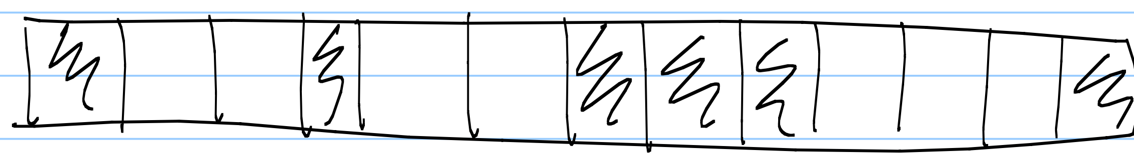
Ex:



Running times:

Linear Probing

Instead of lists, if we hash to a full spot, just keep checking next spot (as long as the next spot is not empty).



Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10

Insert:
(12, E)
(21, R)
(37, I)
(26, N)
(16, C)
(5, H)
(15, A)

Issue

How can we remove here?

If you remove, create "gap" that linear probing won't know was full at time of insertion.

Solution: "dirty bit":

Running Time for Linear Probing

Insert:

Remove:

Find:

Quadratic Probing

Linear probing checks $A[h(k)+1 \bmod N]$
if $A[h(k) \bmod N]$ is full.

To avoid these "primary clusters", try:

$$A[h(k) + j^2 \bmod N]$$

where $j = 0, 1, 2, 3, 4, \ldots$

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10

Insert:
(12, E)
(21, R)
(37, I)
(26, N)
(16, C)
(5, H)
(15, A)
(4, M)

Issues with Quadratic Probing:

- Can still cause "secondary" clustering
- N really must be prime for this to work
- Even with N prime, starts to fail when array gets half full

(Runtimes are essentially the same)