CS180 - Hashing (part 2)

Note Title 4/29/2011

Announcements

- HW 10 up, checkpoint Tues. after break
- HW9 due Sat.
- Class as usual on Monday

Data Storage

keys data

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

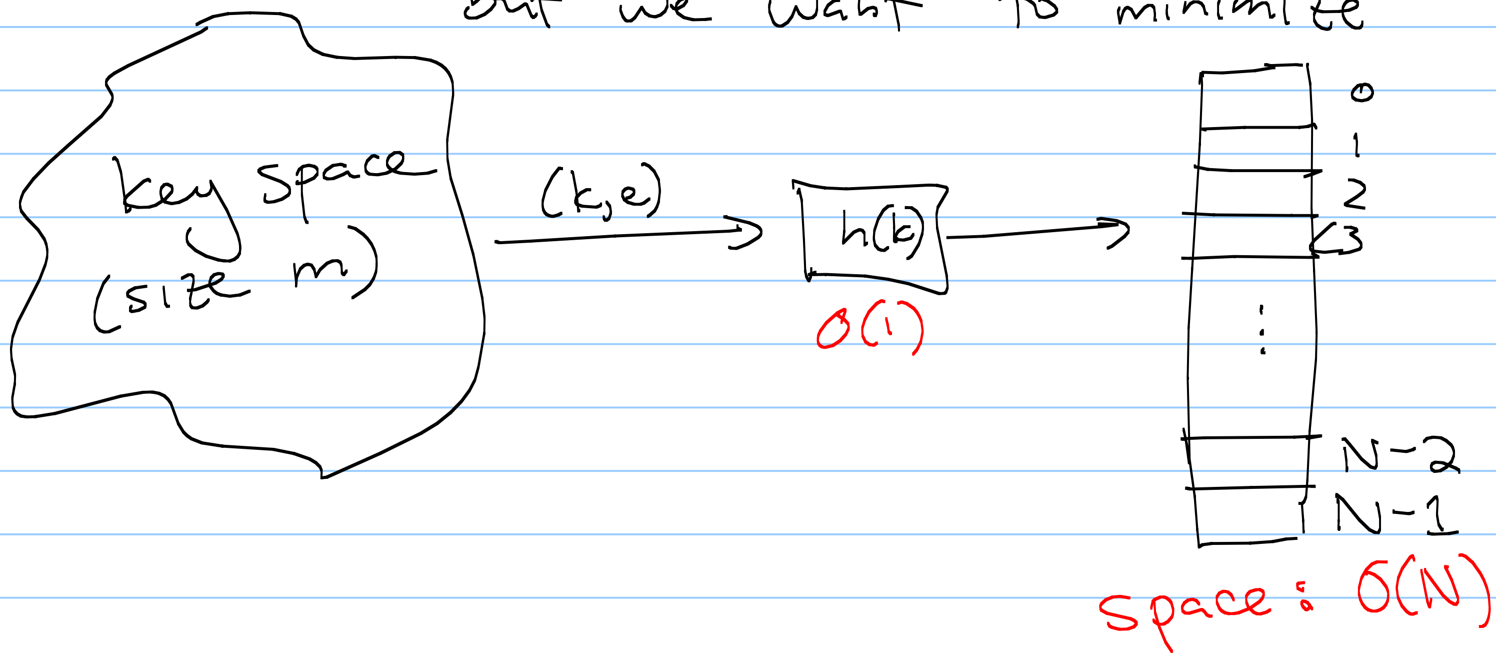
```
void     insert (keyType &k, dataType &d)
dataType find   (keyType &k)
void     remove (keyType &k)
```

Note: Everything is based on keys!

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

Good hash functions:

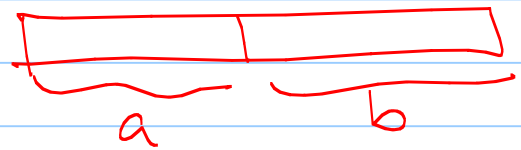
- Are fast goal: $O(1)$
- Don't have collisions ← when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$
  these are unavoidable, but we want to minimize



Step 1: Get a number (& avoid collisions)

char (32-bits) → ASCII

float (64-bits)



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

constant

Goal: Permutations won't collide.

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

Ex: Erin with $a = 37$

69 114 105 110

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

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

Side Note: How long does this take?

(In terms of k = # of parts)

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

$k-2$ 1 0

$$\sum_{i=1}^{k} i = \Theta(k^2) \text{ multiplications}$$

Alternate idea:

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

$O(k)$ additions & mult.

Polynomial Hashing

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

(Works for floats, strings, etc.)

What about overflow?

37^{10} is huge

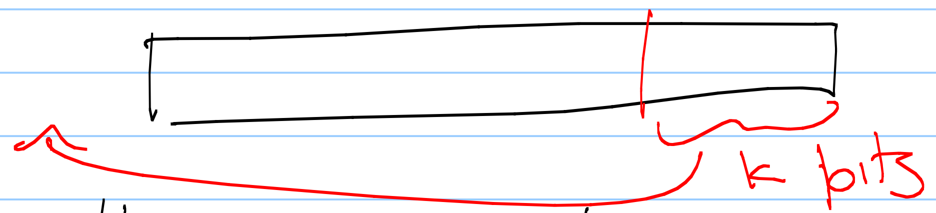
(integers stop at 2^{32})

truncate!

Cyclic shift hash codes

Alternative to polynomial hashing

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

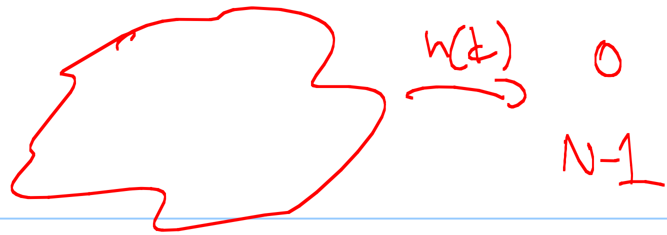


Also works well in practice.

Advantage: fast.

$\underset{3}{E} + \underset{2}{r} + \underset{1}{i} + \underset{0}{n}$ $\quad\quad$ $\underset{3}{r} + \underset{2}{i} + \underset{1}{E} + \underset{0}{r}$

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?

remainder

$3 \bmod 10 = 3$

$50 \bmod 10 = 0$

$14 \bmod 10 = 4$

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

collision

A:

0	1	2	3	4	5	6	7	8	9	10
	(12, E)			(37, I)	(16, N)					(21, R)

key, data

Insert:

- (12, E) — $h(12) = 12 \bmod 11 = 1$
- (21, R) — $h(21) = 21 \bmod 11 = 10$
- (37, I) — $h(37) = 4$
- (16, N) — $h(16) = 5$
- (26, C) — $h(26) = 4$
- (5, H)

(still need collision strategy ...)

Some Comments:

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

Why?

Go take number theory & Cryptography

Idea: more "prime" numbers are less likely to have things collide

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

no common divisors

15, 8

(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
		(21, R)						(12, E)		

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

$h(12) = 3 \cdot 12 + 5 \bmod 11 = 8$

$h(21) = 3 \cdot 21 + 5 \bmod 11 = 2$

→ collisions are much less likely

This is a lot of work!

Why bother?

In practice, drastically reduces collisions.

a & b can be small in practice.

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.)

Impossible in practice, still goal we work towards.

Collisions

Can we ever totally avoid collisions?

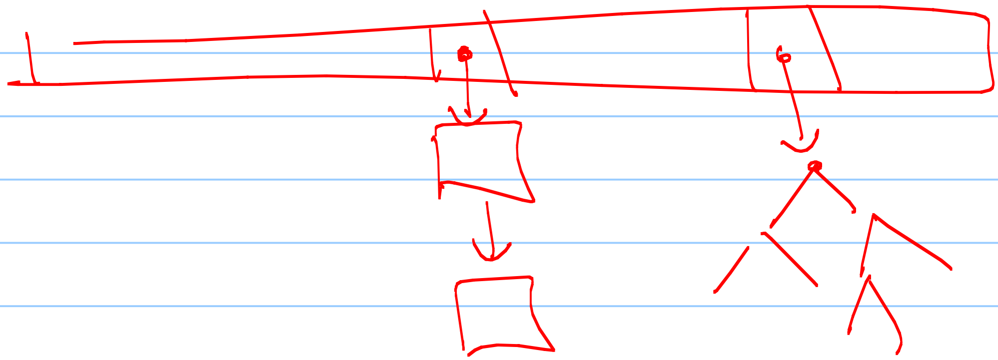
NO

Step 3: Handle collisions (gracefully & quickly)

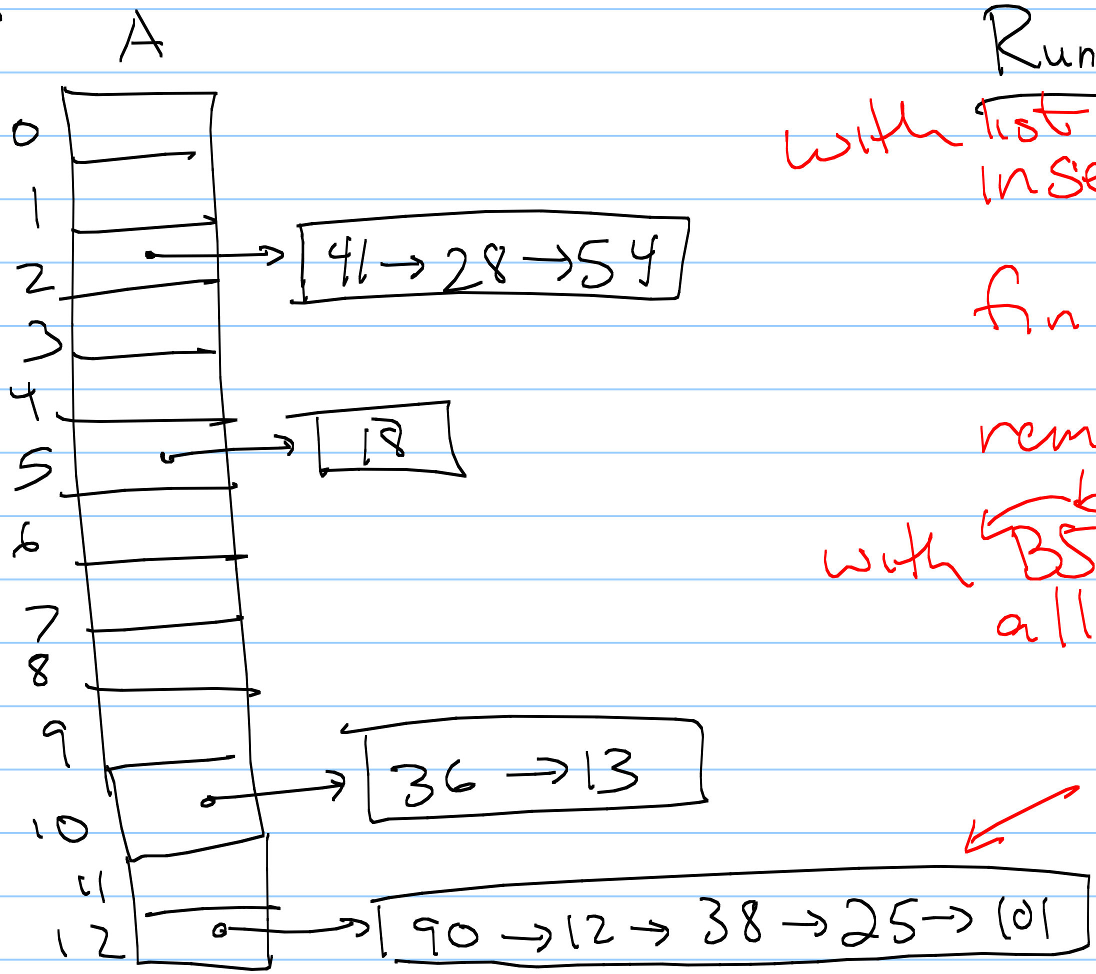
So how can we handle collisions?

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

- list
- tree



Ex:



Running times:

with list:

insert: $O(1)$

find: $O(n)$

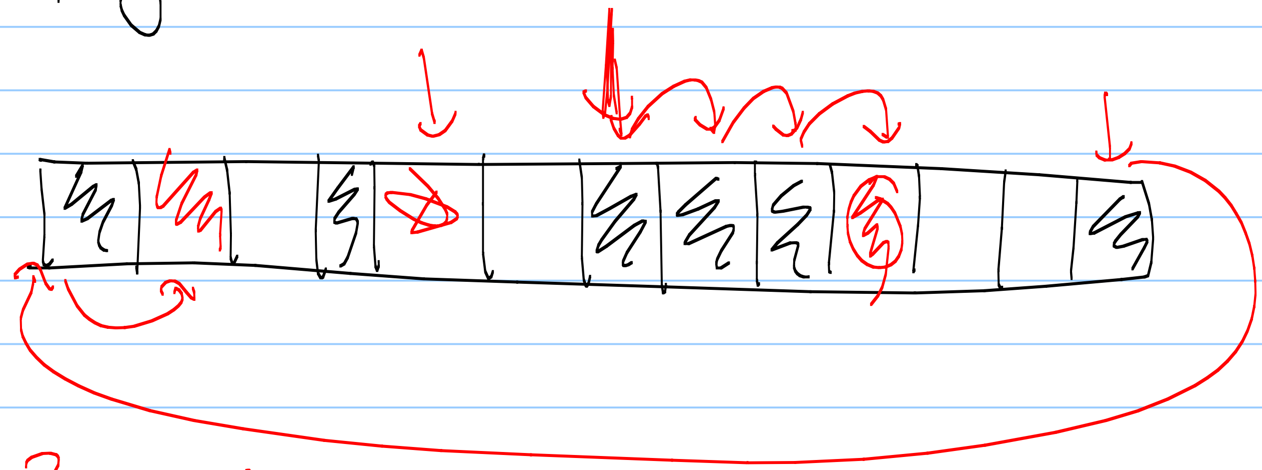
remove: $O(n)$

with balanced BST:

all: $O(\log n)$

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).



find? O(n)

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
	(12, E)			(37, I)	~~(26, N)~~	(16, C)	(5, H)	(15, A)		(21, R)

Insert:

(12, E) $\Rightarrow h(12) = 1$

(21, R) $h(21) = 10$

(37, I) $h(37) = 4$

(26, N) $h(26) = 4$

(16, C) $h(16) = 5$

(5, H) $h(5) = 5$

(15, A) $h(15) = 4$

find(17, Z) : $O(n)$

delete(26)

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":

don't actually remove, instead have a bit that gets flipped when value is removed

Running Time for Linear Probing

Insert: $O(n)$

Remove: $O(n)$

Find: $O(n)$

worst case

in practice: fast

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

if $A[h(k)]$ is full

if $A[h(k)+1]$ is full

$A[h(k)+2^2] = A[h(k)+4]$

$A[h(k)+3^2] = A[h(k)+9]$ ...

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
	(12, E)		(4, M)	(37, I)	(26, N)	(16, C)		(15, A)	(5, H)	(21, R)

Insert:

(12, E)	$h(12) = 1$
(21, R)	$h(21) = 10$
(37, I)	$h(37) = 4$
(26, N)	$h(26) = 4$ full $\rightarrow$ $h(26) + 1^2 = 5$
(16, C)	$h(16) = 5$ full $\rightarrow$ $h(16) + 1^2$
(5, H)	$h(5) = 5$ full, $h(5)+1 \rightarrow$ full, $h(5)+4$
(15, A)	$h(15) = 4$ $h(15)+1$,
(4, M)	$h(4) = 4$

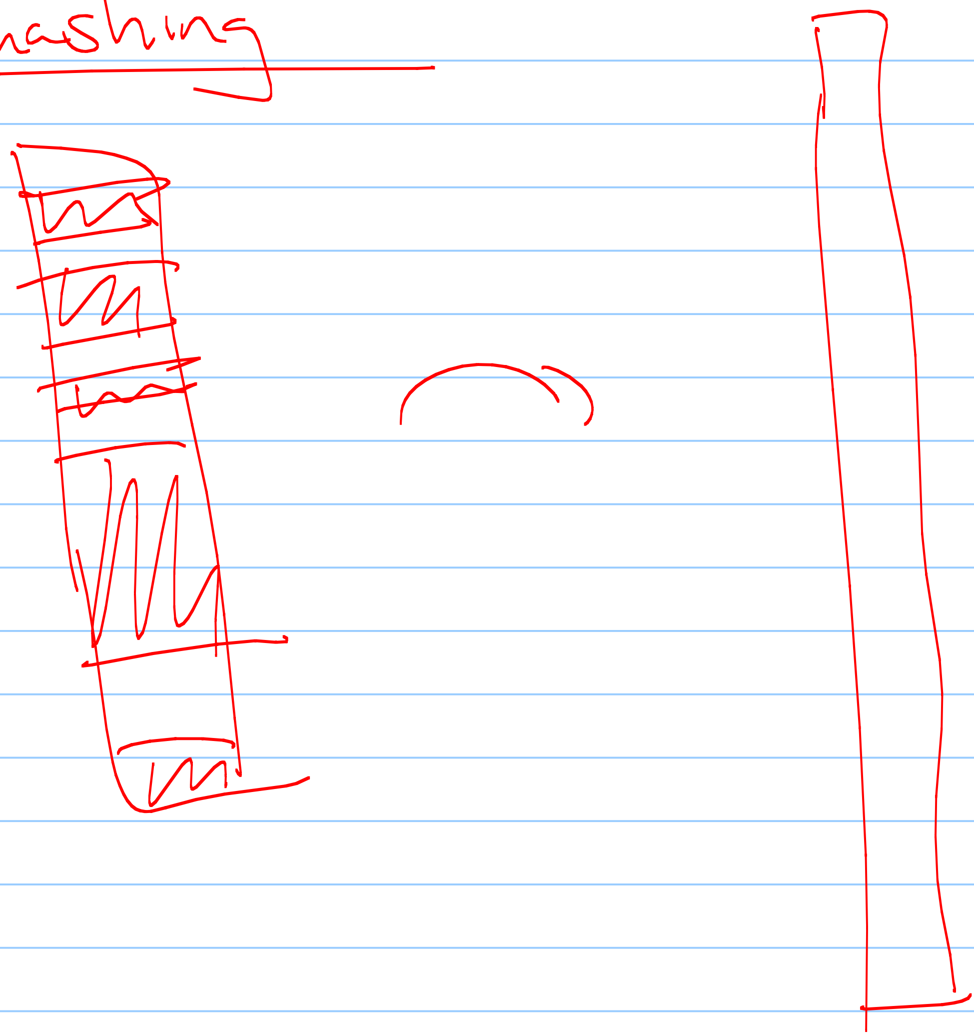
Issues with Quadratic Probing:

- 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 $n > \frac{N}{2}$

(Runtimes are essentially the same)

Rehashing

(twice as big)



pick new a + b

compute h'(all entries)