CS180 – Hashing (part 3)

Note Title 5/2/2011

Announcements

- Checkpoint on Monday
- 1 more HW due final day of class

InBitStream

```
BinaryTree<      >   mytree;
int input;
InBitStream  variable;

variable.open("banana.myzip");

input = variable.read();  //will be 0 for root

if (input == 0)
    mytree.Create root(  );
```

(note - can draw tree!)

Data Storage - Dictionary : insert, find, remove

Ex:

key

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

data

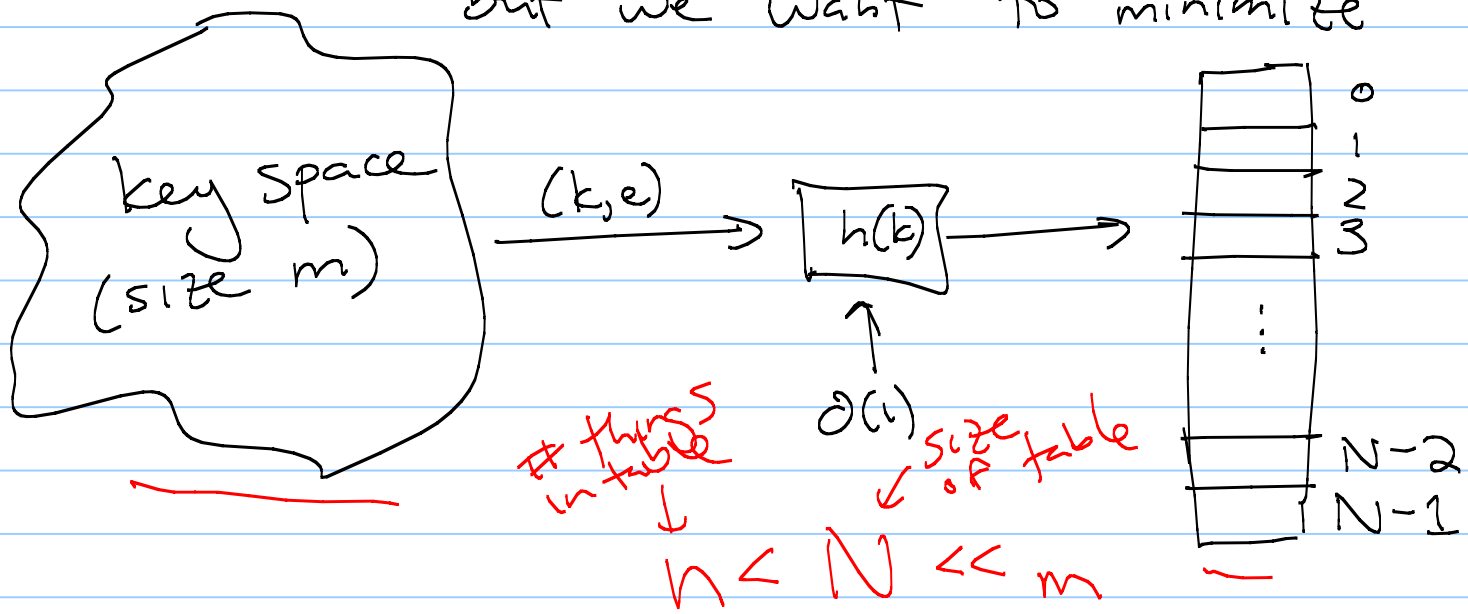
We want to be able to retrieve a name quickly when given a locker number.

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

$n \geq m$

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: Turn key into an integer

Cyclic permutations
or polynomial

Step 2: Compression map

takes us to a value in $[0, \ldots, N-1]$

% N

MAD, etc.

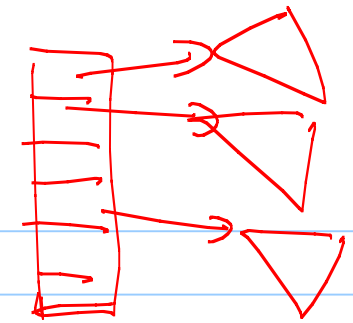
Collisions

Can we ever totally avoid collisions?

No

m is bigger than n or N

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?]

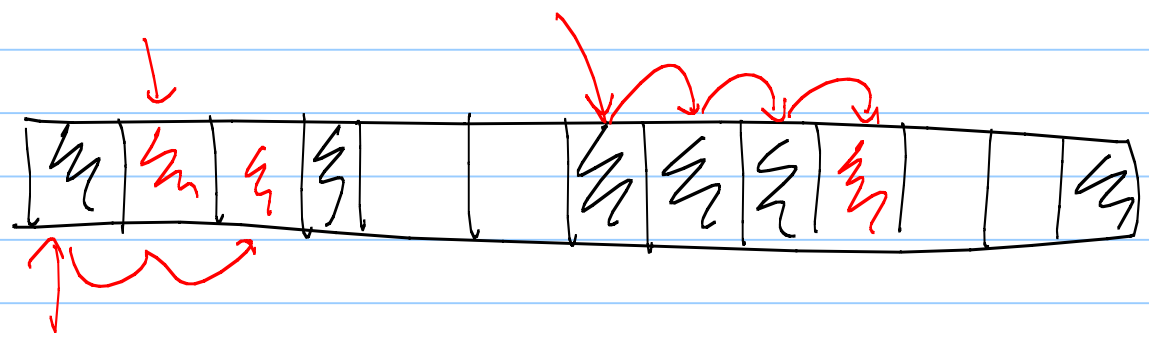
Possibilities:

Store in auxilary DS, such as

- Vector
- List
- AVL trees

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).



$h(k) = (k + i) \bmod 11$

$i = 0, 1, 2, \ldots$

Example

$h(k) = k \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
M	E			~~I~~	N	C	H	A		R

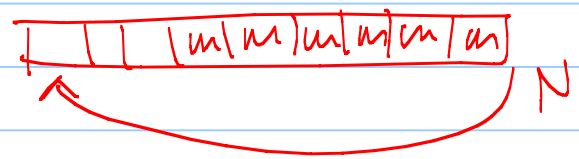
Insert:

(12, E)	12 mod 11 = 1	
(21, R)	21 mod 11 = 10	
remove → (37, I)	37	= 4
(26, N)	26	= 4
(16, C)	16	= 5
(5, H)	5	= 5
→ (15, A)	15	= 4
(10, M)		

find (48, –)

Running Time for Linear Probing

Insert:



← contains n things

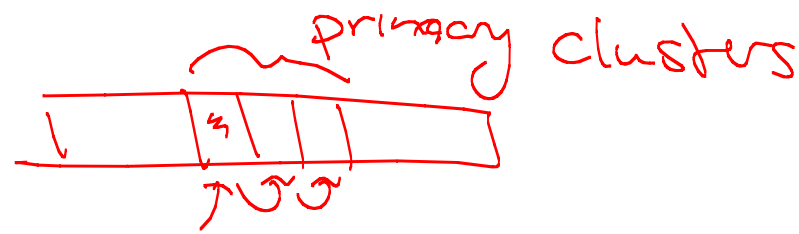
$O(n)$

Remove: $O(n)$ dirty bit

Find: $O(n)$

Issues with linear probing

- "Clusters" form
  - worse if #'s not "good" in hash function
  - terrible when array nears ½ full
- Removing doesn't actually reduce # of elements - just sets the "dirty" bit.



Quadratic Probing

Linear probing checks $A[h(k)+j \bmod N]$ if previous spot is full (for $j = 1, 2, \ldots$)

To avoid primary clusters, try

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

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

$A[h(k)]$ is full

— try $A[h(k)+1]$

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

$A[h(k)+9]$

Example

$h(k) = k \bmod 11$

secondary clusters

0	1	2	3	4	5	6	7	8	9	10
	E	M		I	N	C		A	H	R

Insert:

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

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

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

(26, N) $h(26) = 4 \rightarrow 4 + 1^2$

(16, C) $h(16) = 5 \rightarrow 5 + 1^2$

(5, H) $h(5) = 5 \rightarrow 5 + 1^2 \rightarrow 5 + 2^2$

(15, A) $h(15) = 4 \rightarrow 4 + 1^2 \rightarrow 4 + 2^2$

(4, M) $h(4) = 4 \rightarrow 5 \rightarrow 8 \rightarrow 4 + 3^2$ ✓

Issues with Quadratic Probing:

- Can still cause secondary clustering
- N <u>really</u> must be prime for this to work
- Even with N prime, starts to fail when array gets half full
- Can fail entirely even if array not full.

(Runtimes are essentially the same)

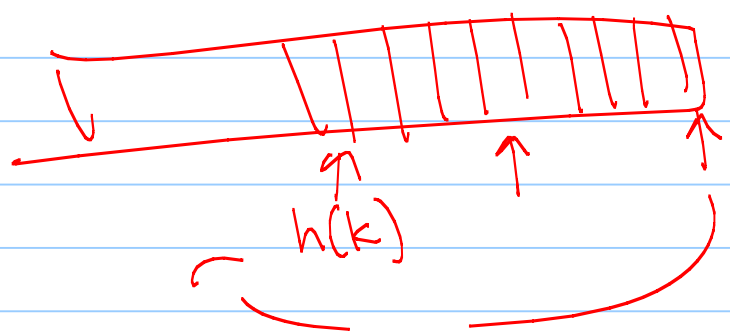
Secondary Hashing

- Try $A[h(k)]$
- If full, try $A[h(k) + f(j) \bmod N]$ for $j = 1, 2, 3, \ldots$

new value

where

$f(j) = j \cdot \ell(k)$ with ℓ a different hash function



spps $h'(k) = 4$

Load Factors

use a list as aux. data structure

Separate chaining actually works as well as most others in practice, although it does use more space.

Most of these methods only work well if $\frac{n}{N} < .5$.

(Even chaining starts to fail if $\frac{n}{N} > .9$)

$\frac{n}{N}$ load factor

Rehashing

Because we need $\frac{n}{N} < .5$, most hash code checks if the array has become more than half full.

If so, it stops & recomputes everything for a larger N, usually at least twice as big.

(Still not too bad in an amortized sense – think vectors.)