CS180 - Hashing (part 3)

Note Title 5/2/2011

Announcements

- Checkpoint due today
- Program due Friday
- Last HW out today
  due Monday (not graded)

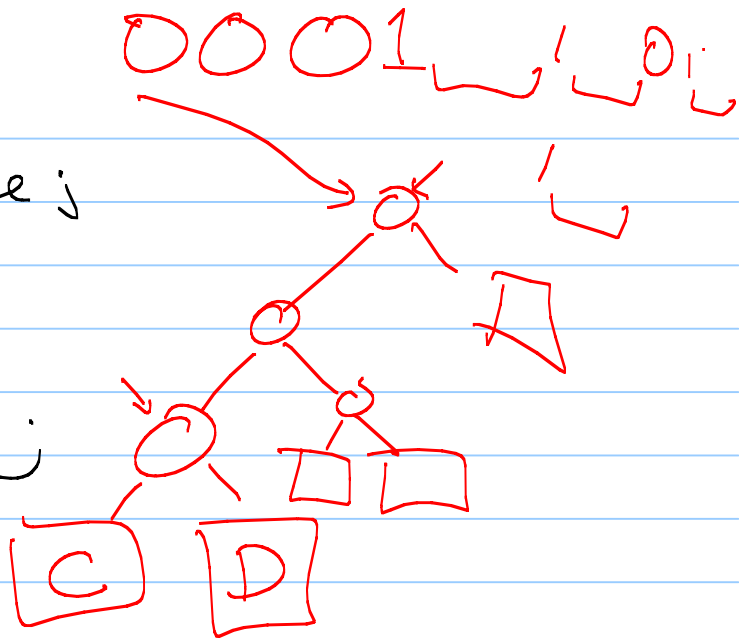
InBitStream

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

variable.open("banana.myzip");

input = variable.read();

if (input == 0)
    mytree.createroot(' ');
```



Data Storage - Dictionary

insert
find
remove

Ex.

Locker # (key)	Name (data)
26	Dan
355	Kevin
101	Tracy
53	Nitish
201	David
⋮	⋮

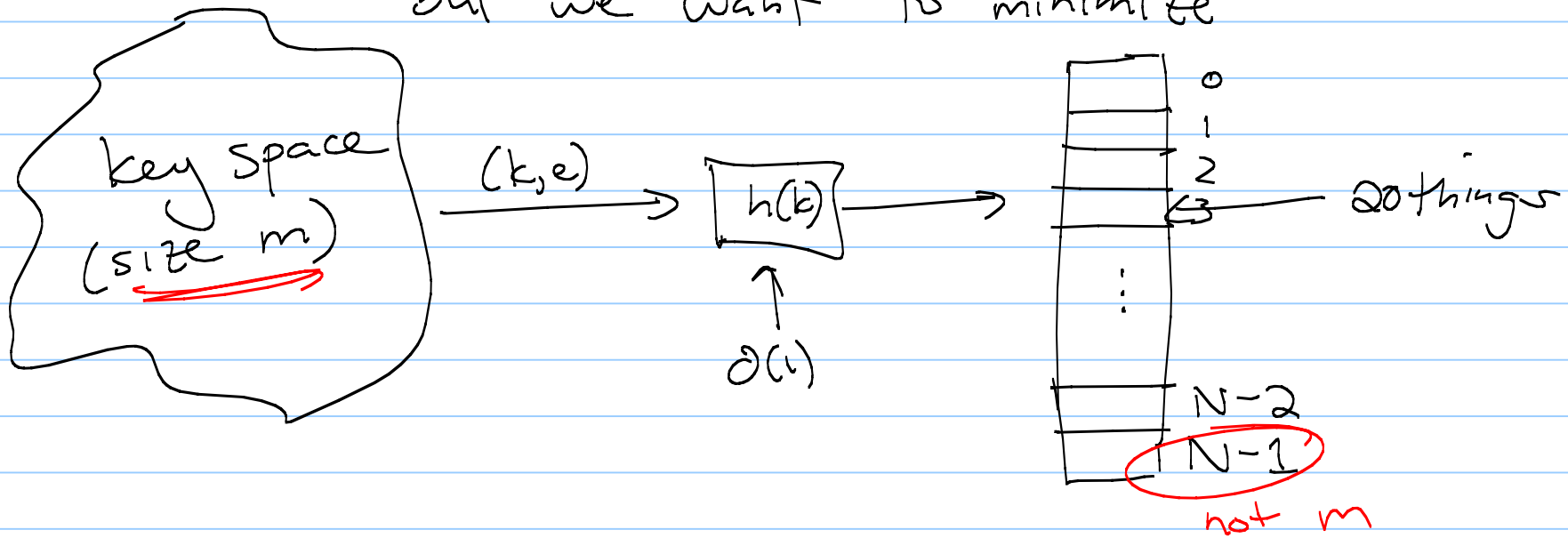
We want to be able to retrieve a name quickly when given a locker number.

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

$n \geq m$

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

polynomial hashing => integer (32-bits)

Step 2: Compression map

Take integer and make it < N.

mod (or %)

MAD

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?]

inserted a list in each array space

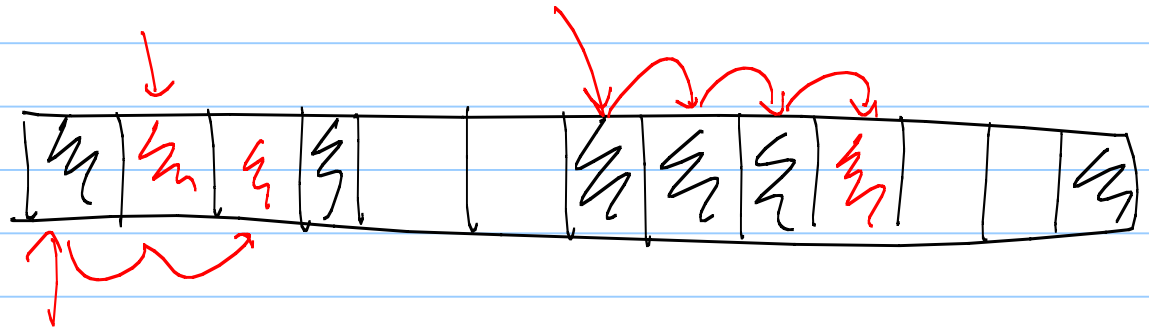
lists = separate chaining

O(n)

AVL tree
Vector

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).

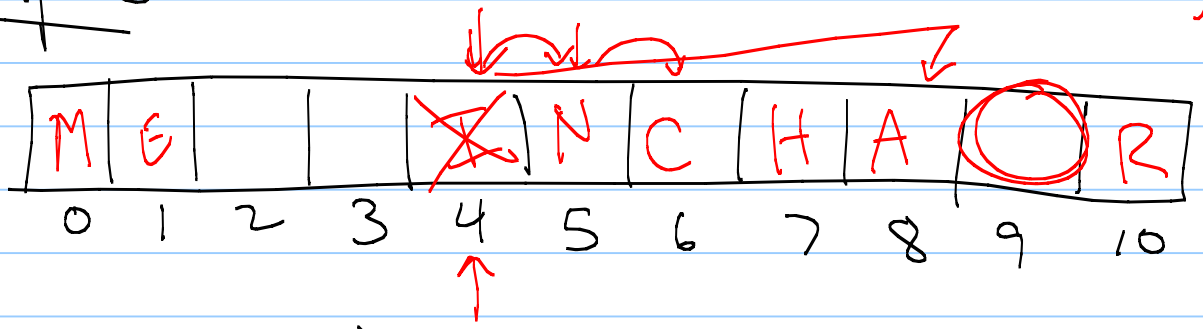


$h(k) = (k + i) \bmod 11$

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

Example

$h(k) = k \bmod 11$



Insert: (12, E)

(21, R)

remove → (37, I)

(26, N)

(16, C)

(5, H)

→ (15, A)

(10, M)

12 mod 11 = 1

21 mod 11 = 10

37 = 4

26 = 4

16 = 5

5 = 5

15 = 4

find (48, –)

Running Time for Linear Probing

Insert: $O(n)$ (not $O(N)$)

n = # of elements in array

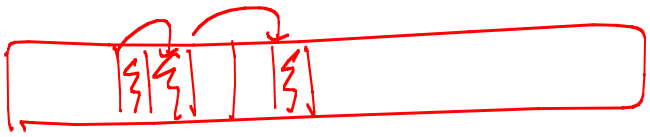
Remove: remove bit (don't actually remove)

"dirty bits"

$O(n)$ if enough of array is dirty, rehash everything

Find: hash to value & keep checking next spot while next not empty, not found &

$O(n)$



Quadratic Probing

Linear probing checks $A[h(k)+j \bmod N]$ for $j = 1, 2, 3, 4, \ldots$ if $A[h(k) \bmod N]$ is full.

primary clustering

To avoid clusters, try

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

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

$h(k)$

$h(k) + 1^2 = h(k) + 1$

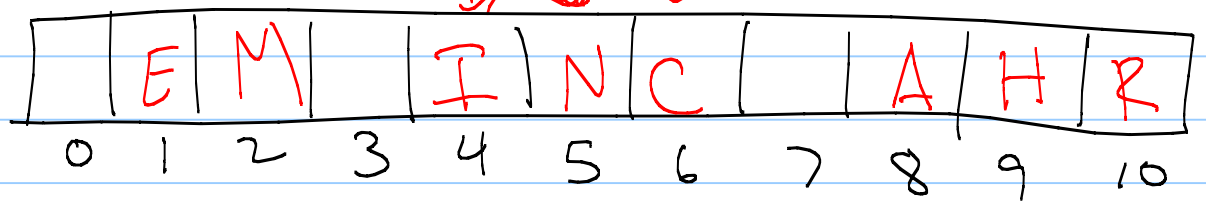
$h(k) + 2^2 = h(k) + 4$

$h(k) + 3^2 = h(k) + 9$

$\vdots$

Example

$h(k) = k \bmod 11$



Insert:

(12, E)	$12 \bmod 11 = 1$
(21, R)	$21 = 10$
(37, I)	$37 = 4$
(26, N)	$26 = 4$
(16, C)	$16 = 5$
(5, H)	$5 = 5$
(15, A)	$15 = 4$
(4, M)	$4 = 4$

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

(Runtimes are essentially the same)

Can fail entirely!

Secondary Hashing

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

where $f(j) = j \cdot h'(k)$ with h' a different hash function

Best if we want to avoid clusters.

Load Factors

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 $\frac{n}{N} > .9$)

usually $.25 < \frac{n}{N} < .5$

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.)