Dictionaries: Hashing, Amortization, and other magic

(there is no magic)

Ilia Lebedev
ilebedev@mit.edu

Bio

Hello!

Context

Part of a data structure - centric Algorithms course (mid-semester)

Prerequisites:

- Basic data structures
- Basic complexity

Handout: http://people.csail.mit.edu/ilebedev/facebook

Big picture

Build a data structure with:
O(lg n) insert, delete
Θ(1) : current 5 median values

TODAY!

1. Dictionary ADT
2. Hashing
3. Amortized complexity

.. as illustrated by table doubling

Too easy?

important for:

Cryptography
Distributed systems
Databases
Image search
etc. (much of CS!)

if time permits:

Open addressing
Rolling hashes
Crypto hashing

Data structures cheat sheet

	Insert	Delete	Search
Array	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
List	$\Theta(1)$	$O(n)$	$O(n)$
Sorted array list*	$O(\log n)$	$O(n)$	$O(\log n)$
Min/Max heap	$O(\log n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Today: Dictionary	$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$

Motivating problem

Given a “document” (list of words),
find the most often-occurring word.

*Also Facebook could not possibly work without hash tables!

Ready? Set. Code! (1/3)

$O(n^2)$


```
def common_word( W ):
    counts = []
    best = [None, 0]
    for w in W:
        for pair in counts:
            if pair[0] == w:
                pair[1] = pair[1] + 1
                if pair[1]>best[1]:
                    best=pair
                break
        counts.append( [w, 1] )
    return best[0]
```

Ready? Set. Code! (2/3)

O(n log n)


```
def common_word( W ):
    SW = sorted(W)
    best = [None, 0]
    current = [None, 0]
    for w in SW:
        if w != current[0]:
            if current[1]>best[1]:
                best=current
            current = [w, 0]
        current[1] = current[1]+1
    return best[0]
```

Ready? Set. Code! (3/3)

Use a heap to store counts?

→ $O(n^2)$

Use a BST? **(augmented to store words)

→ O(n log n)

Can we do any better?

Finding the bottleneck

Output is a reduction of **entire** input

$\rightarrow \Omega(n)$ complexity

To find most often occurring word,

we **count** all the words

So far, O(n log n) at best.

(we can do better)

Given the counts, can find largest in O(n)

The dictionary ADT

```
def count( W ):
   counts = {}
   for w in W:
      count = counts[w]
      if not count: count = 0
      counts[w] = count + 1
   return counts
```

- search (key) → value
- insert (key, value)

  delete (key)

All can be implemented in $\Theta(1)^*$

Pre-hash functions (all keys are now ints)

→ (pre-hash) → int

Ideally:

== for “same”

not == otherwise

not injective!

hash(“\0B”) ==

hash(“\0\0C”)

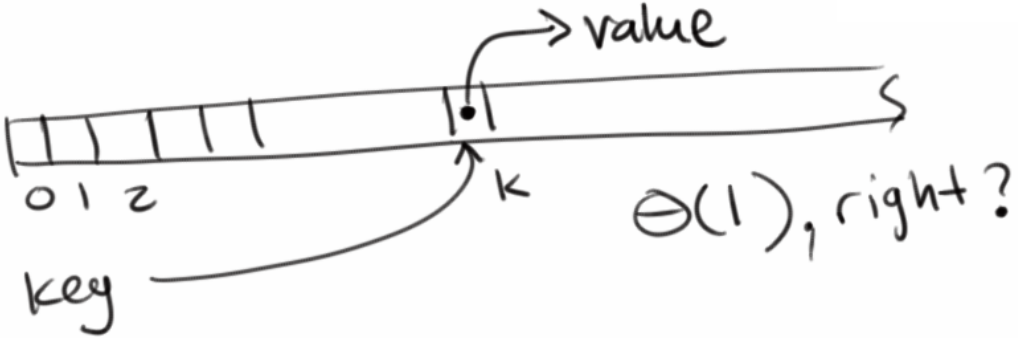
An unbounded array "dictionary"

- all keys are ints $\in Z^+$
- keys index into ∞ array A

search: A[k] $\rightarrow$ v

insert: A[k] = v

delete: A[k] = None

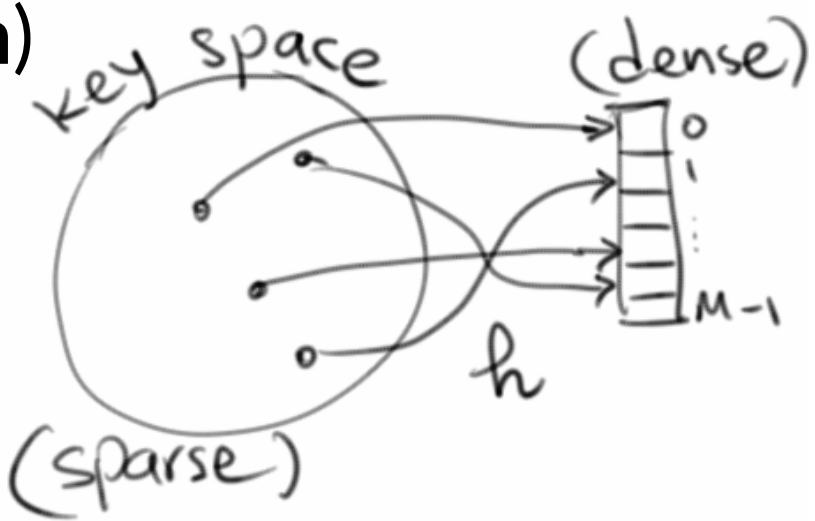


Hashing to "compress" key space

Key space is large and sparse.

Map to dense array!

h: key space → array indexes (mod **m**)



search: A[h(k)] → v

insert: A[h(k)] = v

delete: A[h(k)] = None

Key collision

Augment a dictionary such that it maintains a list of its elements

Pigeonhole principle

Key space is large, but {0,1, .. m-1} is small

Too easy?

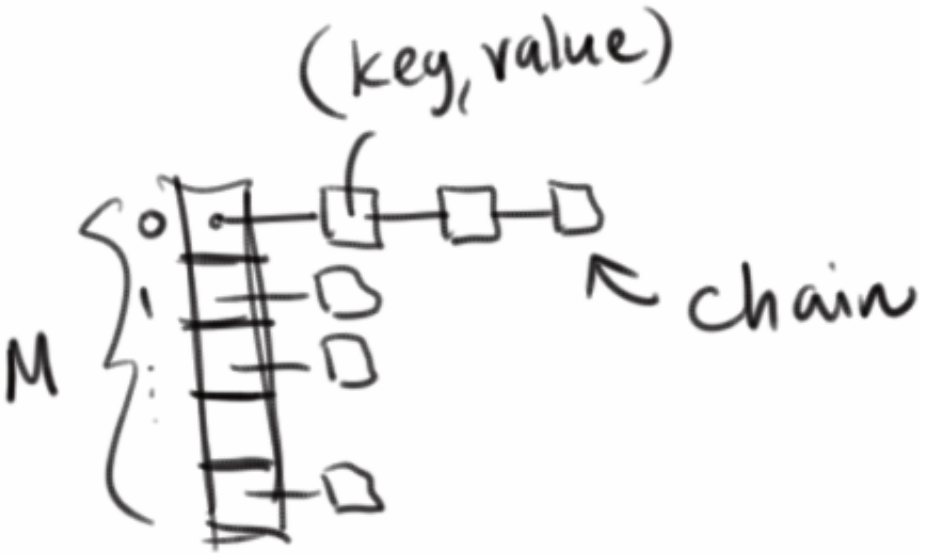
There must exist some (many) $k_1, k_2 \mid h(k_1)==h(k_2)$

A hash table with chaining

Collisions will happen!

Chaining:

store lists (key, value)



*other solutions are very cool!

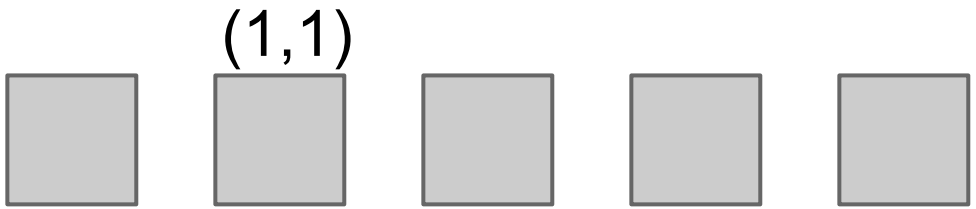
- Open addressing, Cuckoo hashing, ...

Chaining example (1/6)

Insert **(1,1)**,(21,21),(33,33),(11,11),(31,31)

Delete (11,11)

h(k) = k%5

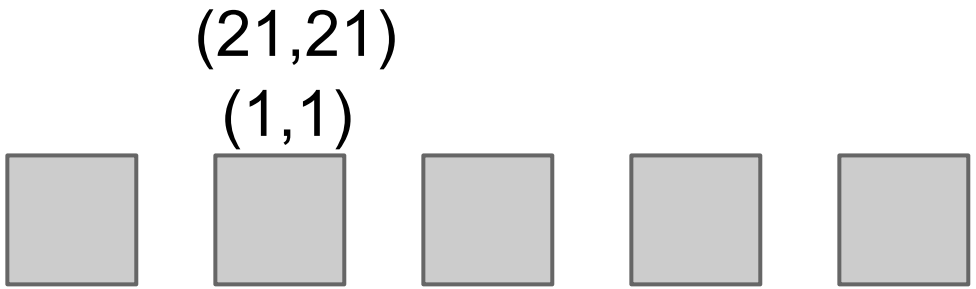


Chaining example (2/6)

Insert **(1,1),(21,21)**,(33,33),(11,11),(31,31)

Delete (11,11)

h(k) = k%5

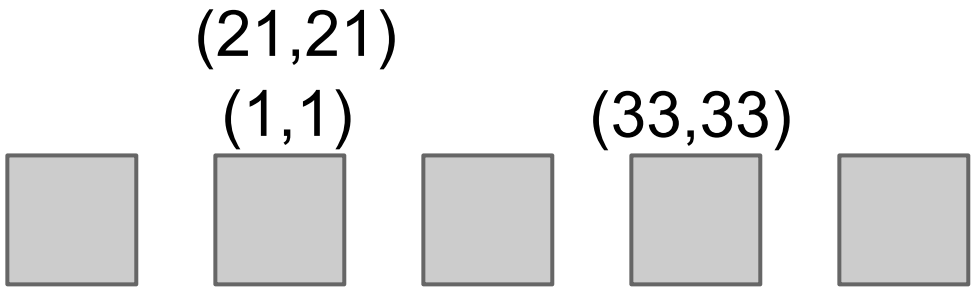


Chaining example (3/6)

Insert **(1,1),(21,21),(33,33)**,(11,11),(31,31)

Delete (11,11)

h(k) = k%5

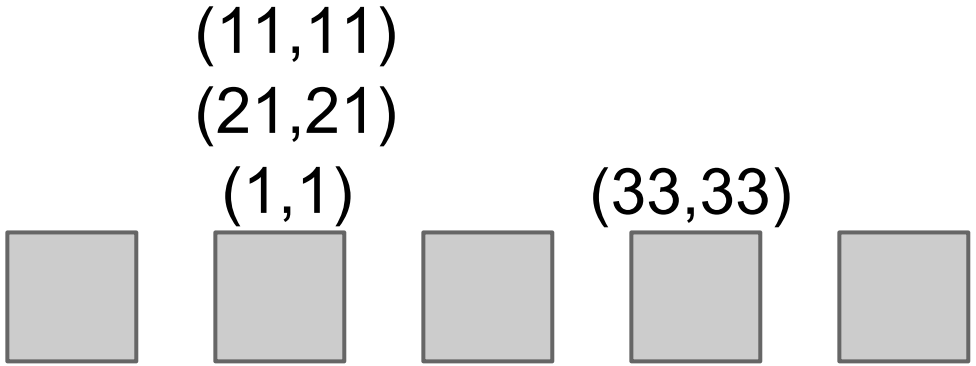


Chaining example (4/6)

Insert **(1,1),(21,21),(33,33),(11,11)**,(31,31)

Delete (11,11)

h(k) = k%5

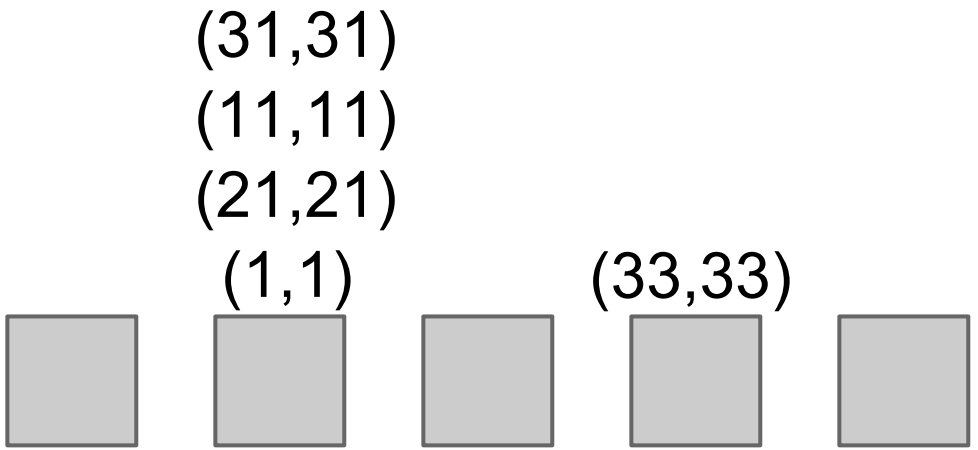


Chaining example (5/6)

Insert **(1,1),(21,21),(33,33),(11,11),(31,31)**

Delete (11,11)

h(k) = k%5

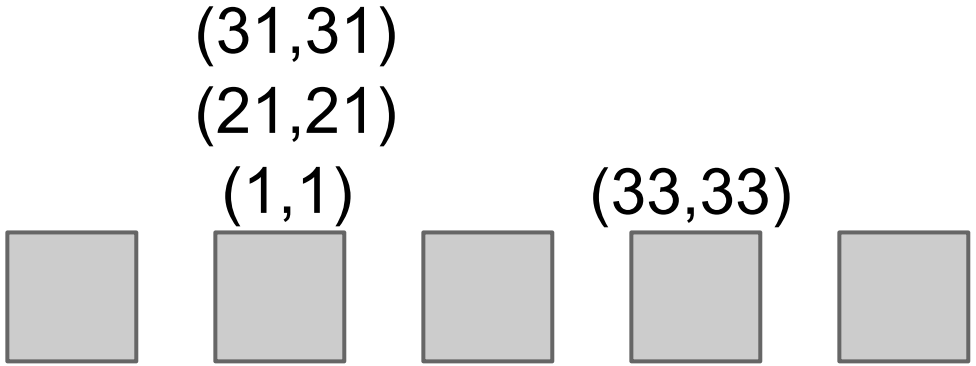


Chaining example (6/6)

Insert (1,1),(21,21),(33,33),(11,11),(31,31)

Delete (11,11)

h(k) = k%5



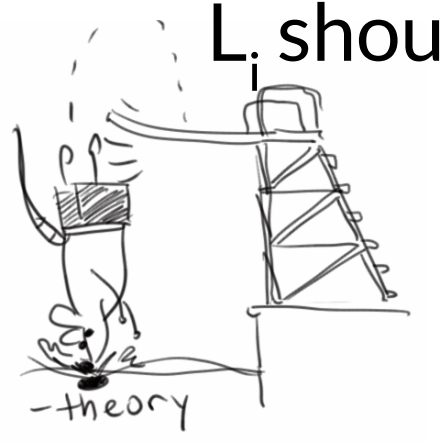
Time complexity with chaining

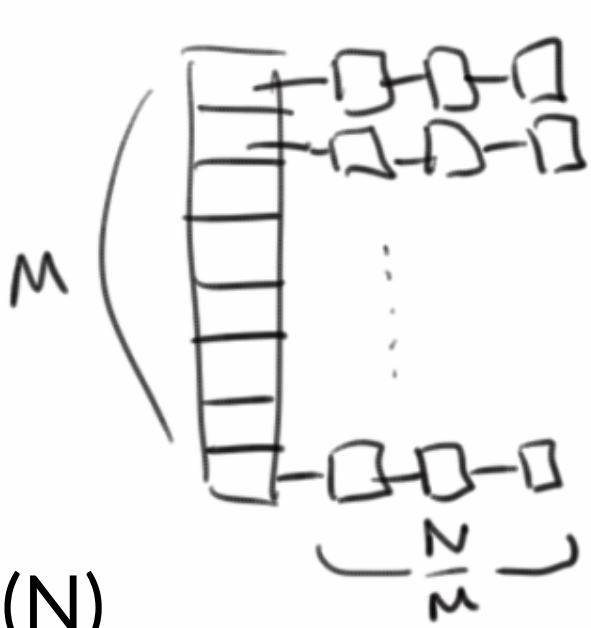
$O(L_i)$ time to traverse the list, where L_i is $|chain_i|$

Best case : all L_i are the same

chains have N/M pairs

L_i should be kept short!



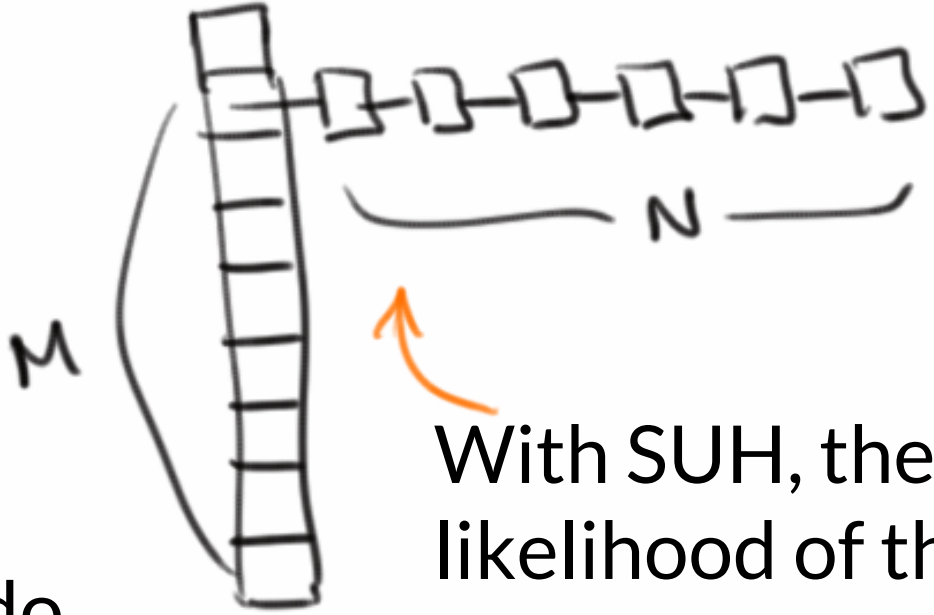


$\Theta(1)$ search $\rightarrow$ M = $\Theta(N)$

The worst case

What is the worst h?

O(N) worst case?!



Pick h such that this is unlikely. Best we can do, but it works out.

With SUH, the likelihood of this happening is $1/(M^{(n-1)})$ - tiny!

Many simple hash functions do well

- division method: $h(k) = k \bmod m$
- multiplication method: $h(k) = (k*a \bmod 2^w) >> (w-r)$, where **w** is the word width, and **m** = 2^r
- universal hashing: $h(k) = ((a*k+b) \bmod p) \bmod m$

(random) (random) (prime)

Demo!

Various simple hash functions perform well against biased distributions. (not crypto-quality but good enough).

Taking stock

what's missing?

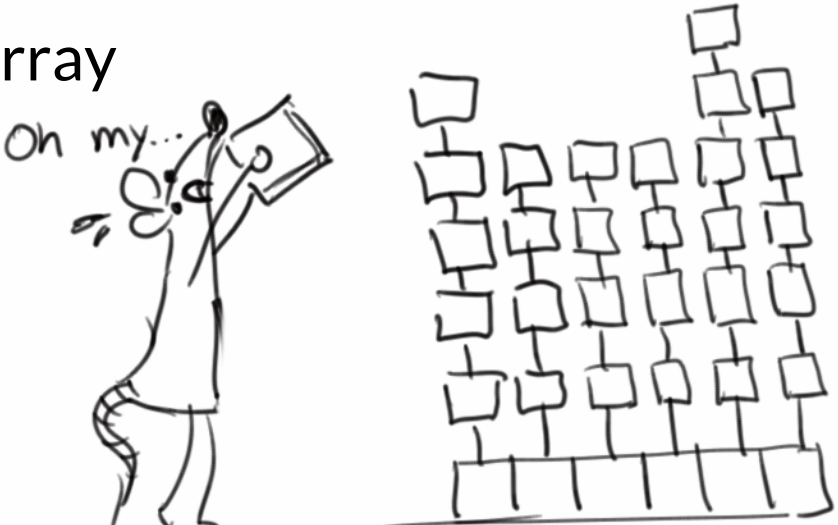
any solutions
to challenges?

Resizing M

Recall M = $\Theta(N)$. If N = M^2, then L=$\Theta(\sqrt{N})$, not const!
We need to resize the array as N grows!

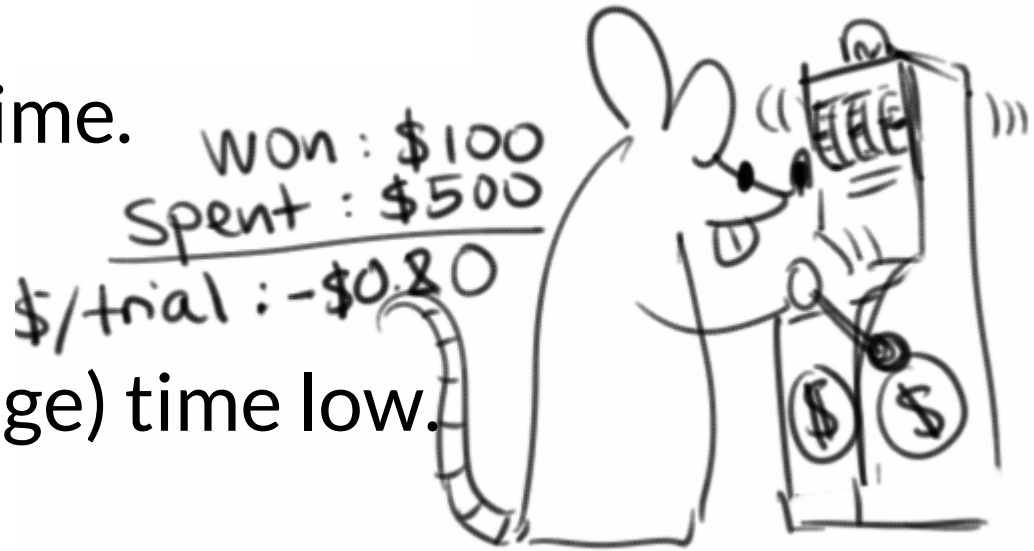
Resizing:
1). Make a new (bigger/smaller) array
2). Make a new hash function
3). Re-hash everything

How long does this take?



Amortized time complexity

Resizing takes O(N) time.



Use <u>amortization</u>
keep expected (average) time low.

(O(1)Θ(M) + O(N)) / Θ(M)
= Θ(M+N)/Θ(M) = Θ(1)

Spread out expensive ops (**amortize** them)

When to resize on insert and by how much?

Let's resize when **n > m**

m → m+1?

No, Θ(M) inserts must happen before next resize.

m → 2m

at least **m/2** inserts before next resize

Resizing on deletions

Tricky case: [ins, del, ins, del, …] at threshold

a resizing *del* must allow $\Theta(M)$ *ins* before resizing

$m \rightarrow m/2$

when $n < m/4$

$m \rightarrow m/2$

when $n < m/2$

is BAD!

Hash table with chaining (1/5)

Now, lets put it all together!

```
class HashTable:
  def __init__(self):
    self.min_m = 4
    self.m = self.min_m
    self.n = 0
    self.D = [ [] ] * self.m
    self.h =
     lambda k : ((k*73+37)%91)%self.m
```

Hash table with chaining (2/5)

```
def search ( self, k ):
  chain = self.D[self.h(k)]
  for (Dk, Dv) in chain:
    if Dk == k:
      return Dv
  return None
```

Hash table with chaining (3/5)

```
def insert ( self, k, v ):
  chain = self.D[self.h(k)]
  for pair in chain:
    if pair[0] == k:
      pair[1] = v
      return
  chain.append( [k, v] )
  self.n = self.n + 1
  if self.n > self.m:
    self._resize( 2*self.m )
```

Hash table with chaining (4/5)

```
def delete ( self, k ):
  chain = self.D[self.h(k)]
  m = self.m
  for pair in chain:
    if pair[0] == k:
      pair[0] = chain[-1][0]
      pair[1] = chain[-1][1]
      chain.pop()
      self.n = self.n - 1
      if (m > self.min_m) and (self.n < m/4):
        self._resize( m/2 )
      return
```

Hash table with chaining (5/5)

```
def resize ( self, new_m ):
  elements = []
   for chain in self.D:
      for element in chian:
         elements.append( element )
  self.D = [ []*new_m ]
  self.m = new_m
  for (k,v) in elements:
     self.insert( k, v )
```


wow
wow
so hash
uch complexity
much table
wow
wow
any questions?