6.006 Recitation

Build 2008.14

Coming up next...

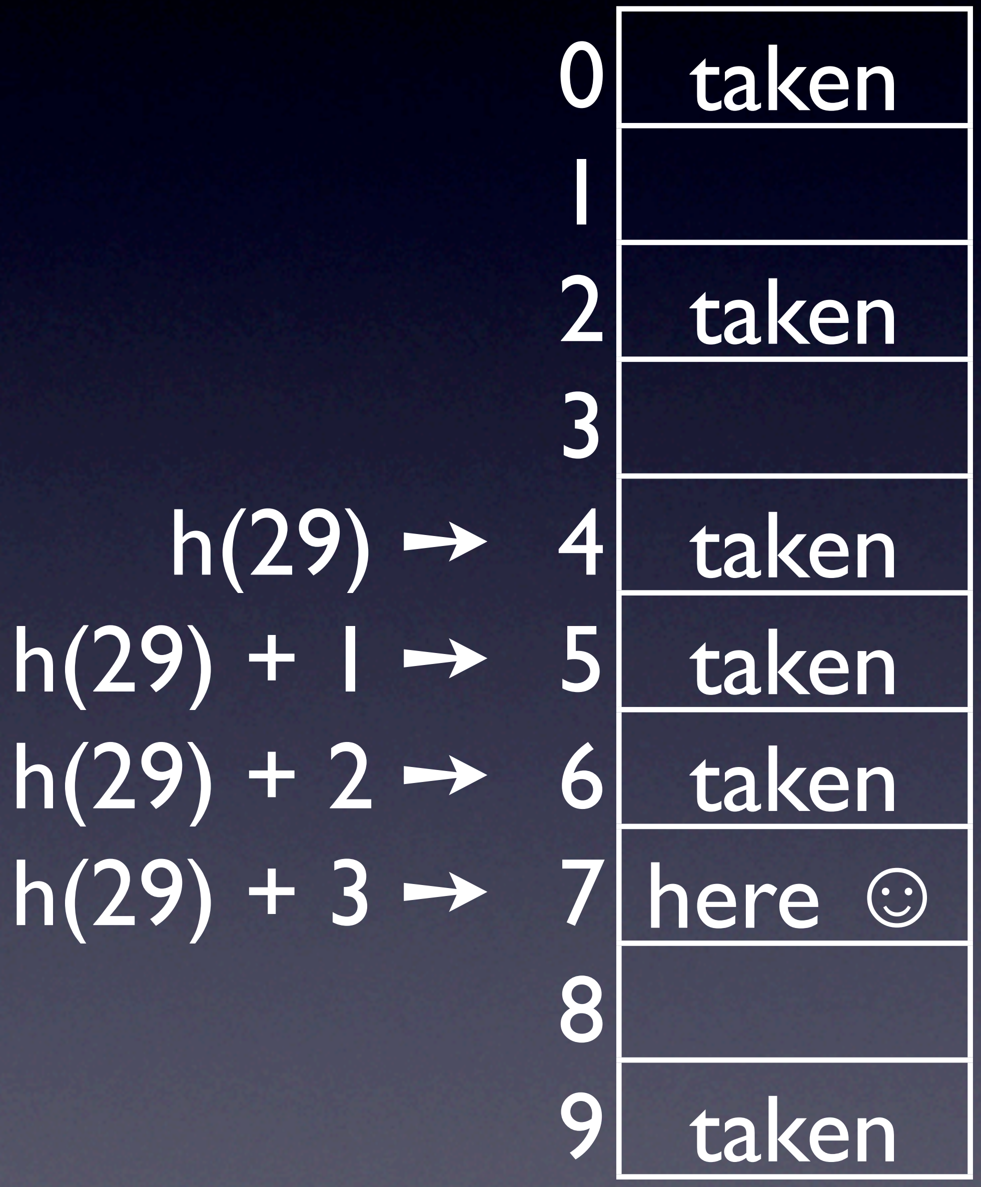
- Open addressing
- Karp-Rabin
  - coming back from the dead to hunt us

Open Addressing

- Goal: use nothing but the table
    - Hoping for less code, better caching
- Hashing ⇒ we must handle collisions
    - Solution: try another location

Easy Collision handling

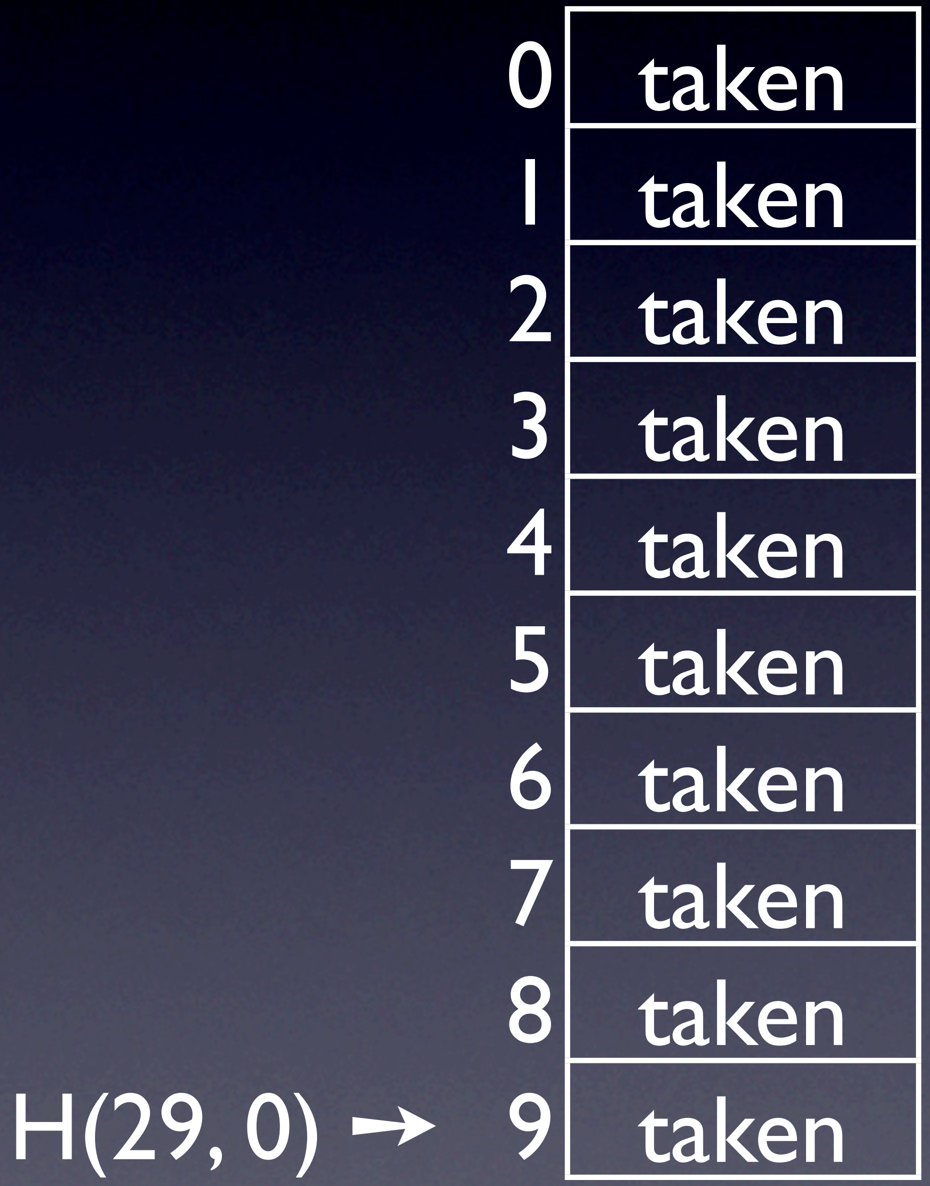
- h(x) = standard hash function
- if T[h(x)] is taken
  - try T[h(x)+1]
  - then T[h(x) + 2]
  - then T[h(x) + 3]
- just like parking a car



	0	taken
	1	
	2	taken
	3	
h(29) →	4	taken
h(29) + 1 →	5	taken
h(29) + 2 →	6	taken
h(29) + 3 →	7	here ☺
	8	
	9	taken

Collision Handling: Abstracting it Up

- h(k) grows up to H(k, i) where i is the attempt number
- first try T[H(k, 0)]

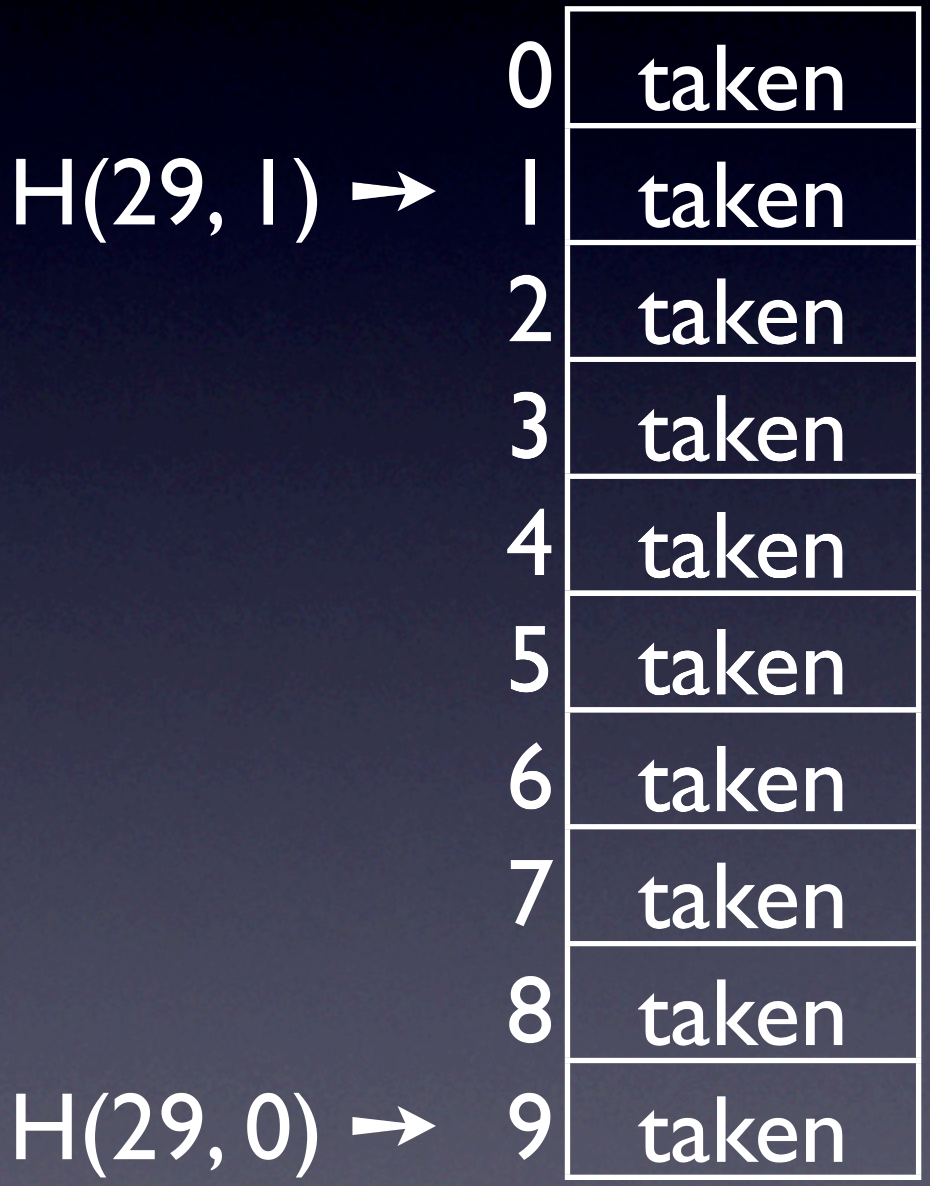


0	taken
1	taken
2	taken
3	taken
4	taken
5	taken
6	taken
7	taken
8	taken
9	taken

H(29, 0) → 9

Collision Handling: Abstracting it Up

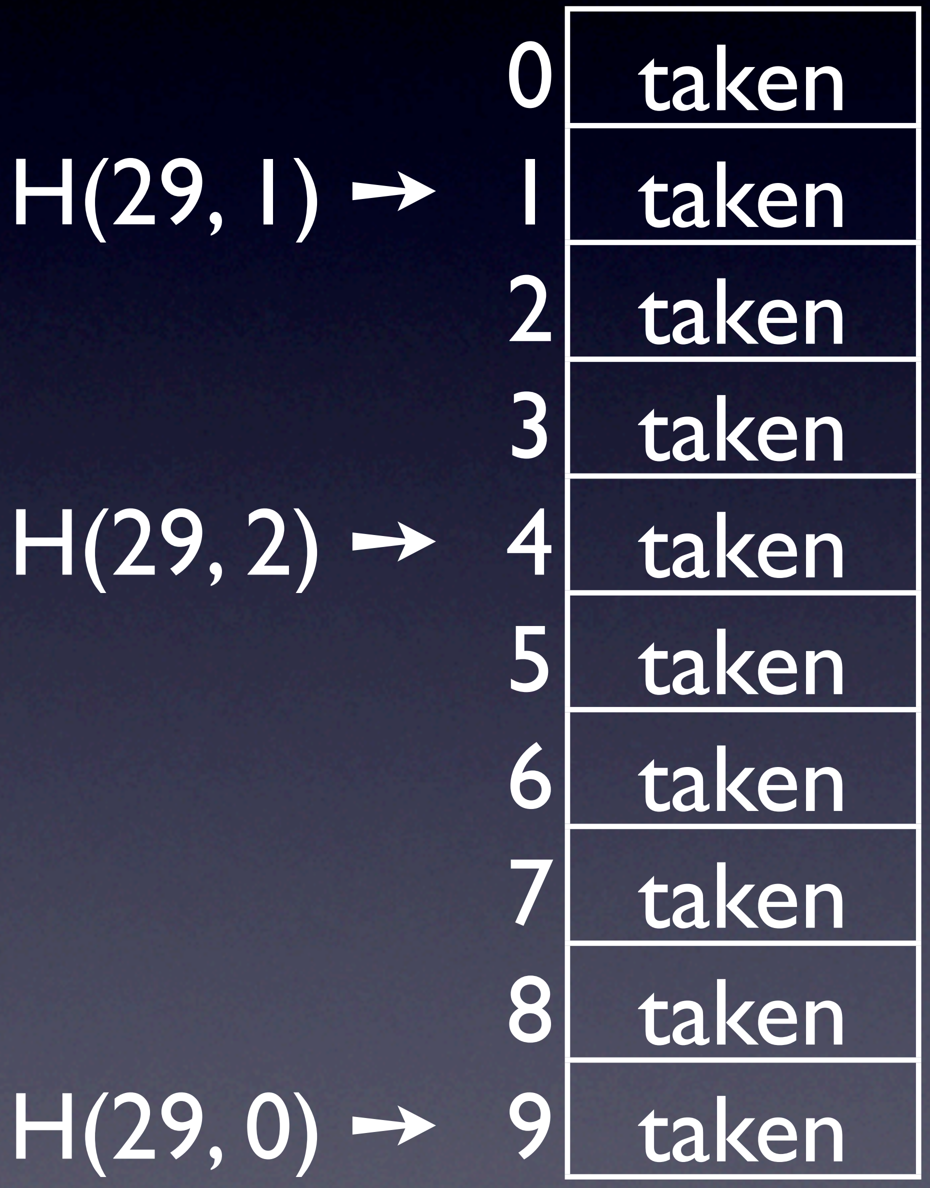
- h(k) grows up to H(k, i) where i is the attempt number
- first try T[H(k, 0)]
  - then T[H(k, 1)]



	0	taken
H(29, 1) →	1	taken
	2	taken
	3	taken
	4	taken
	5	taken
	6	taken
	7	taken
	8	taken
H(29, 0) →	9	taken

Collision Handling: Abstracting it Up

- h(k) grows up to H(k, i) where i is the attempt number
- first try T[H(k, 0)]
  - then T[H(k, 1)]
  - then T[H(k, 2)]



	0	taken
H(29, 1) →	1	taken
	2	taken
	3	taken
H(29, 2) →	4	taken
	5	taken
	6	taken
	7	taken
	8	taken
H(29, 0) →	9	taken

Collision Handling: Abstracting it Up

- h(k) grows up to H(k, i) where i is the attempt number
- first try T[H(k, 0)]
  - then T[H(k, 1)]
  - then T[H(k, 2)]
- stop after trying all

H(29, 3) →	0	taken
H(29, 1) →	1	taken
H(29, 4) →	2	taken
H(29, 9) →	3	taken
H(29, 2) →	4	taken
H(29, 5) →	5	taken
H(29, 6) →	6	taken
H(29, 7) →	7	taken
H(29, 8) →	8	taken
H(29, 0) →	9	taken

Collision Handling: Abstracting it Up

- H(k) = <H(k, 0), H(k, 1), H(k, 2) ... >
- Linear probing, h(29) = 4, H_{linear}(29) = ?

<4, 5, 6, 7, 8, 9, 0, 1, 2, 3>

- General properties?

H(29, 3) →	0	taken
H(29, 1) →	1	taken
H(29, 4) →	2	taken
H(29, 9) →	3	taken
H(29, 2) →	4	taken
H(29, 5) →	5	taken
H(29, 6) →	6	taken
H(29, 7) →	7	taken
H(29, 8) →	8	taken
H(29, 0) →	9	taken

Collision Handling: Abstracting it Up

- Any collision handling strategy comes to:
    - for key k, probe H(k,0), then H(k,1) etc.
- No point in trying the same place twice
- Probes should cover the whole table (otherwise we raise 'table full' prematurely)
- Conclusion: H(k, 0), H(k, 1) ... H(k, m-1) are a permutation of {1, 2, 3 ... m}

Linear Probing and Permutations

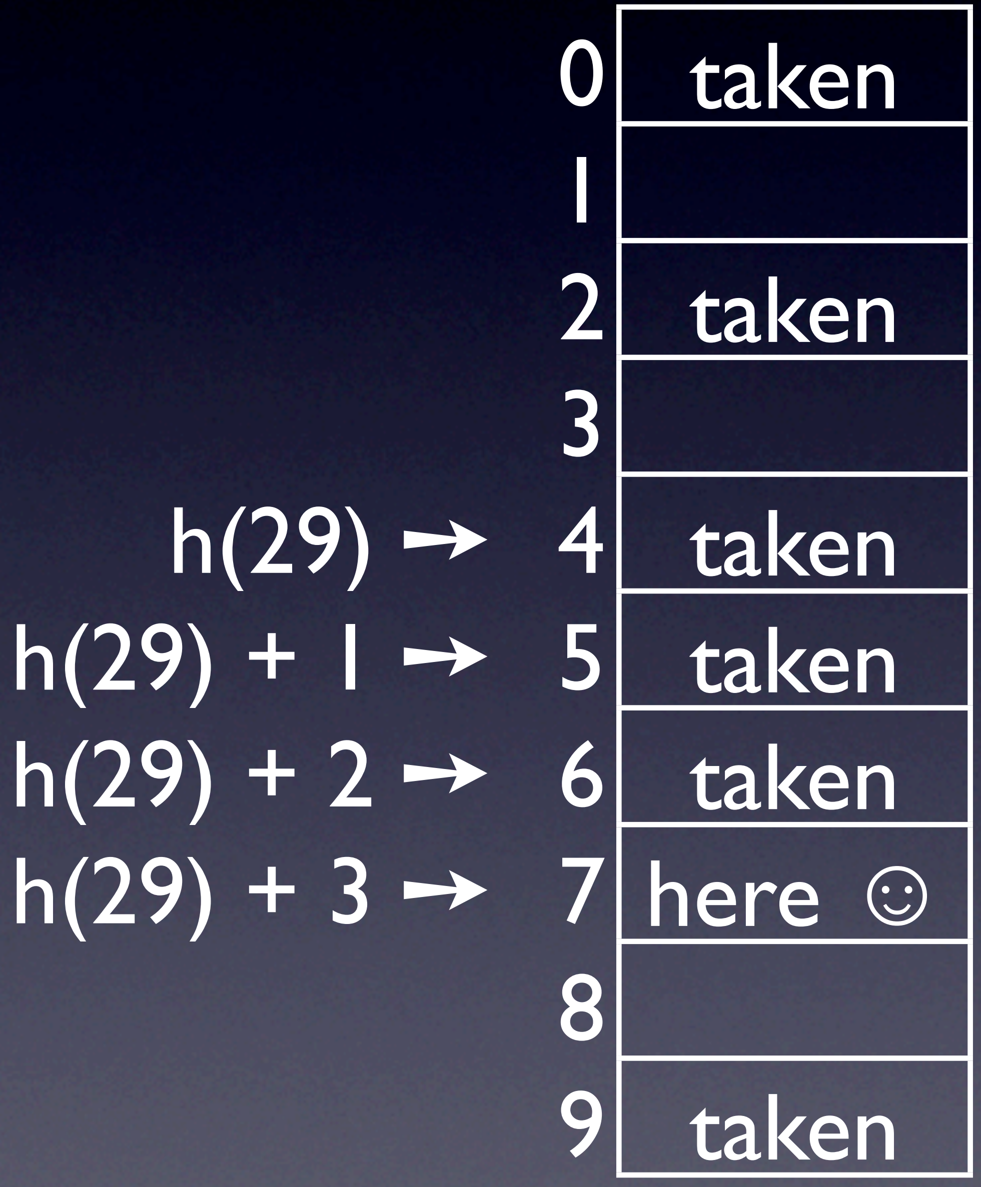
- h(29) = 4; H(29) =

  <4, 5, 6, 7, 8, 9, 0, 1, 2, 3>

- h(k) = h_0(mod m); H(k) =

  <h_0 mod m, (h_0 + 1) mod m, (h_0 + 2) mod m, ... (h_0 + m - 1) mod m >

- m permutations (max m!)



	0	taken
	1	
	2	taken
	3	
h(29) →	4	taken
h(29) + 1 →	5	taken
h(29) + 2 →	6	taken
h(29) + 3 →	7	here ☺
	8	
	9	taken

Ideal Collision Handling

- Simple Hashing (collision by chaining)
    - Ideal hashing function: uniformly distributes keys across hash values
- Open Addressing
    - Ideal hashing function: uniformly distributes keys across permutations
    - a.k.a. uniform hashing

Uniform Hashing: Achievable?

- Simple mapping between permutations of m and numbers 1 ... m!
- Convert key to big number, then use permutation number (bignum mod m!)
- ... right?

k mod 6	Permutation
0	<1, 2, 3>
1	<1, 3, 2>
2	<2, 1, 3>
3	<2, 3, 1>
4	<3, 1, 2>
5	<3, 2, 1>

Uniform Hashing: Achievable?

- Number of digits in m!
    - O(log(m!))
    - O(m*log(m))
- Working mod m! is slow
    - check your Python cost model

k mod 6	Permutation
0	<1, 2, 3>
1	<1, 3, 2>
2	<2, 1, 3>
3	<2, 3, 1>
4	<3, 1, 2>
5	<3, 2, 1>

Working Compromise

- Why does linear probing suck?
  - We jump in the table once, then walk
- Improvement
  - Keep jumping after the initial jump
  - Jumping distance: 2nd hash function
  - Name: double hashing

Double Hashing: Math

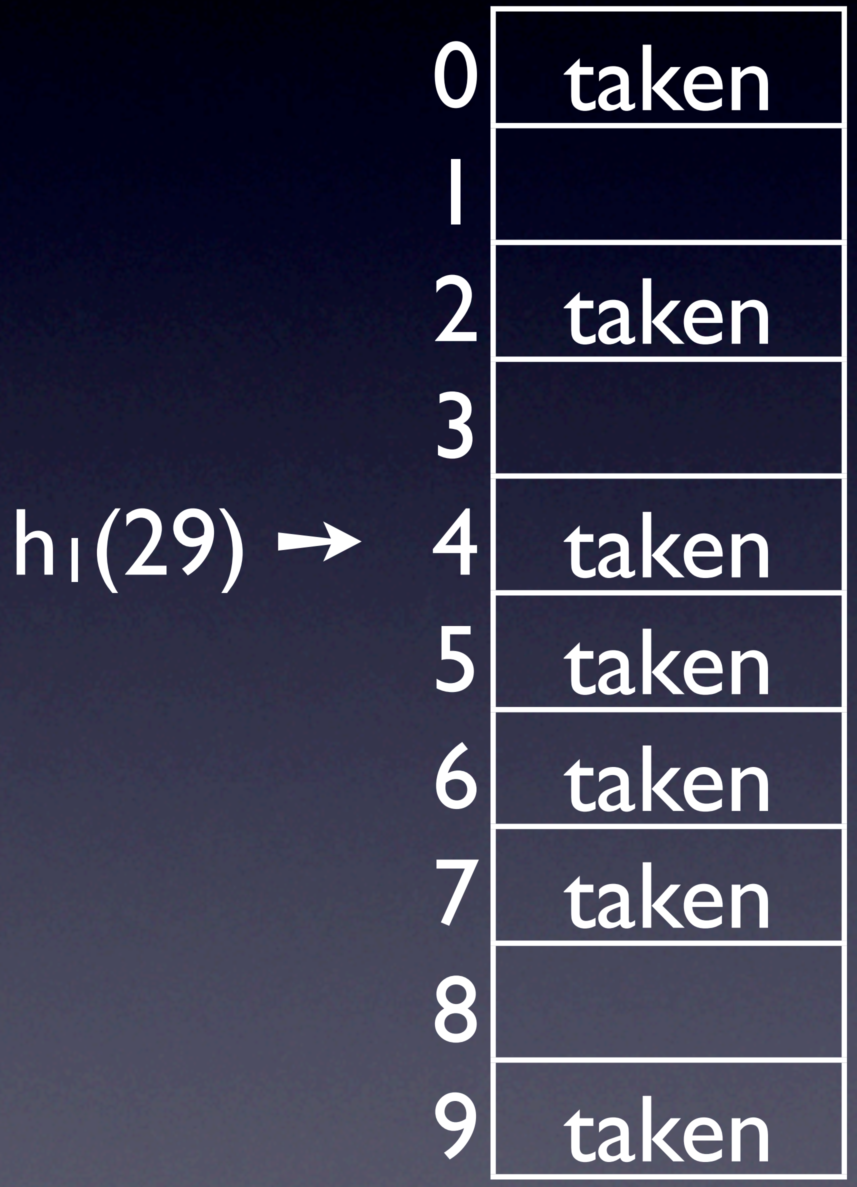
- $h_1(k)$ and $h_2(k)$ are hashing functions

0	taken
1	
2	taken
3	
4	taken
5	taken
6	taken
7	taken
8	
9	taken

Double Hashing: Math

- $h_1(k)$ and $h_2(k)$ are hashing functions

- $H(k, 0) = h_1(k)$

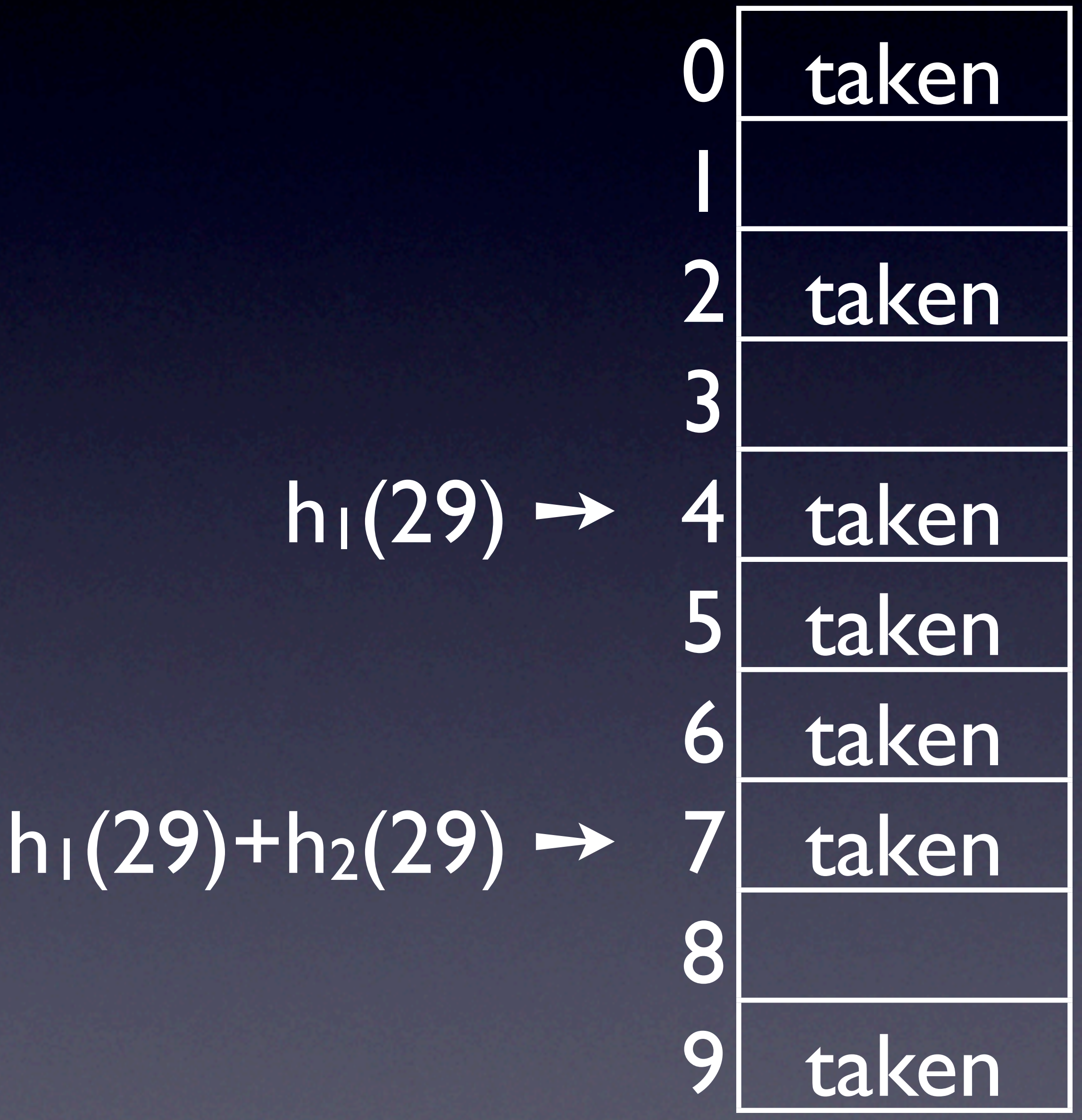


$h_1(29)$ → 4

0	taken
1	
2	taken
3	
4	taken
5	taken
6	taken
7	taken
8	
9	taken

Double Hashing: Math

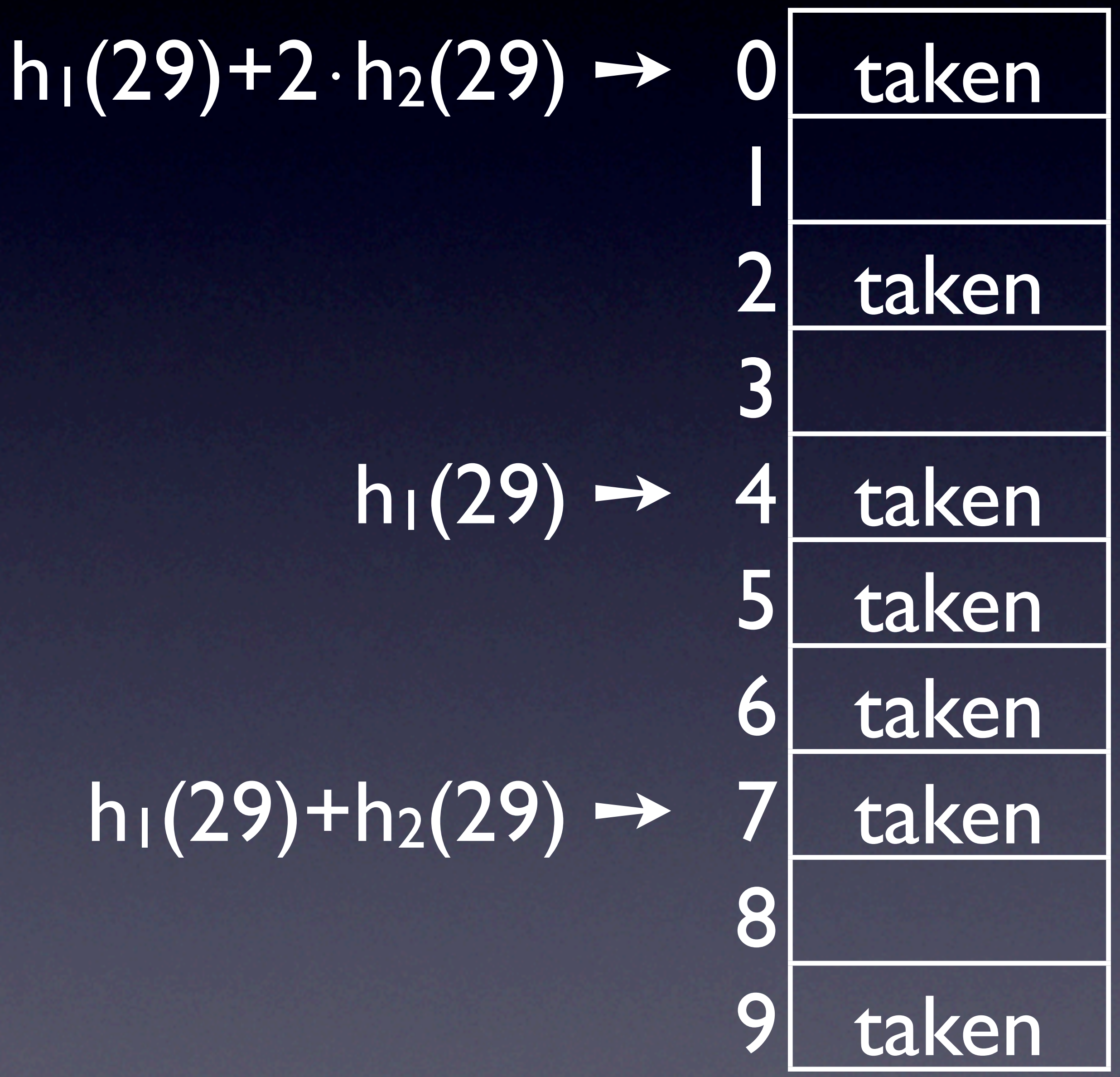
- $h_1(k)$ and $h_2(k)$ are hashing functions
- $H(k, 0) = h_1(k)$
- $H(k, 1) = h_1(k) + h_2(k)$



	0	taken
	1	
	2	taken
	3	
$h_1(29)$ →	4	taken
	5	taken
	6	taken
$h_1(29)+h_2(29)$ →	7	taken
	8	
	9	taken

Double Hashing: Math

- $h_1(k)$ and $h_2(k)$ are hashing functions
- $H(k, 0) = h_1(k)$
- $H(k, 1) = h_1(k) + h_2(k)$



$h_1(29)+2 \cdot h_2(29)$ → 0

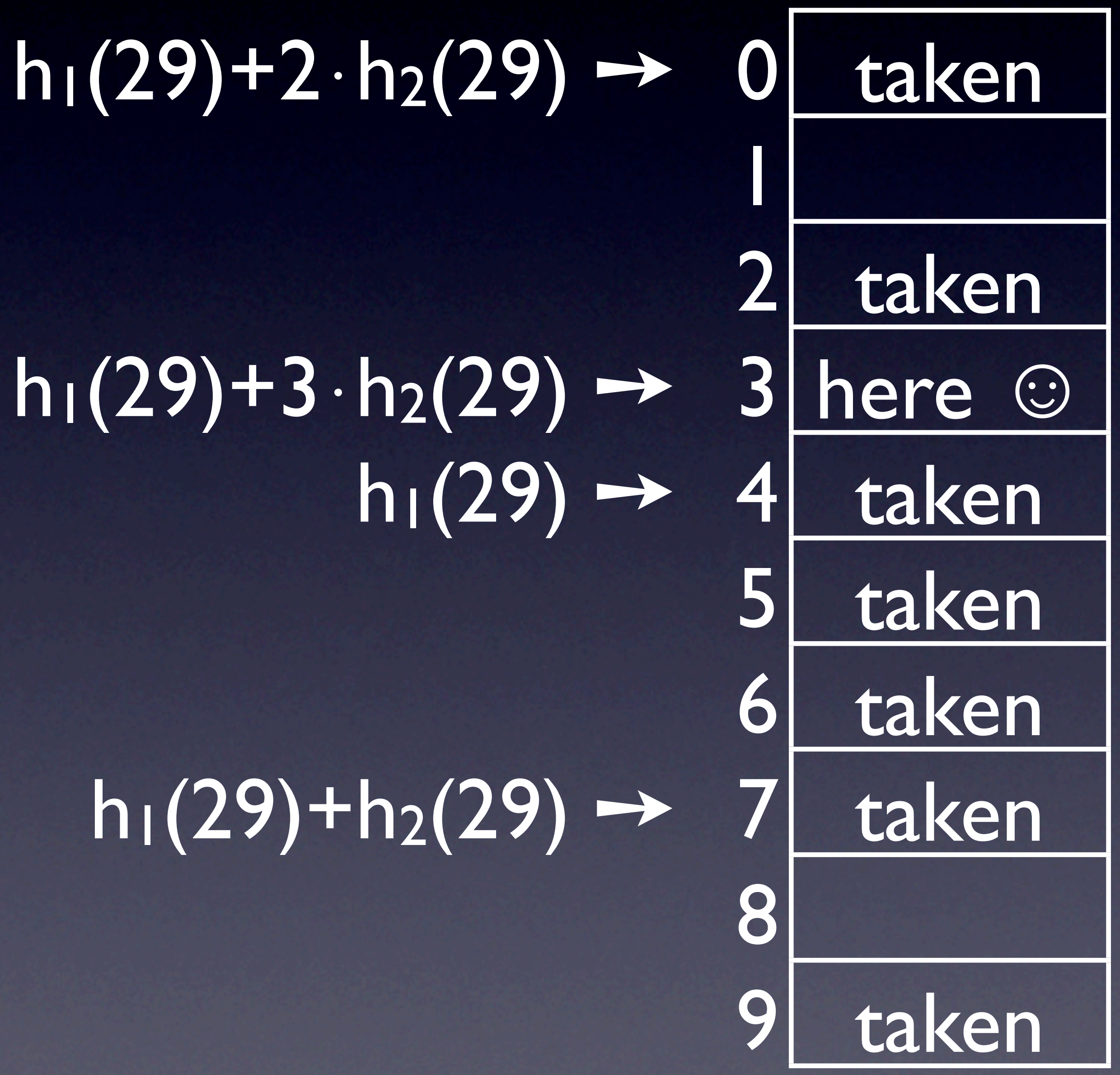
$h_1(29)$ → 4

$h_1(29)+h_2(29)$ → 7

0	taken
1	
2	taken
3	
4	taken
5	taken
6	taken
7	taken
8	
9	taken

Double Hashing: Math

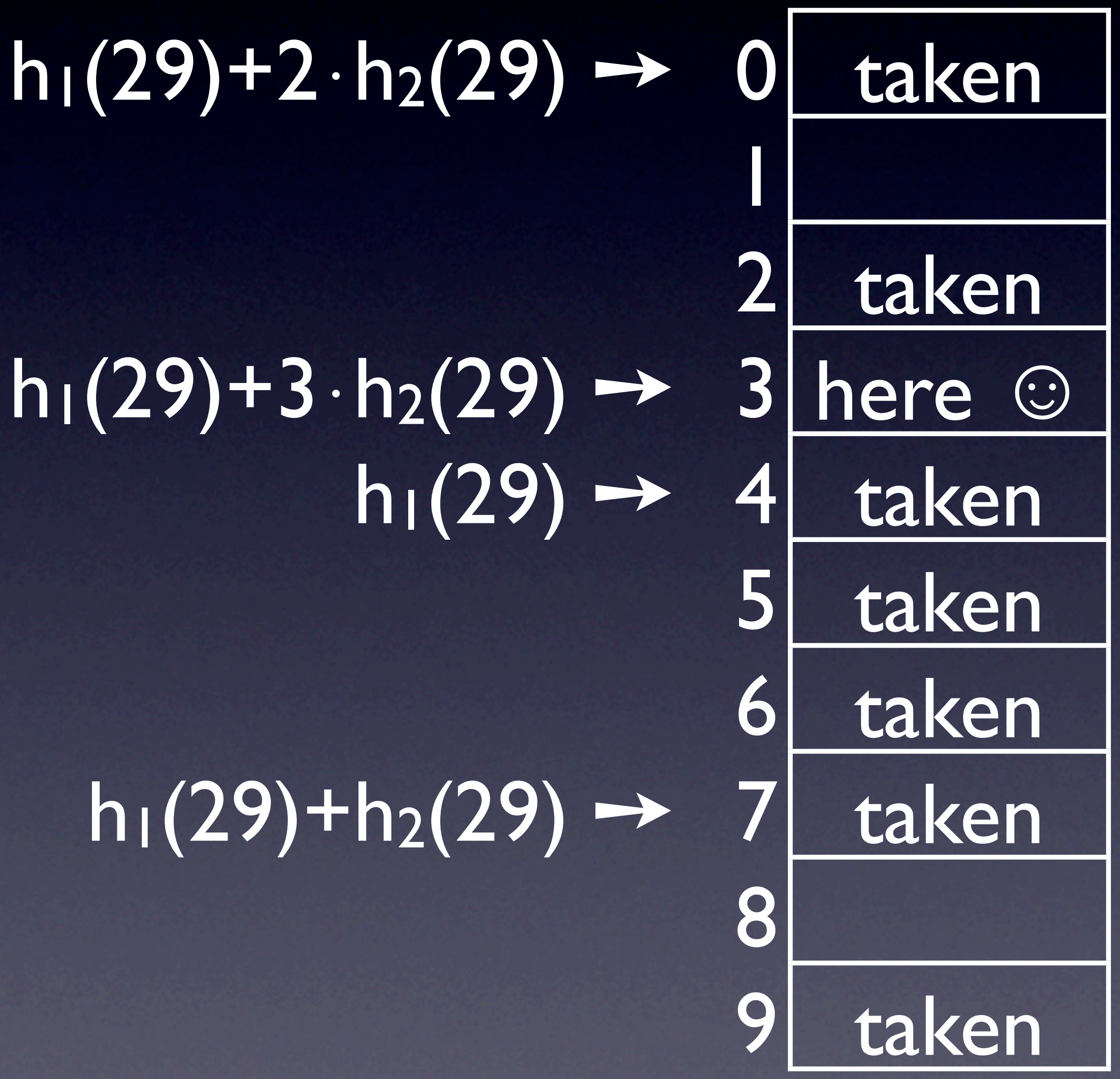
- $h_1(k)$ and $h_2(k)$ are hashing functions
- $H(k, 0) = h_1(k)$
- $H(k, 1) = h_1(k) + h_2(k)$



$h_1(29)+2\cdot h_2(29)$	→	0	taken
		1	
		2	taken
$h_1(29)+3\cdot h_2(29)$	→	3	here ☺
$h_1(29)$	→	4	taken
		5	taken
		6	taken
$h_1(29)+h_2(29)$	→	7	taken
		8	
		9	taken

Double Hashing: Math

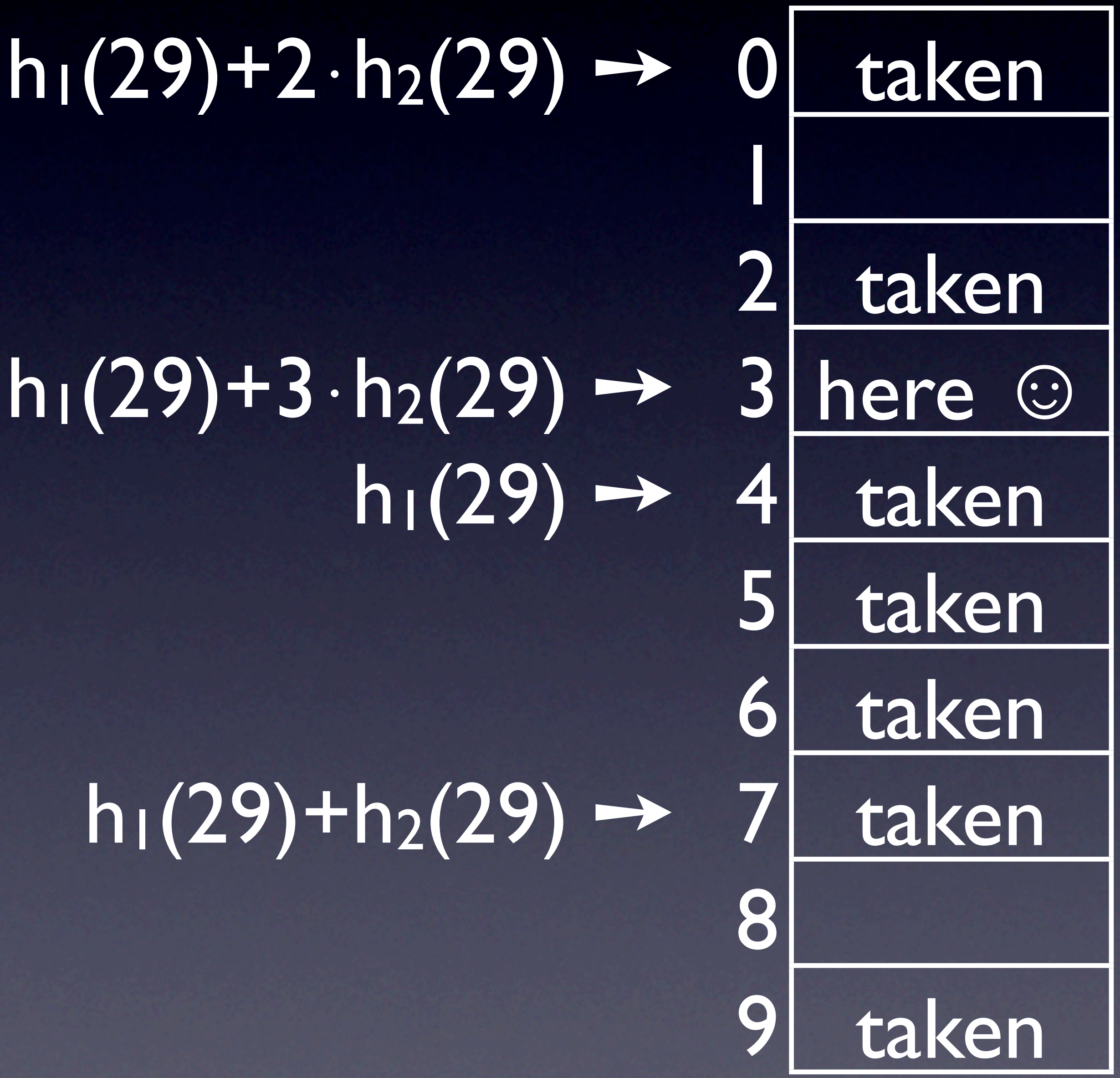
- $h_1(k)$ and $h_2(k)$ are hashing functions
- $H(k, 0) = h_1(k)$
- $H(k, 1) = h_1(k) + h_2(k)$
- $H(k, i) = h_1(k) + i \cdot h_2(k)$
  - mod m
  - you knew that, right?



$h_1(29)+2 \cdot h_2(29)$ →	0	taken
	1	
	2	taken
$h_1(29)+3 \cdot h_2(29)$ →	3	here ☺
$h_1(29)$ →	4	taken
	5	taken
	6	taken
$h_1(29)+h_2(29)$ →	7	taken
	8	
	9	taken

Double Hashing Trap

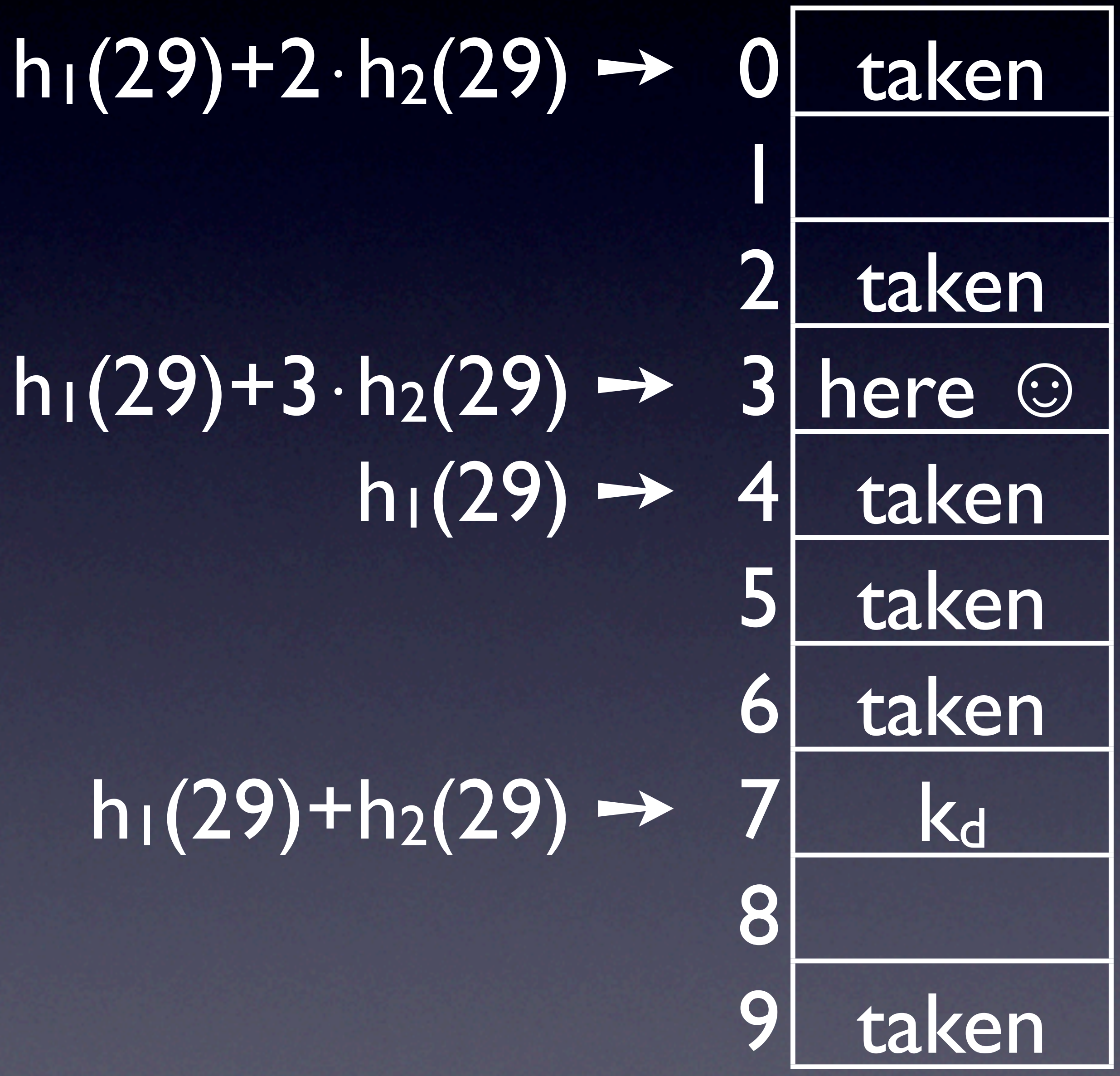
- gcd($h_2(k)$, m) must be 1
- solution 1 (easy to get)
    - m prime, $h_2(k) = k \bmod q$ (where $q < m$)
- solution 2 (faster, better)
    - $m = 2^r$ (table can grow)
    - $h_2(k)$ is odd (not even)



$h_1(29)+2 \cdot h_2(29)$ →	0	taken
	1	
	2	taken
$h_1(29)+3 \cdot h_2(29)$ →	3	here ☺
$h_1(29)$ →	4	taken
	5	taken
	6	taken
$h_1(29)+h_2(29)$ →	7	taken
	8	
	9	taken

Open Addressing: Deleting Keys

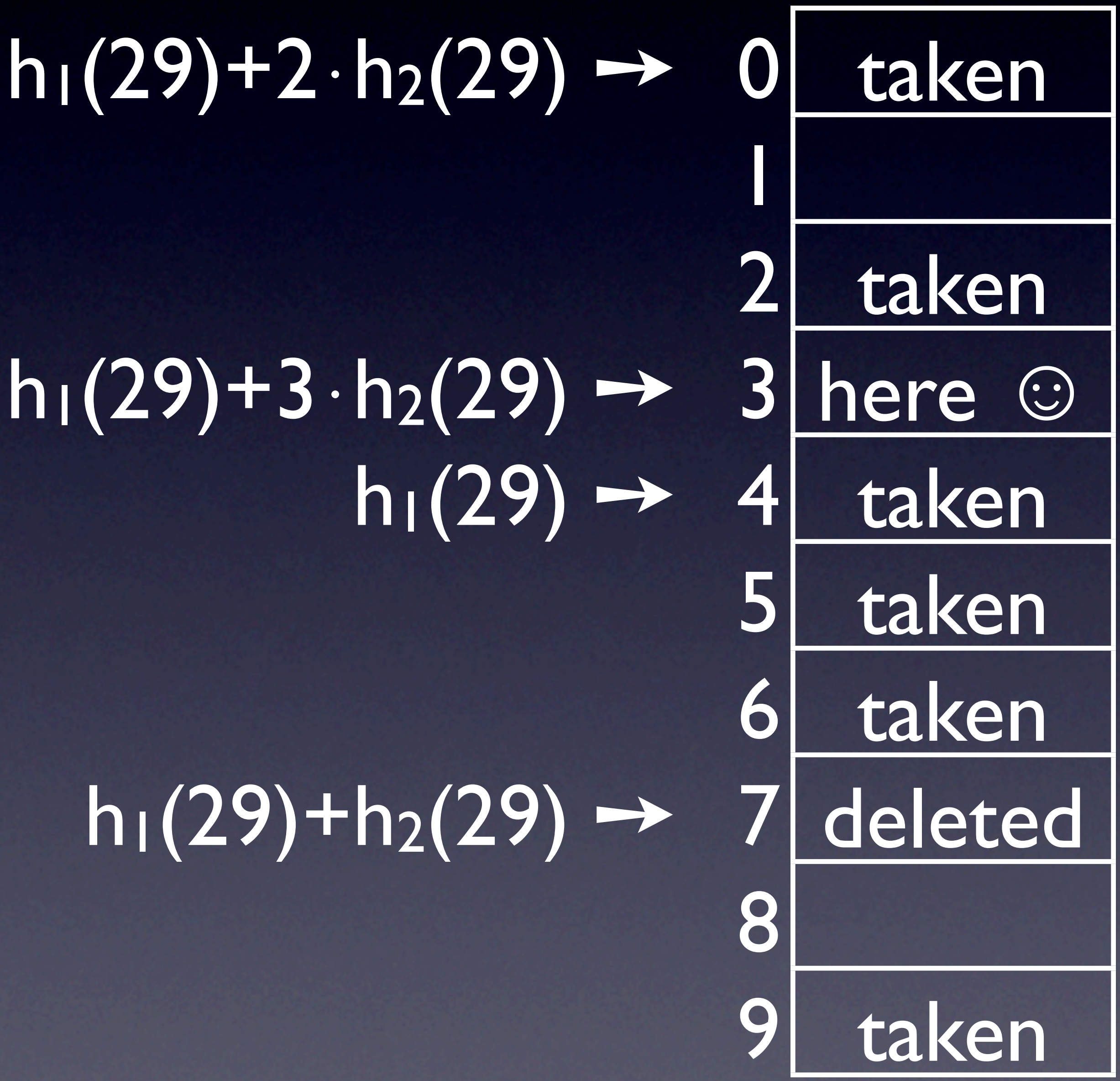
- Suppose we want to delete k_d stored at 7
- Can't simply wipe the entry, because key 29 wouldn't be found anymore
    - rember H(29) = <4, 7, 0, 3 ...>



$h_1(29)+2 \cdot h_2(29)$ →	0	taken
	1	
	2	taken
$h_1(29)+3 \cdot h_2(29)$ →	3	here ☺
$h_1(29)$ →	4	taken
	5	taken
	6	taken
$h_1(29)+h_2(29)$ →	7	k_d
	8	
	9	taken

Open Addressing: Deleting Keys

- Entry meaning ‘deleted’
- Handling ‘deleted’
  - Search: Keep looking
  - Insert: Stop, replace ‘deleted’ with the new key/value



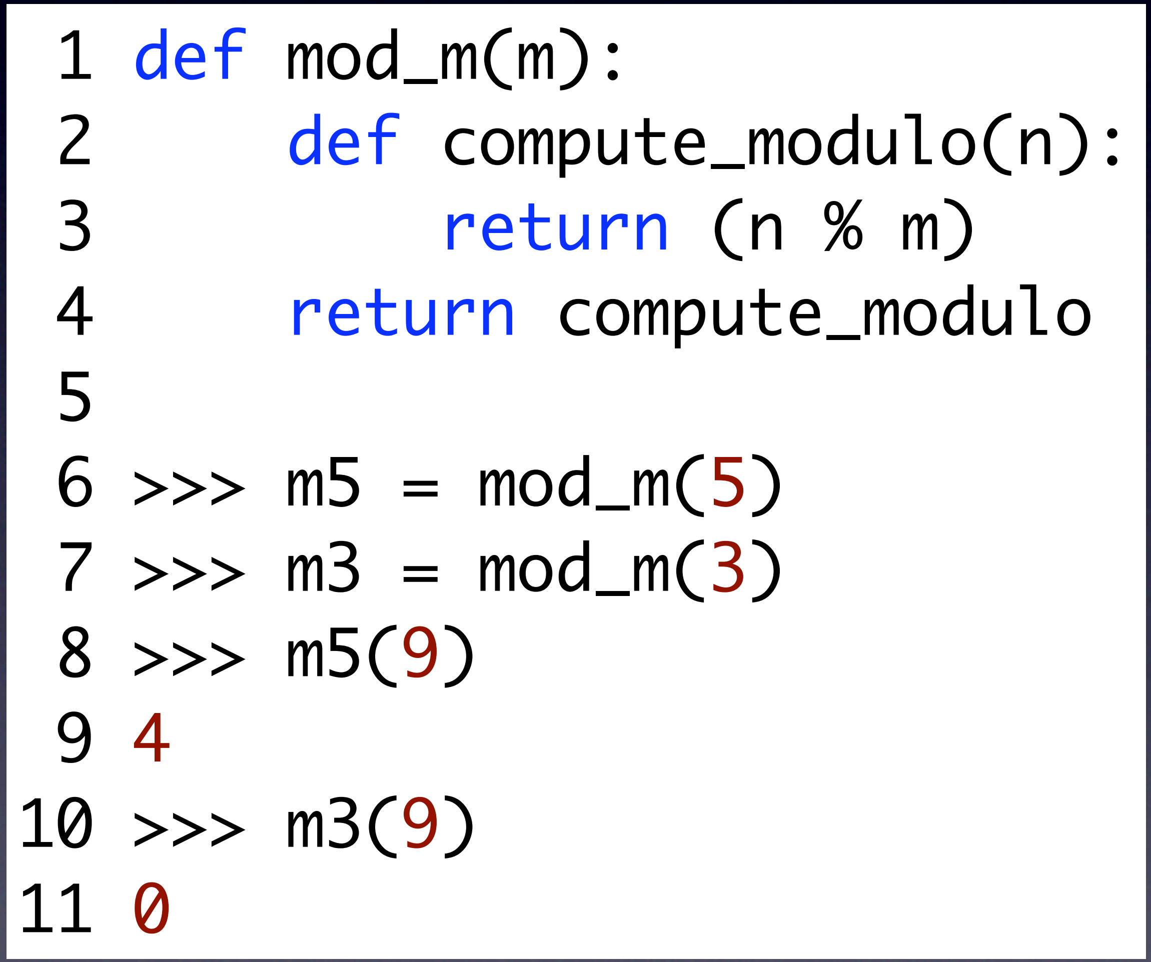
$h_1(29)+2 \cdot h_2(29)$ →	0	taken
	1	
	2	taken
$h_1(29)+3 \cdot h_2(29)$ →	3	here ☺
$h_1(29)$ →	4	taken
	5	taken
	6	taken
$h_1(29)+h_2(29)$ →	7	deleted
	8	
	9	taken

Open Addressing:Code

- Design: implementing a collection in Python
  - **__getitem__**(self, key)
    - return key item or raise KeyError(key)
  - **__setitem__**(self, key, item)
    - insert / replace (key, item)
  - **__delitem__**(self, key)

Open Addressing: Code

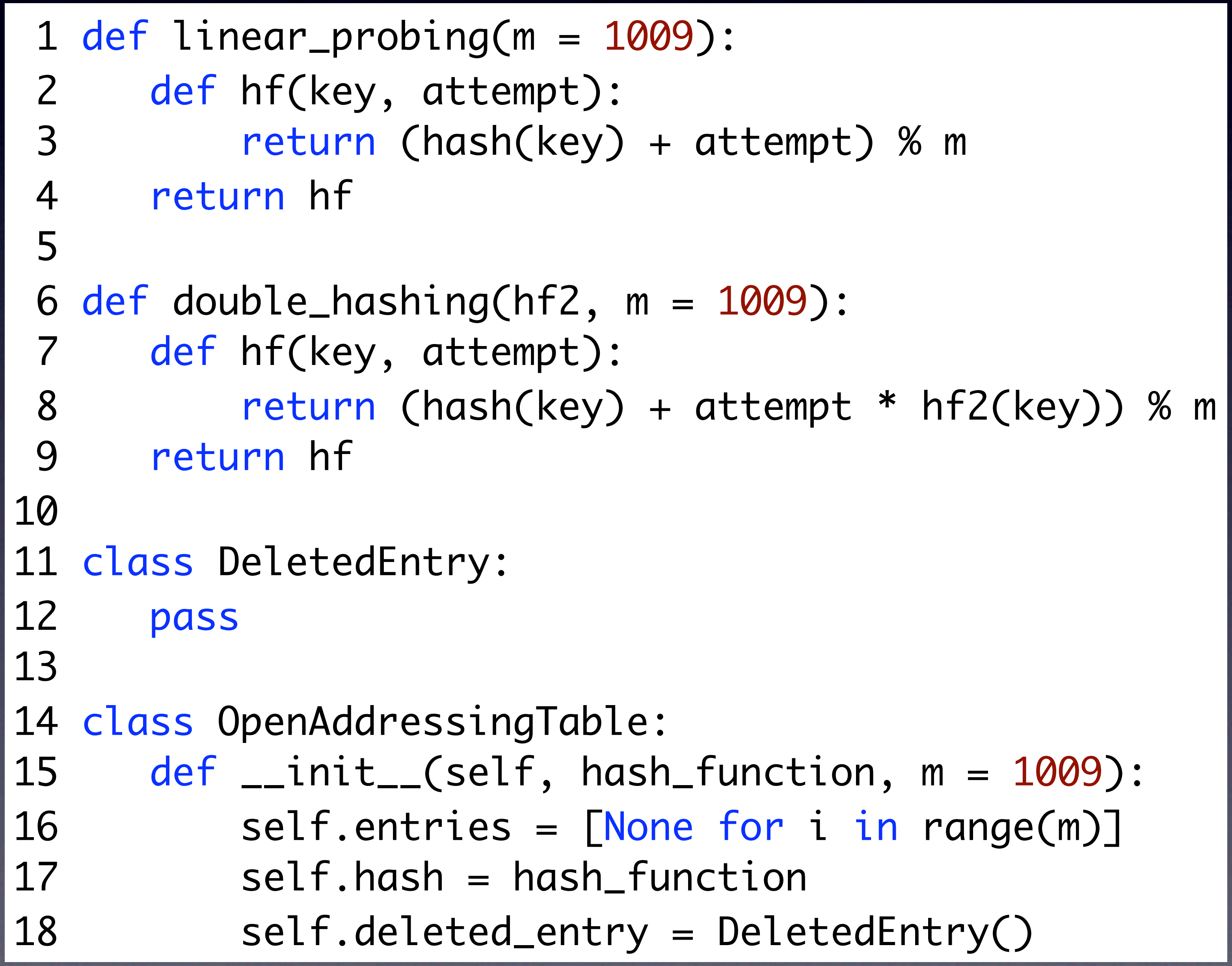
- Closures: not for n00bs
- `def compute_modulo` is local to the `mod_m` call
- the function created by `def compute_modulo` is returned like any object
- the object remembers the context around the def (the value of m)



```
def mod_m(m):
    def compute_modulo(n):
        return (n % m)
    return compute_modulo

>>> m5 = mod_m(5)
>>> m3 = mod_m(3)
>>> m5(9)
4
>>> m3(9)
0
```

Open Addressing:Code



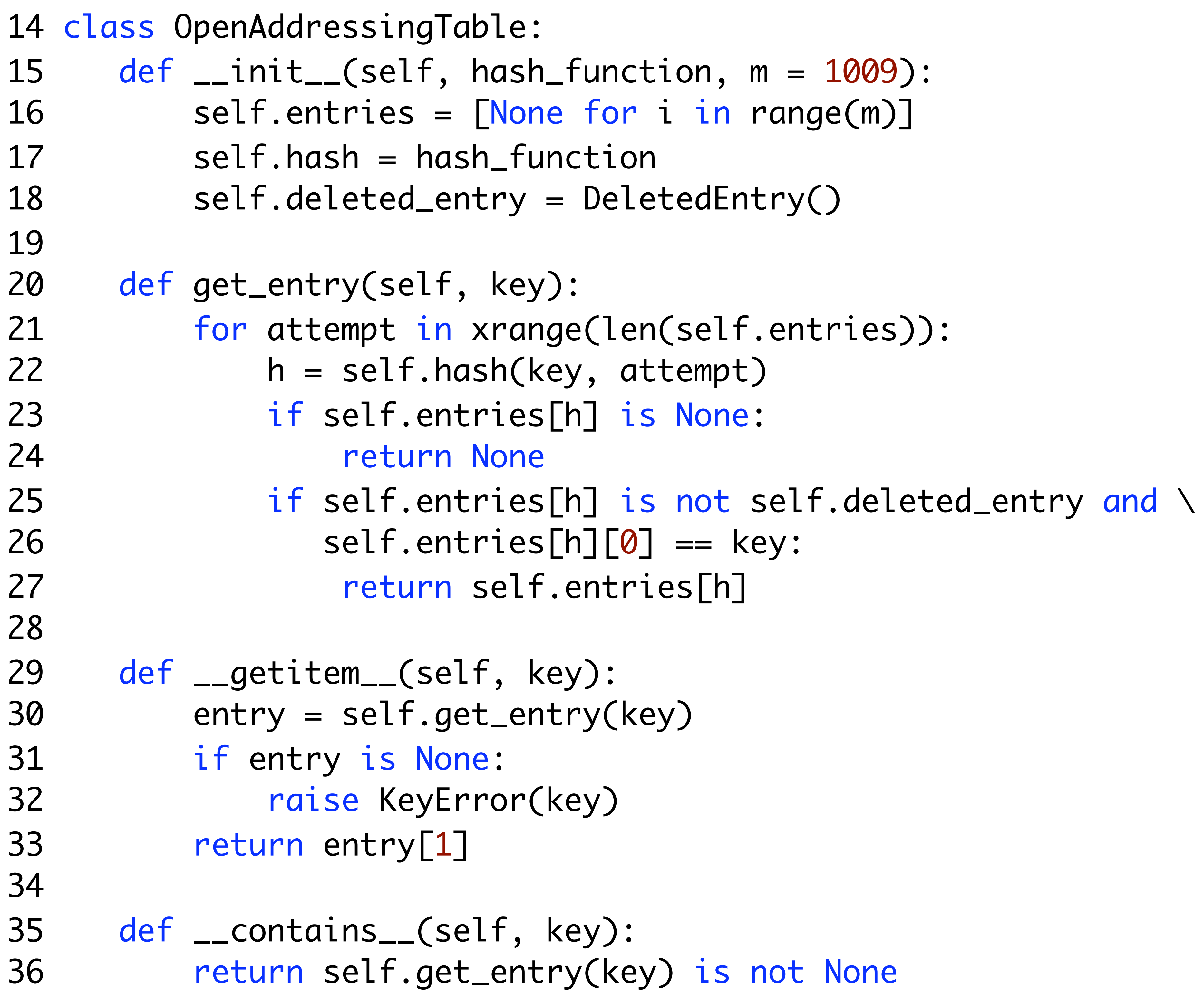
```python
def linear_probing(m = 1009):
    def hf(key, attempt):
        return (hash(key) + attempt) % m
    return hf

def double_hashing(hf2, m = 1009):
    def hf(key, attempt):
        return (hash(key) + attempt * hf2(key)) % m
    return hf

class DeletedEntry:
    pass

class OpenAddressingTable:
    def __init__(self, hash_function, m = 1009):
        self.entries = [None for i in range(m)]
        self.hash = hash_function
        self.deleted_entry = DeletedEntry()
```

Open Addressing: Code



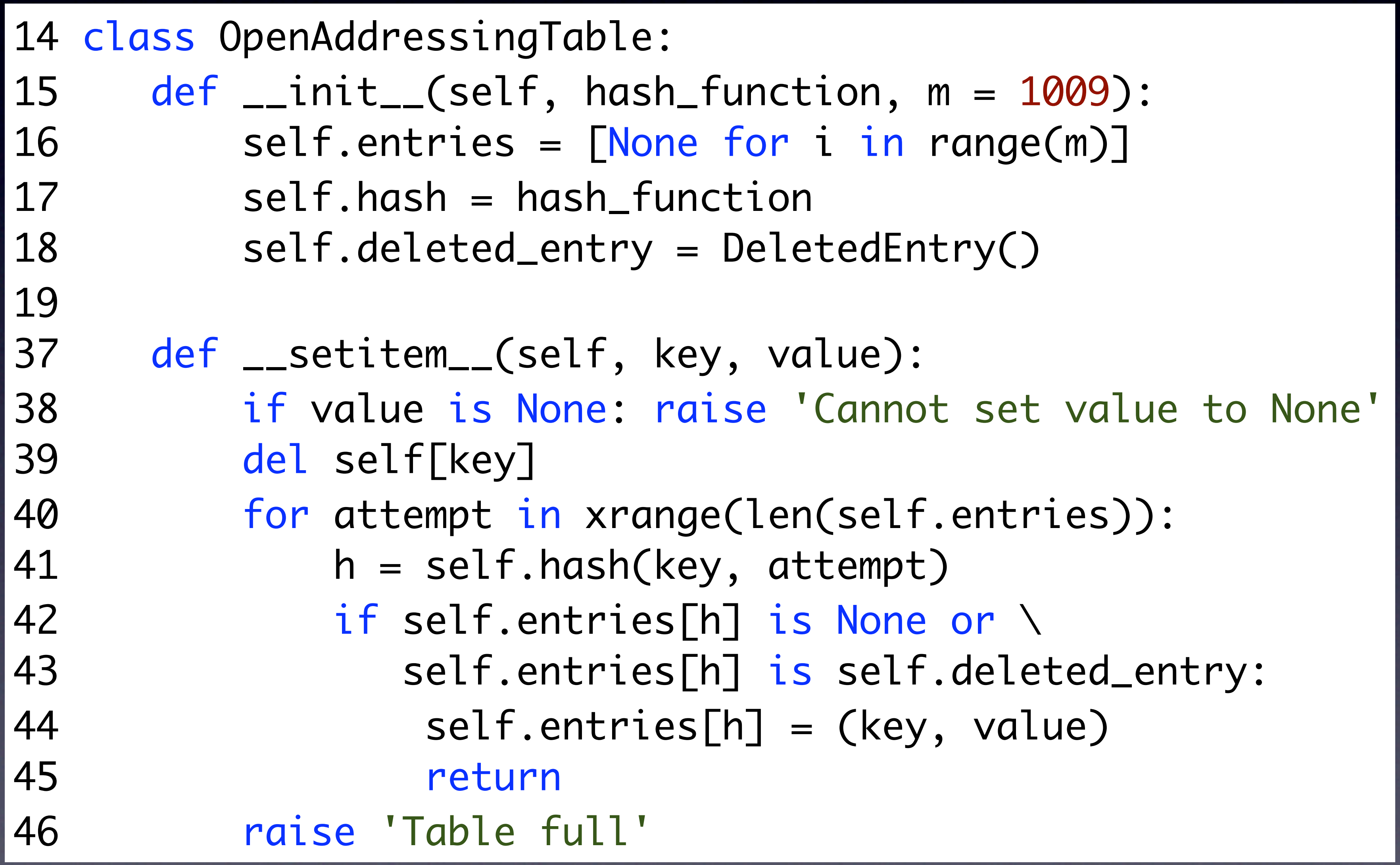
```
class OpenAddressingTable:
    def __init__(self, hash_function, m = 1009):
        self.entries = [None for i in range(m)]
        self.hash = hash_function
        self.deleted_entry = DeletedEntry()

    def get_entry(self, key):
        for attempt in xrange(len(self.entries)):
            h = self.hash(key, attempt)
            if self.entries[h] is None:
                return None
            if self.entries[h] is not self.deleted_entry and \
               self.entries[h][0] == key:
                return self.entries[h]

    def __getitem__(self, key):
        entry = self.get_entry(key)
        if entry is None:
            raise KeyError(key)
        return entry[1]

    def __contains__(self, key):
        return self.get_entry(key) is not None
```

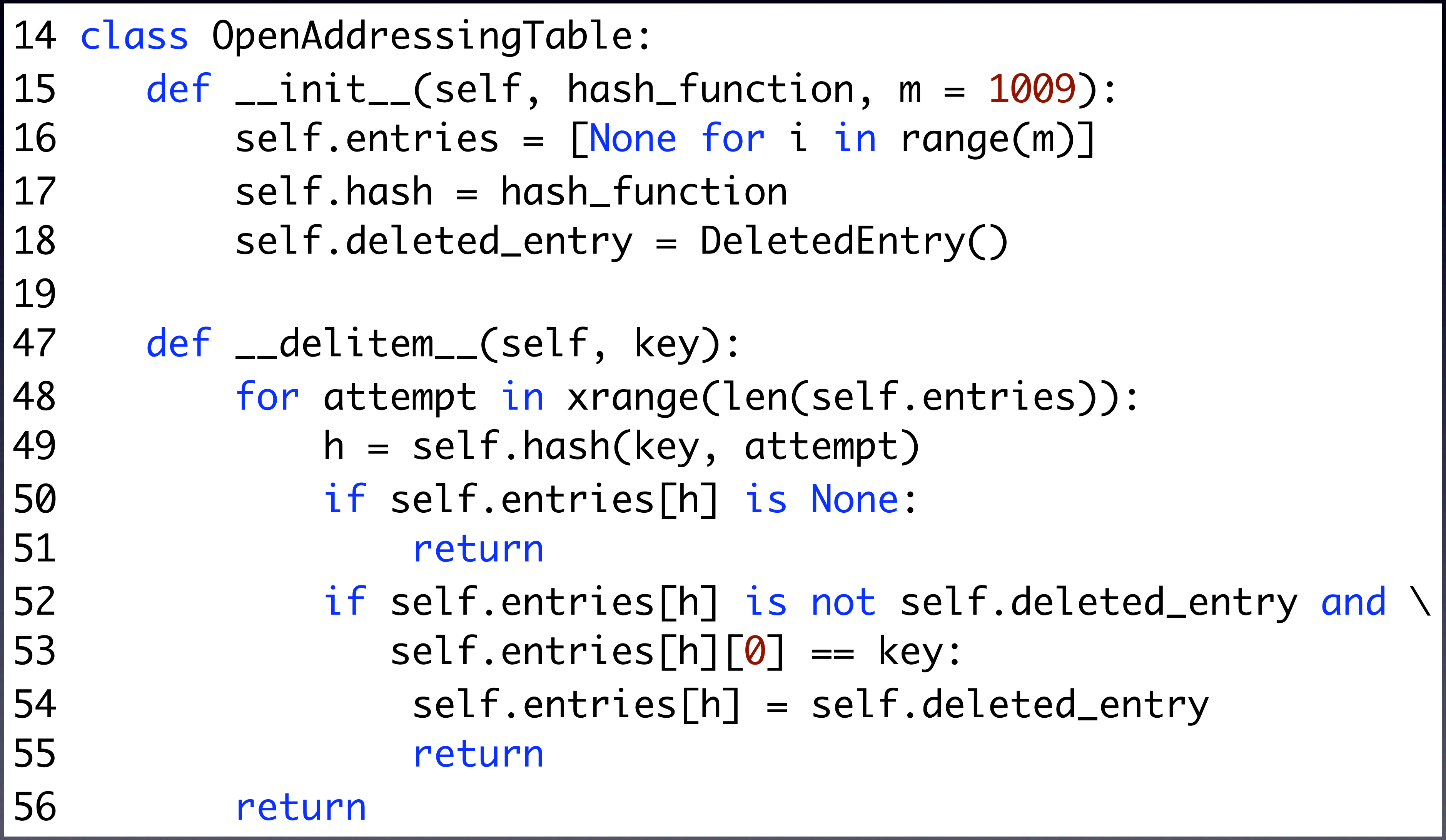
Open Addressing: Code



```python
class OpenAddressingTable:
   def __init__(self, hash_function, m = 1009):
       self.entries = [None for i in range(m)]
       self.hash = hash_function
       self.deleted_entry = DeletedEntry()

   def __setitem__(self, key, value):
       if value is None: raise 'Cannot set value to None'
       del self[key]
       for attempt in xrange(len(self.entries)):
           h = self.hash(key, attempt)
           if self.entries[h] is None or \
              self.entries[h] is self.deleted_entry:
               self.entries[h] = (key, value)
               return
       raise 'Table full'
```

Open Addressing: Code



```
class OpenAddressingTable:
   def __init__(self, hash_function, m = 1009):
       self.entries = [None for i in range(m)]
       self.hash = hash_function
       self.deleted_entry = DeletedEntry()

   def __delitem__(self, key):
       for attempt in xrange(len(self.entries)):
           h = self.hash(key, attempt)
           if self.entries[h] is None:
               return
           if self.entries[h] is not self.deleted_entry and \
              self.entries[h][0] == key:
               self.entries[h] = self.deleted_entry
               return
       return
```

Ghosts of Karp & Rabin

Getting Rolling Hashes Right

Modular Arithmetic

- Foundation:
  - $(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
- From that, it follows that:
  - $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
    - induction: multiplication is repeated +

Modular Gotcha

- Never give mod a negative number
    - want q = (a - b) mod m, but a - b < 0
    - q mod m = (a - (b mod m)) mod m
    - but (b mod m) is < m
    - so (a + m - (b mod m)) > 0
    - q = (a + m - (b mod m)) mod m

Modular Arithmetic-Fu

- Multiplicative inverses: assume p is prime
- For every a and p, there is a^{-1} so that:
  - $(a * a^{-1}) \bmod p = 1$
  - example: p = 23, a = 8 ⇒ $a^{-1} = 3$
    - check: 8 * 23 = 24, 24 mod 23 = 1
- Multiplying by a^{-1} is like dividing by a

Modular Arithmetic-Fu

- How do we compute a^{-1}?
- Fermat's Little Theorem:
    - p prime ⇒ $a^{a-1} \bmod p = 1$
- Huh?
    - $a^{a-1} \bmod p = a * a^{a-2} \bmod p = 1$
    - so (for p) $a^{-1} \bmod p = a^{a-2} \bmod p$

Back to Rolling Hashes

- Data Structure (just like hash table)
  - start with empty list
  - append(val): appends val at the end of list
  - skip(): removes the first list element
  - hash(): computes a hash of the list

And we're done!

- costan@mit.edu
- (617) 230-9694, no voicemail
- AIM: victorcostan
- Google Talk: costan@gmail.com
- 32G-8th Floor

v. Next

- Open Addressing takes 40/45 minutes
- Maybe enough time to cover modular arithmetic, not enough time for rolling hash tricks
- Can improve structure, more visualizations will definitely help students get it faster