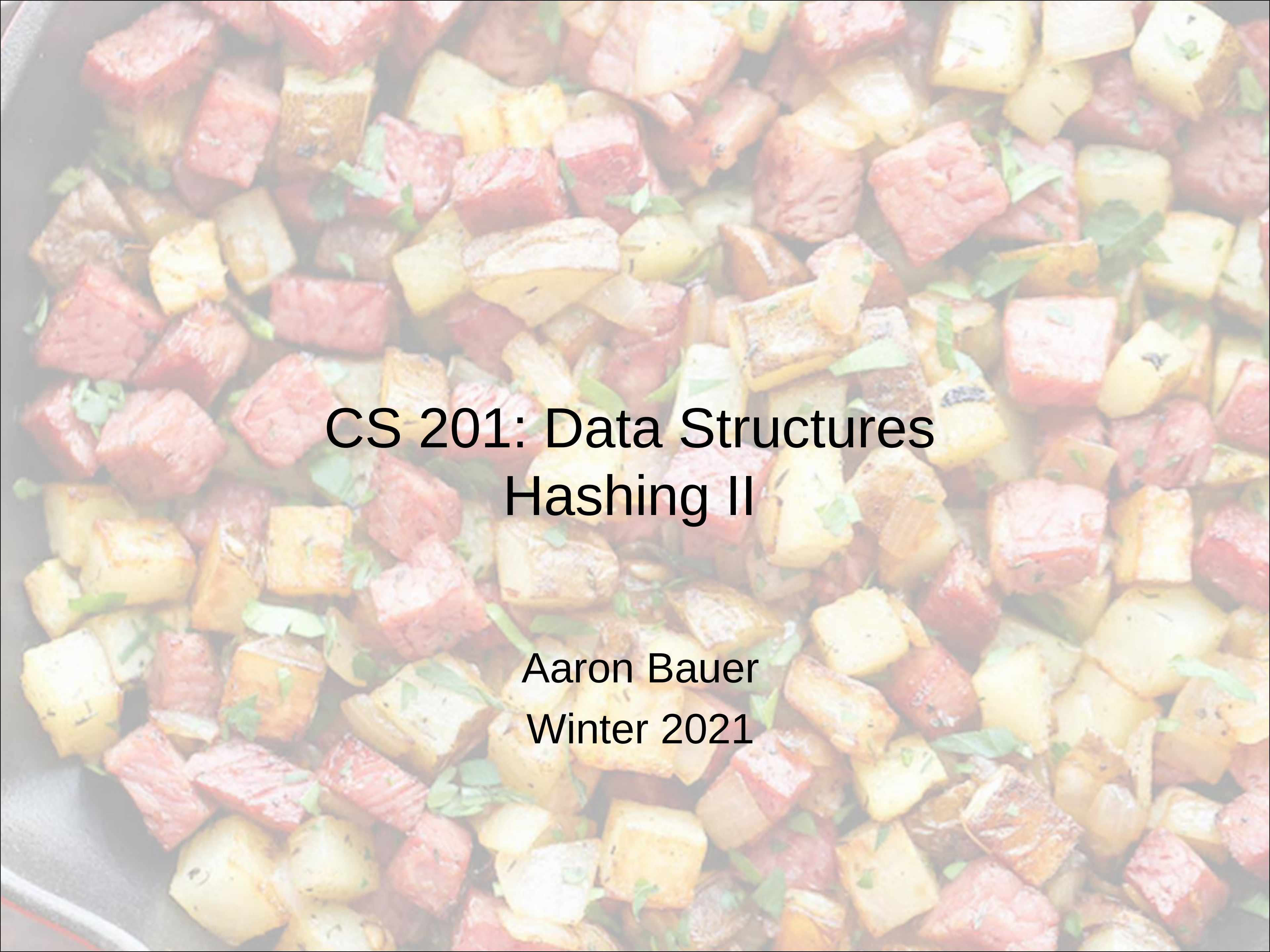


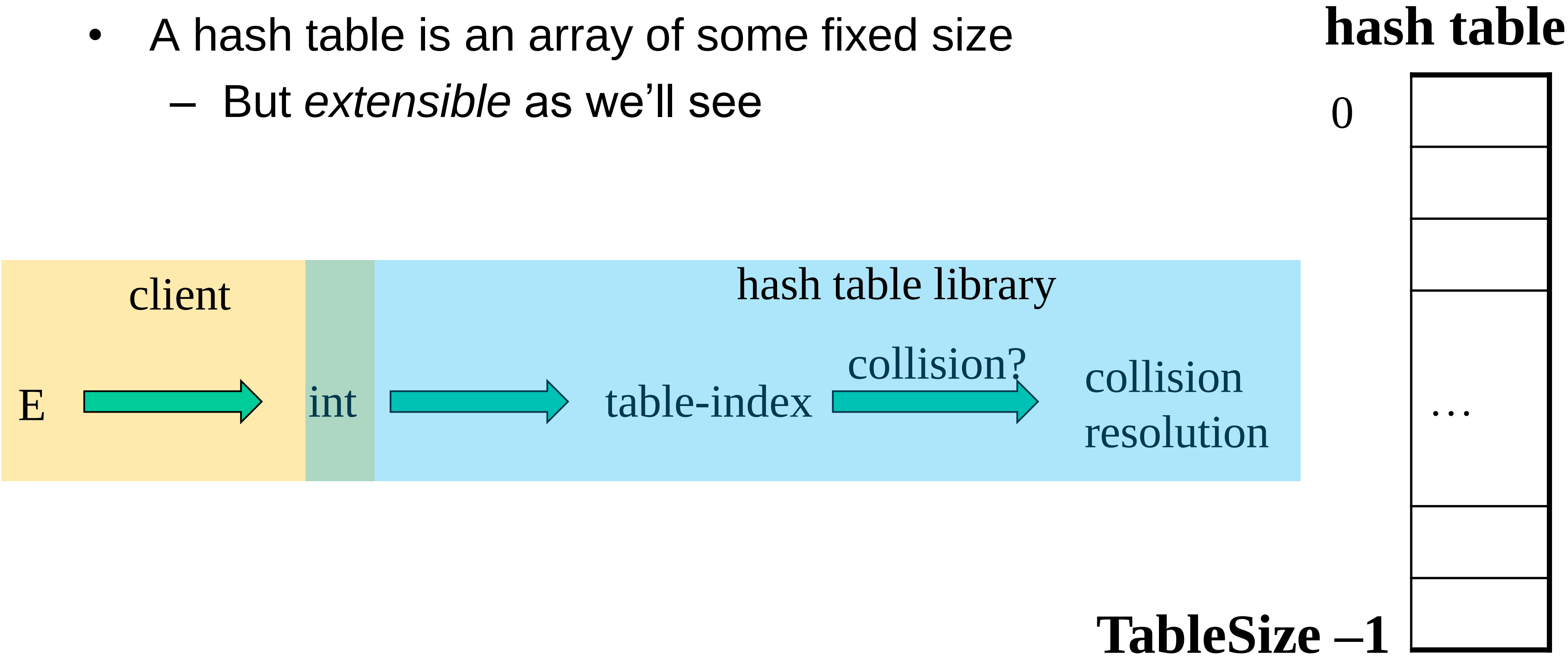
CS 201: Data Structures
Hashing II

Aaron Bauer

Winter 2021

Hash Tables: Review

- Aim for constant-time **`put`**, **`contains`**, and **`remove`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size
  - But *extensible* as we'll see



Collision resolution

Collision:

When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution

Collision-avoidance

- With “`x % TableSize`” the number of collisions depends on
  - the ints inserted (obviously)
  - `TableSize`
- Larger table-size tends to help, but not always
  - Example: 70, 24, 56, 43, 10
    with `TableSize` = 10 and `TableSize` = 60
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - “Multiples of 61” are probably less likely than “multiples of 60”

More on prime table size

If **`TableSize`** is 60 and…

- Lots of keys hash to multiples of 5, wasting 80% of table
- Lots of keys hash to multiples of 10, wasting 90% of table
- Lots of keys hash to multiples of 2, wasting 50% of table

If **`TableSize`** is 61…

- Collisions can still happen, but 5, 10, 15, 20, … will fill table
- Collisions can still happen but 10, 20, 30, 40, … will fill table
- Collisions can still happen but 2, 4, 6, 8, … will fill table

This “table-filling” property happens whenever the multiple and the table-size have a *greatest-common-divisor* of 1

Handling Collisions: Separate Chaining

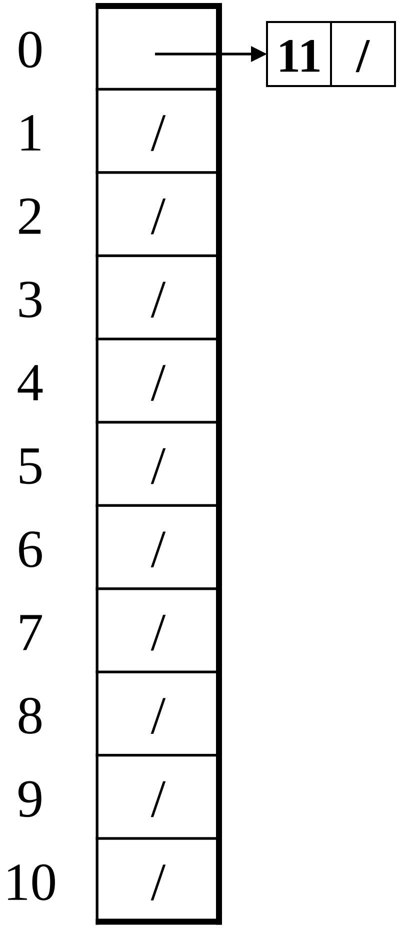
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/
10	/

Chaining:
All keys that map to the same table location are kept in a linked list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

Handling Collisions: Separate Chaining



Chaining:

All keys that map to the same

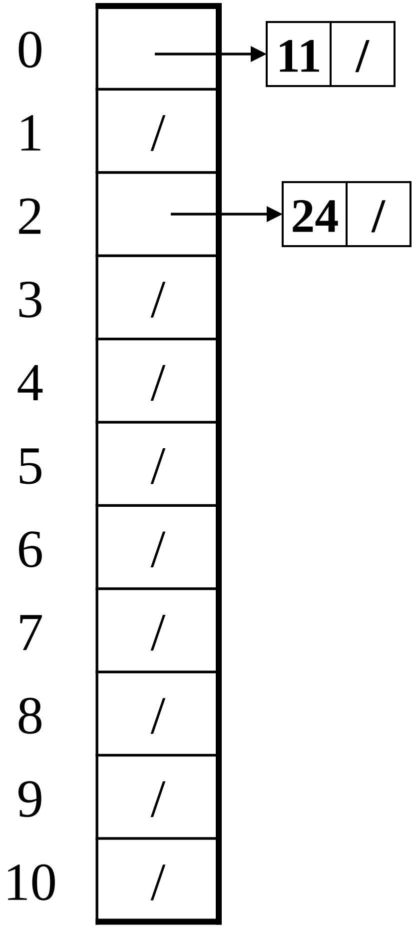
table location are kept in a linked list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

Handling Collisions: Separate Chaining



Chaining:

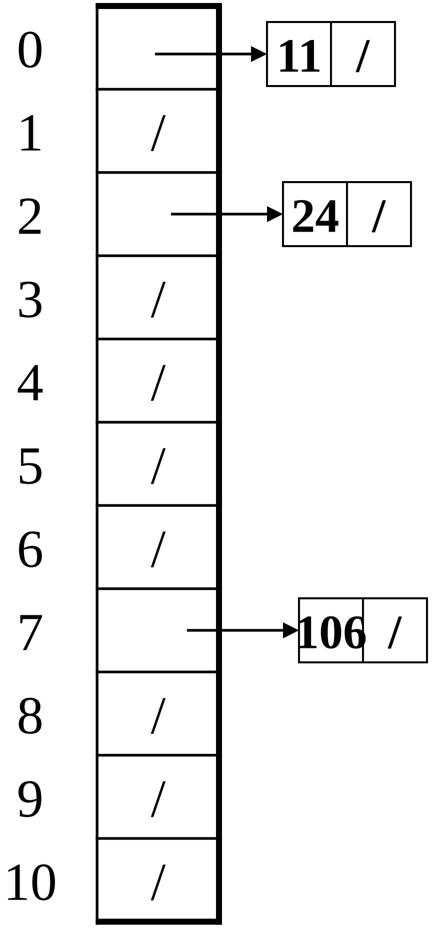
All keys that map to the same table location are kept in a linked list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

Handling Collisions: Separate Chaining



Chaining:

All keys that map to the same

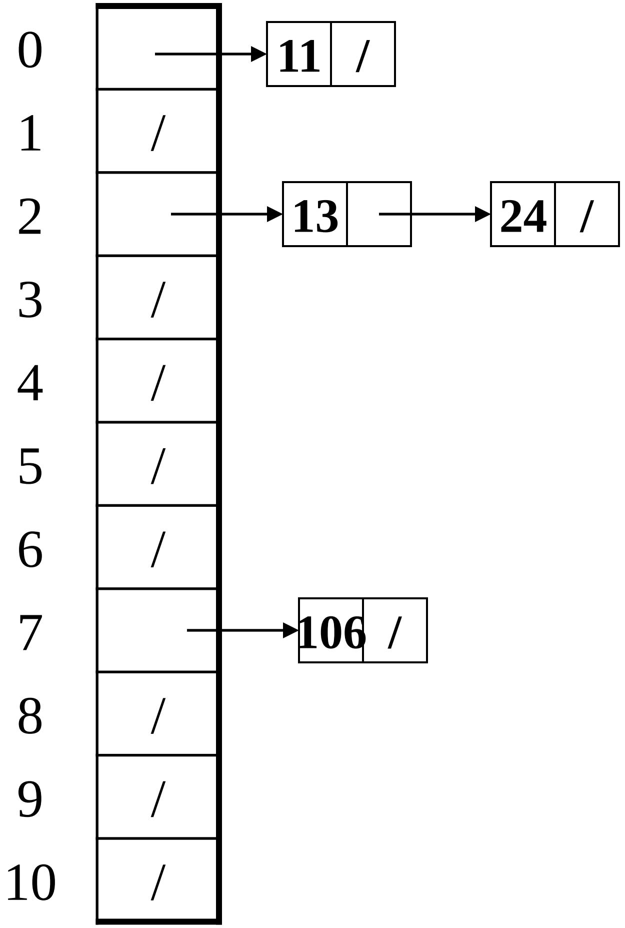
table location are kept in a linked list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

Handling Collisions: Separate Chaining



Chaining:

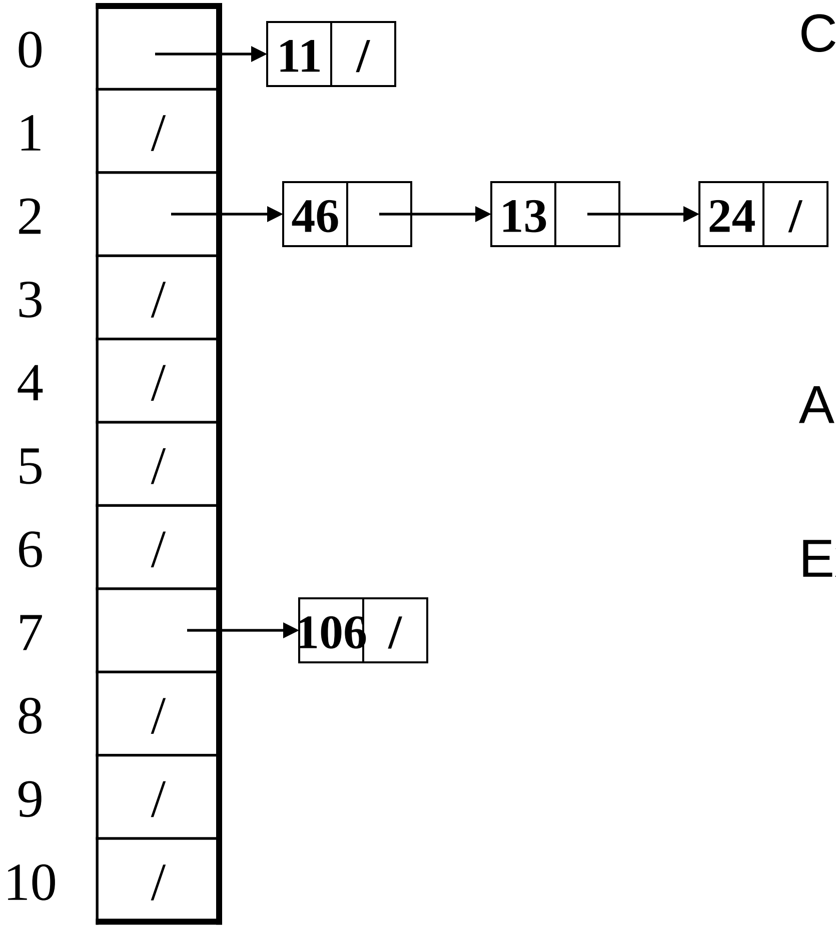
All keys that map to the same table location are kept in a linked list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

Handling Collisions: Separate Chaining



Chaining:

All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

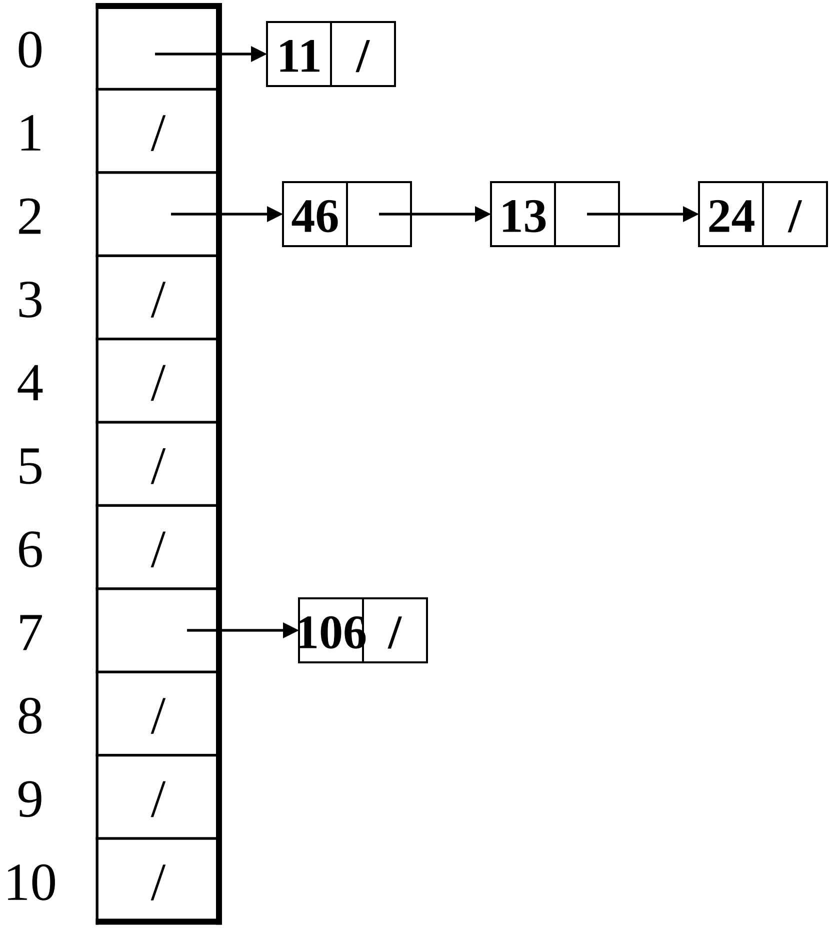
insert 11, 24, 106, 13, 46
with mod hashing
and **`TableSize`** = 11

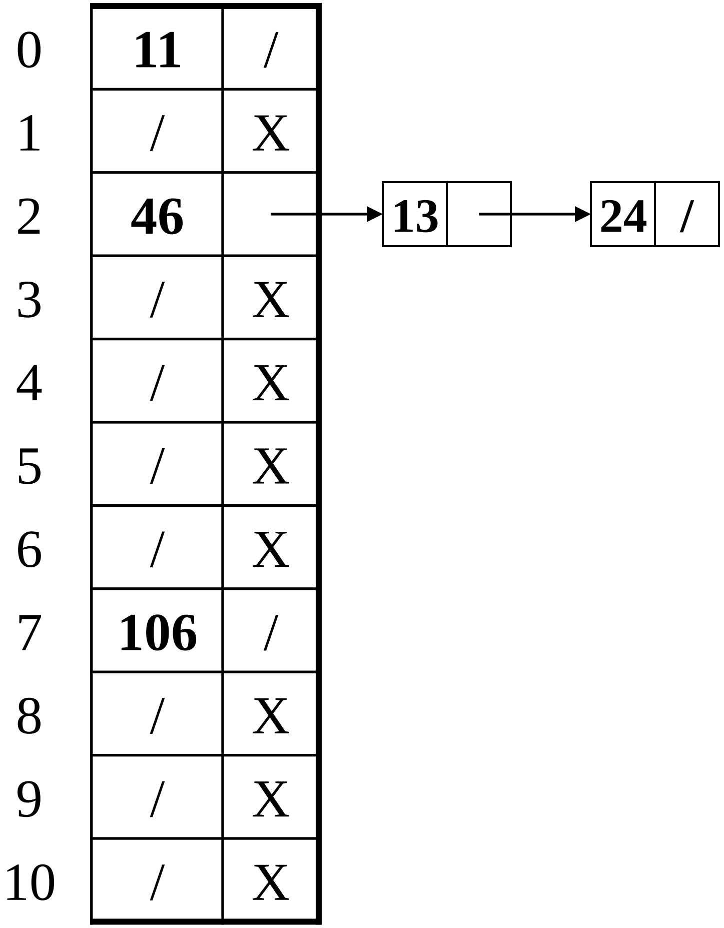
Thoughts on chaining

- Worst-case time for **`contains`**?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth doing extra work to avoid this worst case

- Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  - Linked list vs. array vs. chunked list (lists should be short!)
  - Move-to-front
  - Maybe leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off…

Time vs. space (constant factors only here)





More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← **number of elements**

Under chaining, the average number of elements per bucket is ___

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← **number of elements**

Under chaining, the average number of elements per bucket is λ

So if some puts are followed by *random* contains, then on average:

- Each unsuccessful **`contains`** compares against ____ items

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\mathrm{N}}{\mathrm{TableSize}}$$

← **number of elements**

Under chaining, the average number of elements per bucket is λ

So if some puts are followed by *random* contains, then on average:

- Each unsuccessful **`contains`** compares against λ items
- Each successful **`contains`** compares against _____ items

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← **number of elements**

Under chaining, the average number of elements per bucket is λ

So if some puts are followed by *random* contains, then on average:

- Each unsuccessful **`contains`** compares against λ items
- Each successful **`contains`** compares against $\lambda / 2$ items

So we like to keep λ fairly low (e.g., 1 or 1.5 or 2) for chaining

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	/

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	10
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Probing hash tables

Trying the next spot is called probing (also called open addressing)

- We just did linear probing
  - `i`th probe was `(h(key) + i) % TableSize`

Open addressing does poorly with high load factor λ

- So want larger tables
- Too many probes means no more $O(1)$

Other operations

`put` finds an open table position using a probe function

What about **`contains`**?

- Must use same probe function to “retrace the trail” for the data
- Unsuccessful search when reach empty position

What about **`remove`**?

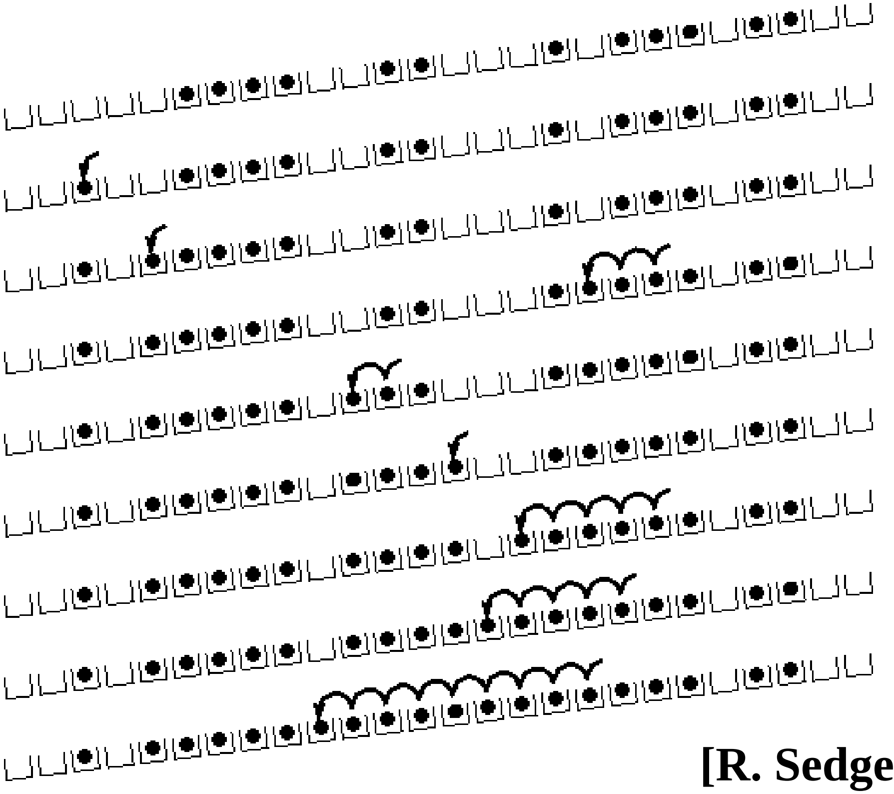
- ***Must*** use “lazy” deletion. Why?
  - Marker indicates “no data here, but don’t stop probing”
- Note: **`remove`** with chaining is plain-old list-remove

(Primary) Clustering

It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (which is a good thing)

Tends to produce *clusters*, which lead to long probing sequences

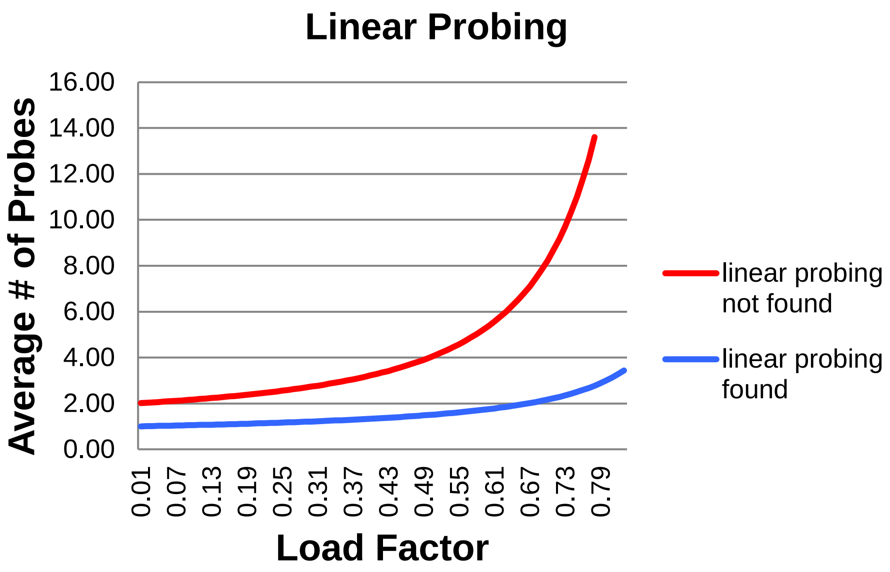
- Called primary clustering
- Saw this starting in our example



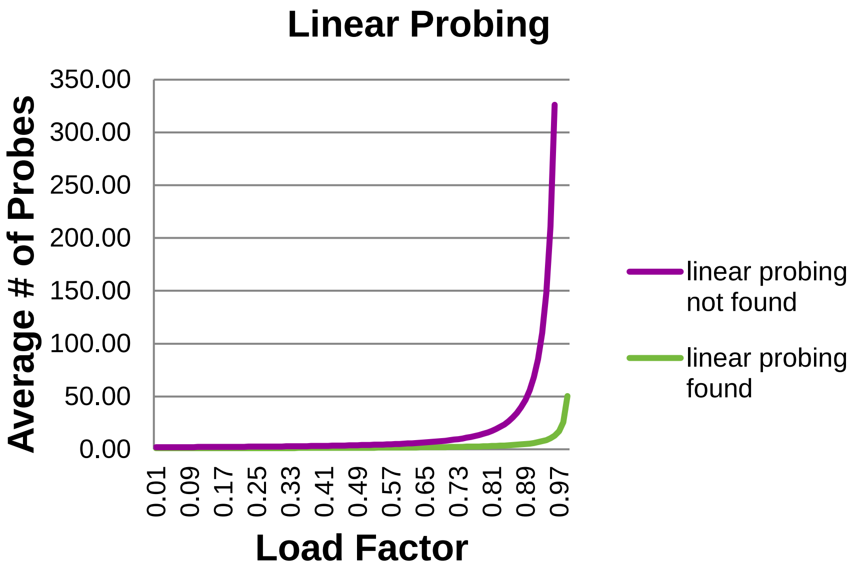
[R. Sedgewick]

Analysis of Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
  - It is “safe” in this sense: no infinite loop unless table is full
- Linear-probing performance degrades rapidly as table gets full



Linear Probing



Linear Probing

- By comparison, chaining performance is linear in λ and has no trouble with $\boldsymbol{\lambda>1}$

Quadratic probing

- We can avoid primary clustering by changing the probe function

  `(h(key) + f(i)) % TableSize`

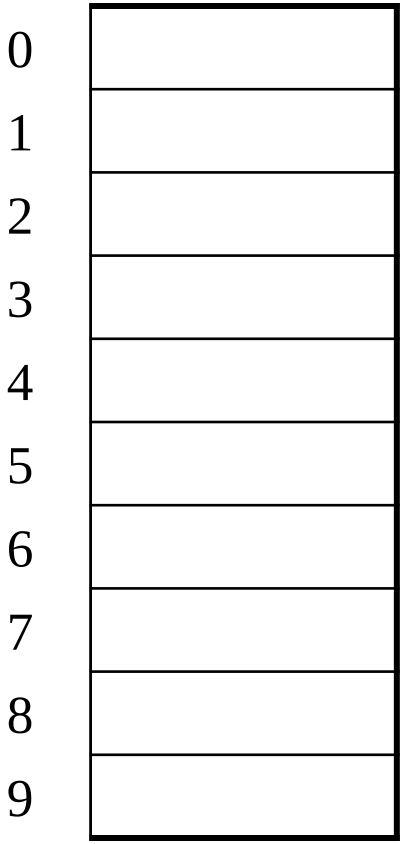
- A common technique is quadratic probing:

  `f(i)` $= i^2$

  - So probe sequence is:
    - 0th probe: **`h(key) % TableSize`**
    - 1st probe: **`(h(key) + 1) % TableSize`**
    - 2nd probe: **`(h(key) + 4) % TableSize`**
    - 3rd probe: **`(h(key) + 9) % TableSize`**
    - …
    - ith probe: **`(h(key)`** $+ i^2$**`) % TableSize`**

- Intuition: Probes quickly “leave the neighborhood”

Quadratic Probing Example



TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10

Insert:

89

18

49

58

79

Quadratic Probing Example

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

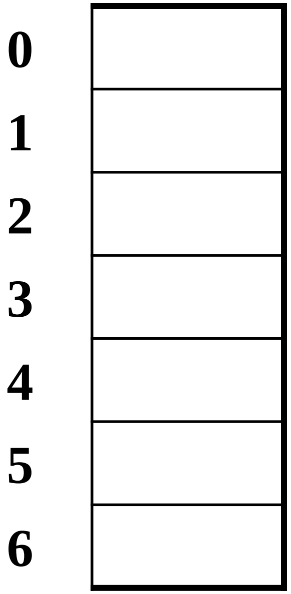
TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Another Quadratic Probing Example



TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76 (76 % 7 = 6)
40 (40 % 7 = 5)
48 (48 % 7 = 6)
5 (5 % 7 = 5)
55 (55 % 7 = 6)
47 (47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:
76 (76 % 7 = 6)
40 (40 % 7 = 5)
48 (48 % 7 = 6)
5 (5 % 7 = 5)
55 (55 % 7 = 6)
47 (47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Oh nooooo!: For all *n*, **`((n*n) +5) % 7 is 0, 2, 5, or 6`**

From Bad News to Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- Good news:
  - If **`TableSize`** is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
  - So: If you keep $\lambda < ½$ and **`TableSize`** is *prime*, no need to detect cycles

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
  - Re-apply the hash function to find the next index for each key

- With chaining, we get to decide what “too full” means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?

- For probing, half-full is a good rule of thumb

- New table size
  - Twice-as-big is a good idea, except that won’t be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you won’t grow more than 20-30 times