CSE 373: Hash Tables

Hunter Zahn

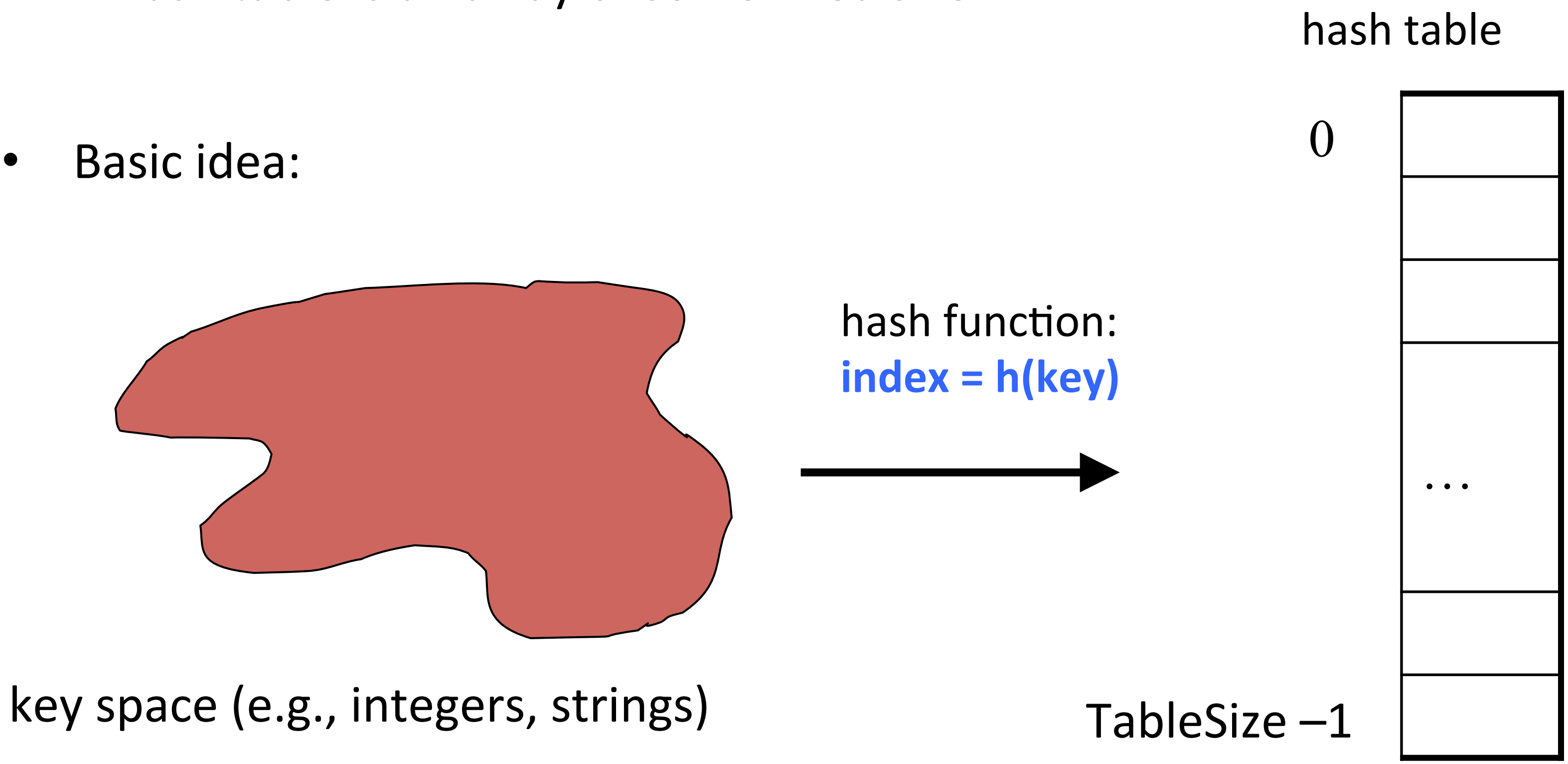
Summer 2016

Announcements

- HW 2 Due tonight (11PM)
- HW 3 out tomorrow (due July 18th, 11PM)

Hash Tables

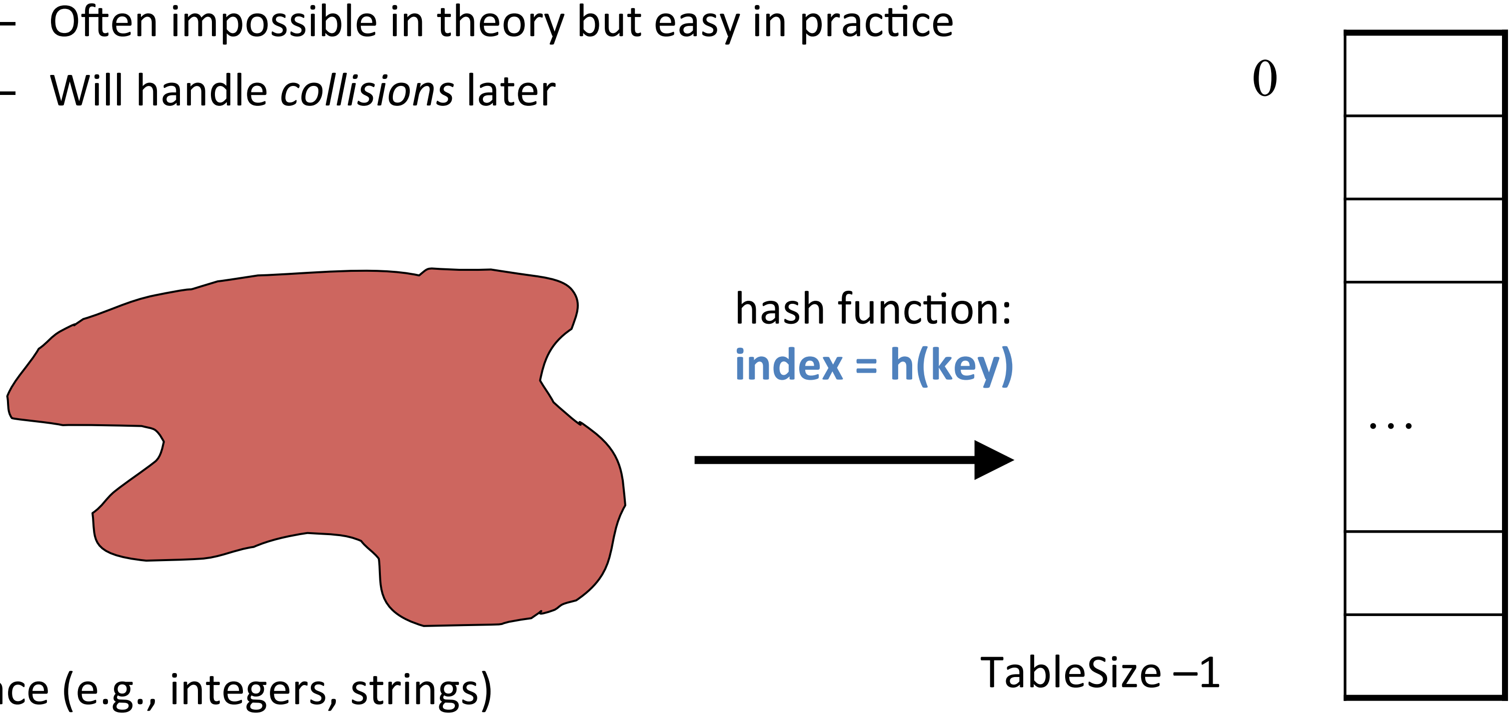
- Aim for constant-time (i.e., $O(1)$) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some often-reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:

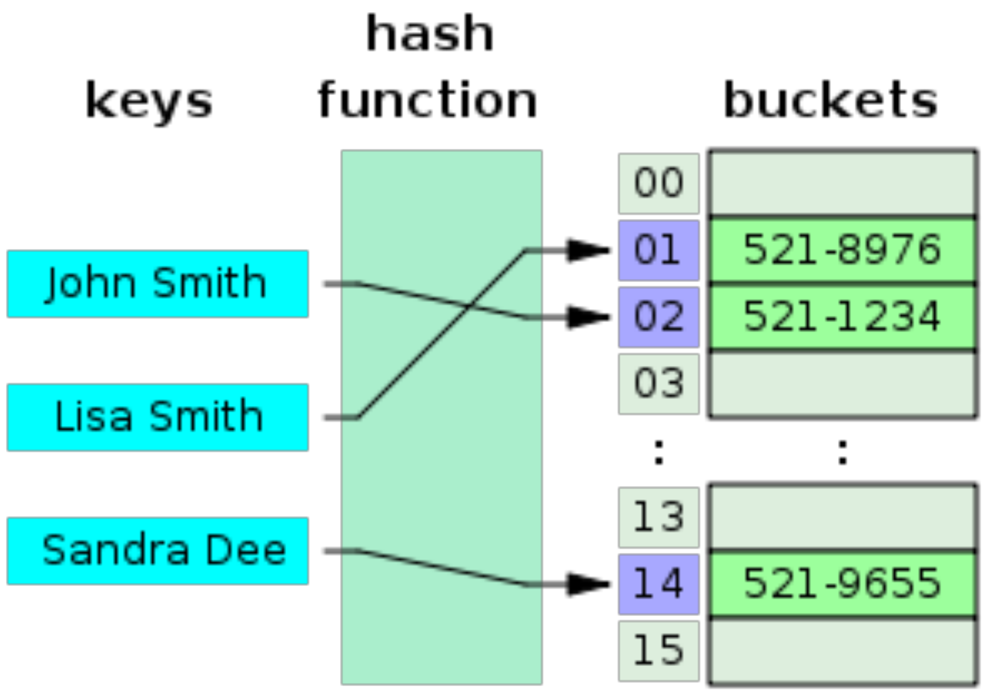


Hash functions

An ideal hash function:

- Fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory but easy in practice
  - Will handle *collisions* later




hash function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

Collision resolution

Collision:

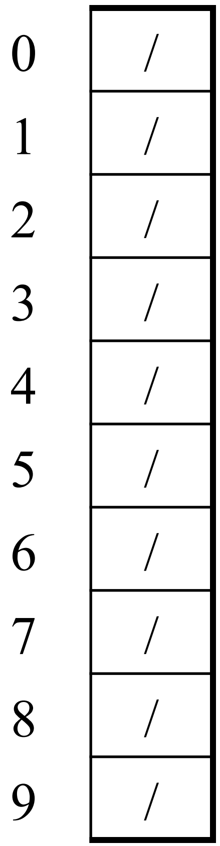
When two keys map to the same location in the hash table

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

So hash tables should support collision resolution

– Ideas?

Separate Chaining



Chaining:

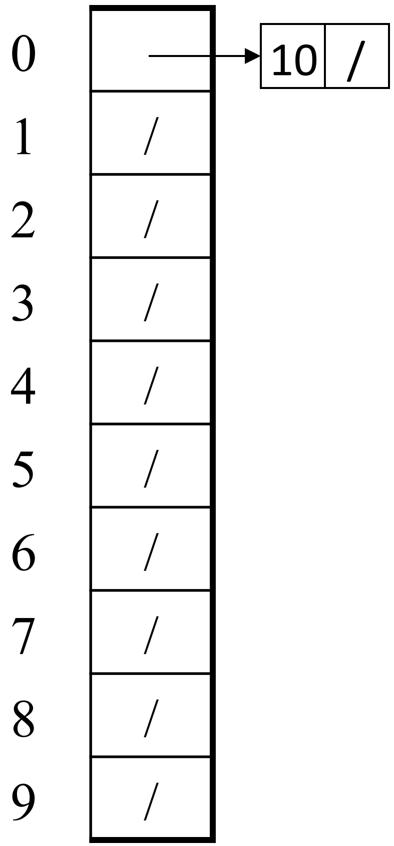
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

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

As easy as it sounds

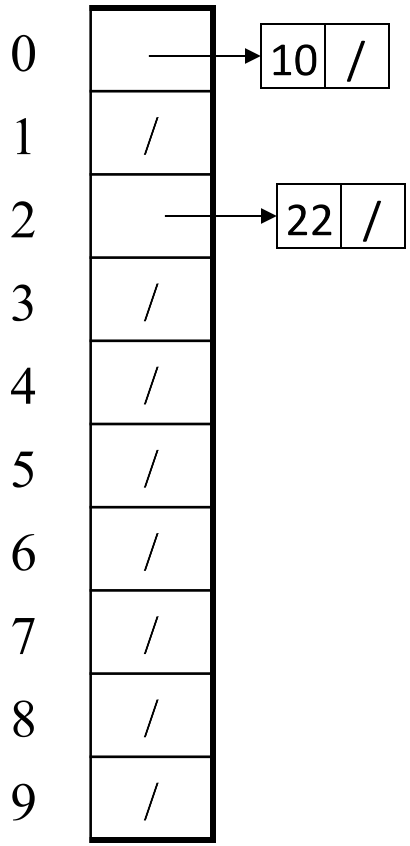
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and `TableSize` = 10

Separate Chaining



Chaining:

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

As easy as it sounds

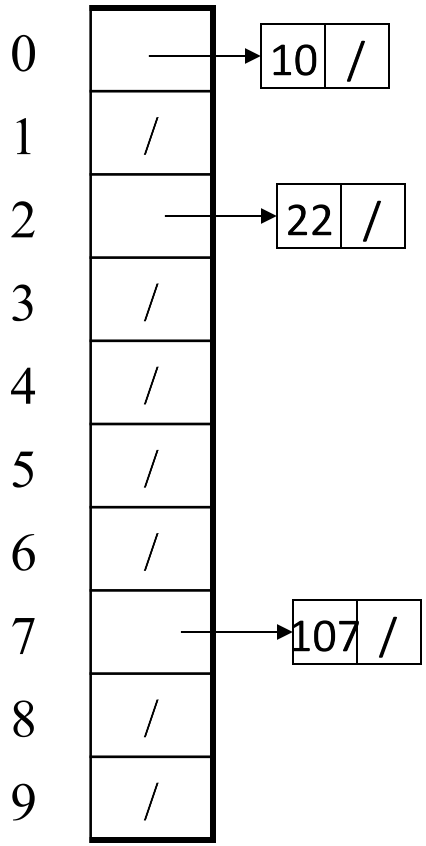
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

Separate Chaining



Chaining:

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

As easy as it sounds

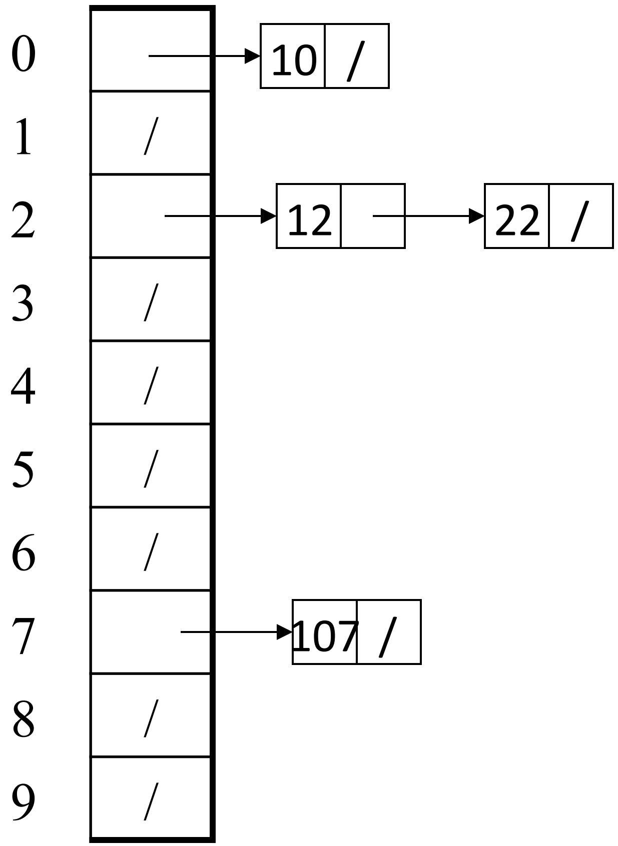
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

Separate Chaining



Chaining:

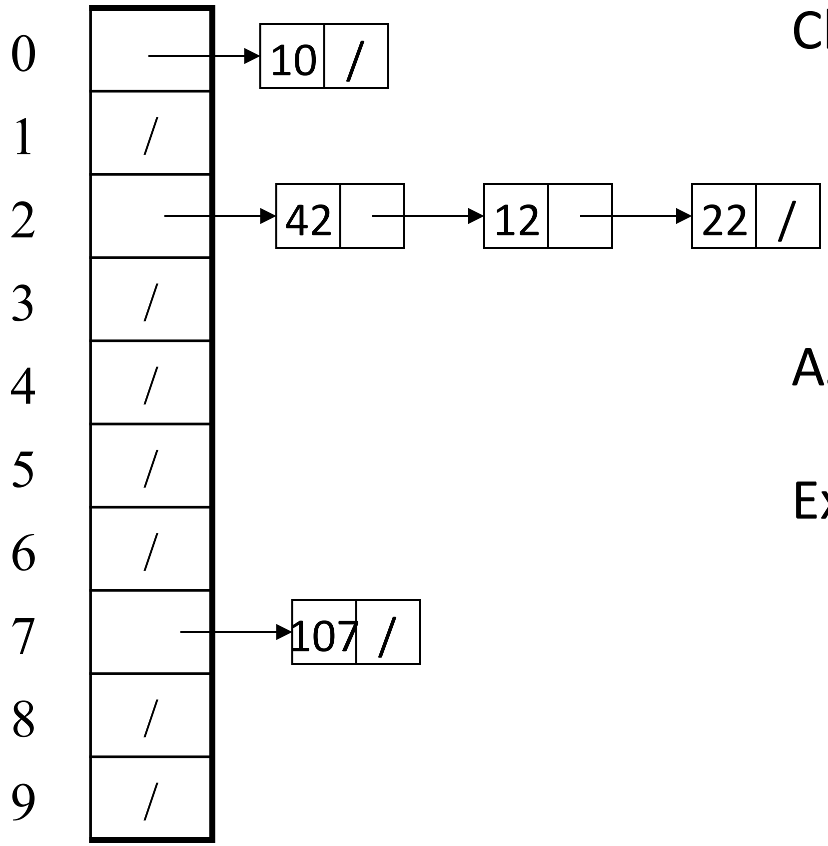
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

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

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42 with mod hashing and **`TableSize`** = 10

More rigorous chaining analysis

Definition: The load factor, $\boldsymbol{\lambda}$, of a hash table is

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

$\leftarrow$ number of elements

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

So if some inserts are followed by *random* finds, then on average:

- Each "unsuccessful" `find` compares against λ items

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

Deleting an element using Separate 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

Open addressing

This is *one example* of open addressing

In general, open addressing means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called probing

- We just did linear probing
  - **i**th probe was **(h(key) + i) % TableSize**
- In general have some probe function **f** and use **h(key) + f(i) % TableSize**

Open addressing does poorly with high load factor λ

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

Open Addressing

Write pseudocode for find(), assuming everything we've inserted is in the table.

Deletion in open addressing

- Brainstorm!

Deletion in Open Addressing

0	
1	
2	16
3	23
4	59
5	
6	76

h(k) = k % 7
Linear probing

Delete(23)
Find(59)
Insert(30)

Need to keep track of deleted items... leave a "marker"

Open Addressing

What will our pseudocode for find() look like if we're using lazy deletion?

Other operations

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

What about **`find`**?

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

What about **`delete`**?

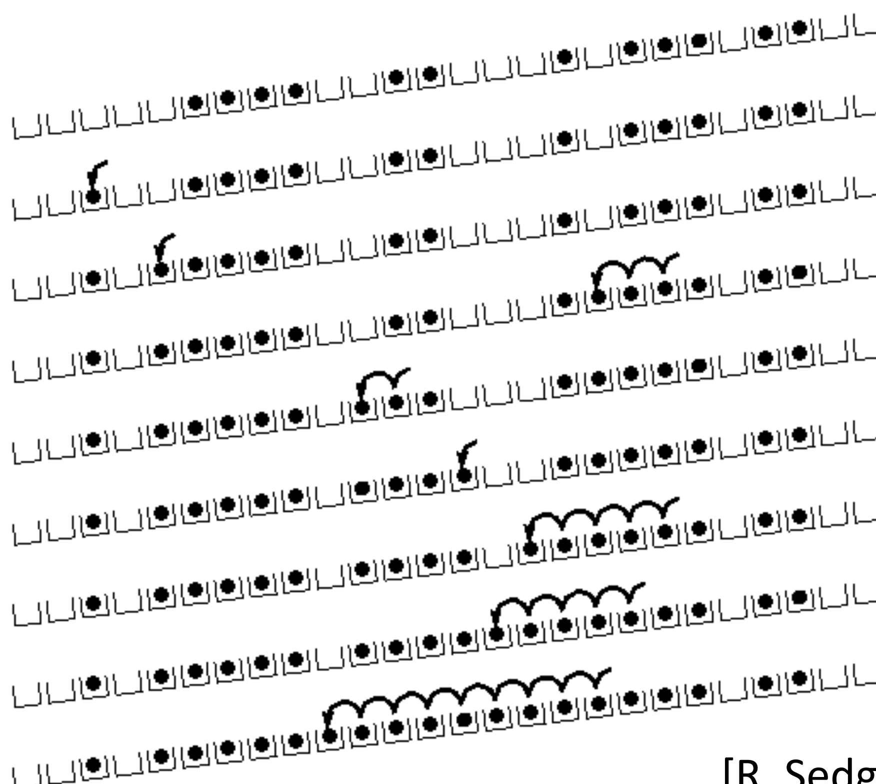
- ***Must*** use "lazy" deletion. Why?
  - Marker indicates "no data here, but don't stop probing"
- Note: **`delete`** with chaining is plain-old list-remove

(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

- **Trivial fact**: For any $\lambda < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

- **Non-trivial facts we won't prove**:
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

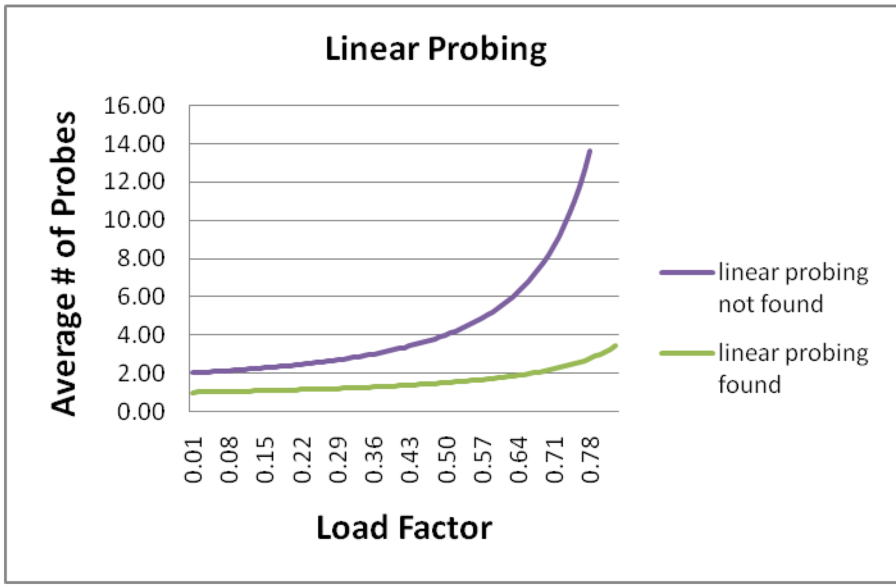
  - Successful search:

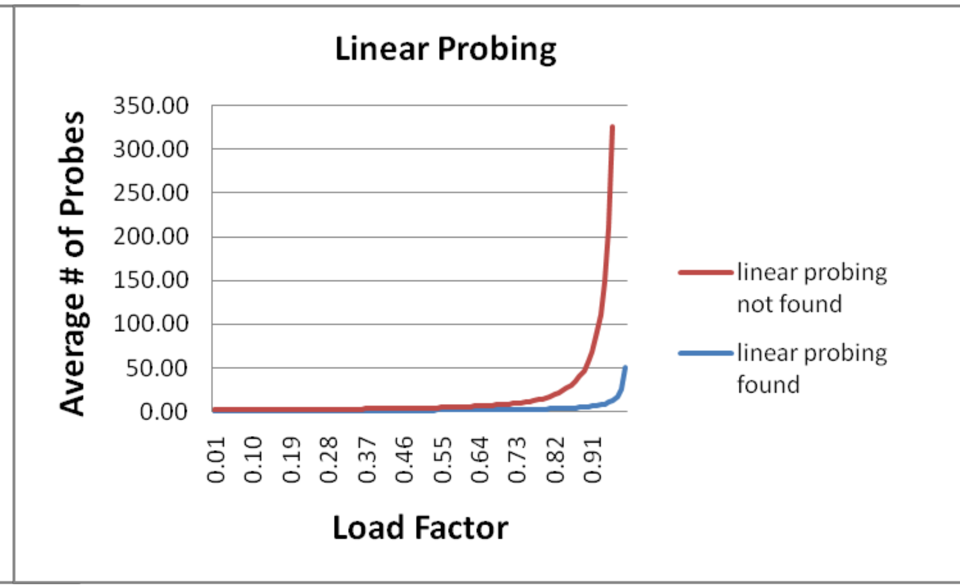
$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- This is pretty bad: need to leave sufficient empty space in the table to get decent performance

In a chart

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)





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

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

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

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

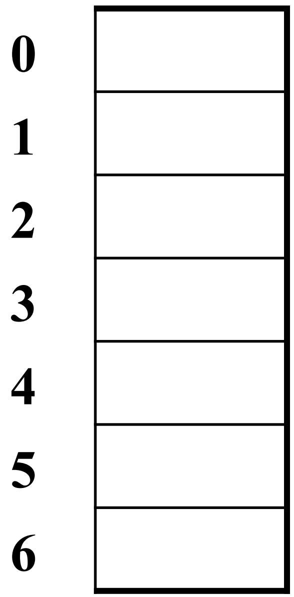
18

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)

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

- Excel shows takes "at least" 50 probes and a pattern
- Proof uses induction and $(n^2+5) \% 7 = ((n-7)^2+5) \% 7$
    - In fact, for all *c* and *k*, $(n^2+c) \% k = ((n-k)^2+c) \% k$

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
  - Optional
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime **`T`** and **`0 < i,j < T/2`** where **`i`** $\neq$ **`j`**,
      $(k + i^2) \% T \neq (k + j^2) \% T$ (i.e., no index repeat)

Quadratic Probing:
Success guarantee for $\lambda < \frac{1}{2}$

Assertion #1: If T = TableSize is **prime** and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in $\leq$ T/2 probes

Assertion #2: For prime T and all $\mathbf{0 \leq i,j \leq T/2}$ and $\mathbf{i \neq j}$,

$$\mathbf{(h(K) + i^2)\ \%\ T \neq (h(K) + j^2)\ \%\ T}$$

Assertion #3: Assertion #2 proves assertion #1.

Quadratic Probing:
Success guarantee for $\lambda < ½$

We can prove assertion #2 by contradiction.

Suppose that for some i $\neq$ j, `0 ≤ i,j ≤ T/2`, prime T:

$$\texttt{(h(K) + i}^2\texttt{) \% T = (h(K) + j}^2\texttt{) \% T}$$

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: no problem with keys initially hashing to the same neighborhood

- But it's no help if keys initially hash to the same index
  - Called secondary clustering

- Can avoid secondary clustering with a probe function that depends on the key: double hashing…

Double hashing

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely that for some *key*, `h(key) == g(key)`
- So make the probe function `f(i) = i*g(key)`

Probe sequence:

- 0^{th} probe: `h(key) % TableSize`
- 1^{st} probe: `(h(key) + g(key)) % TableSize`
- 2^{nd} probe: `(h(key) + 2*g(key)) % TableSize`
- 3^{rd} probe: `(h(key) + 3*g(key)) % TableSize`
- …
- i^{th} probe: `(h(key) + i*g(key)) % TableSize`

Detail: Make sure `g(key)` cannot be **0**

Double Hashing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7
$h(K) = K \% 7$
$g(K) = 5 - (K \% 5)$

Insert(76) $76 \% 7 = 6$ and $5 - 76 \% 5 =$
Insert(93) $93 \% 7 = 2$ and $5 - 93 \% 5 =$
Insert(40) $40 \% 7 = 5$ and $5 - 40 \% 5 =$
Insert(47) $47 \% 7 = 5$ and $5 - 47 \% 5 =$
Insert(10) $10 \% 7 = 3$ and $5 - 10 \% 5 =$
Insert(55) $55 \% 7 = 6$ and $5 - 55 \% 5 =$

Double-hashing analysis

- Intuition: Because each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"

- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - **`h(key) = key % p`**
    - **`g(key) = q – (key % q)`**
    - **`2 < q < p`**
    - **`p`** and **`q`** are prime

More double-hashing facts

- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`
- Non-trivial facts we won't prove:

  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search (intuitive): $$\frac{1}{1-\lambda}$$
  - Successful search (less intuitive): $$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
- 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 open addressing, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except, uhm, 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

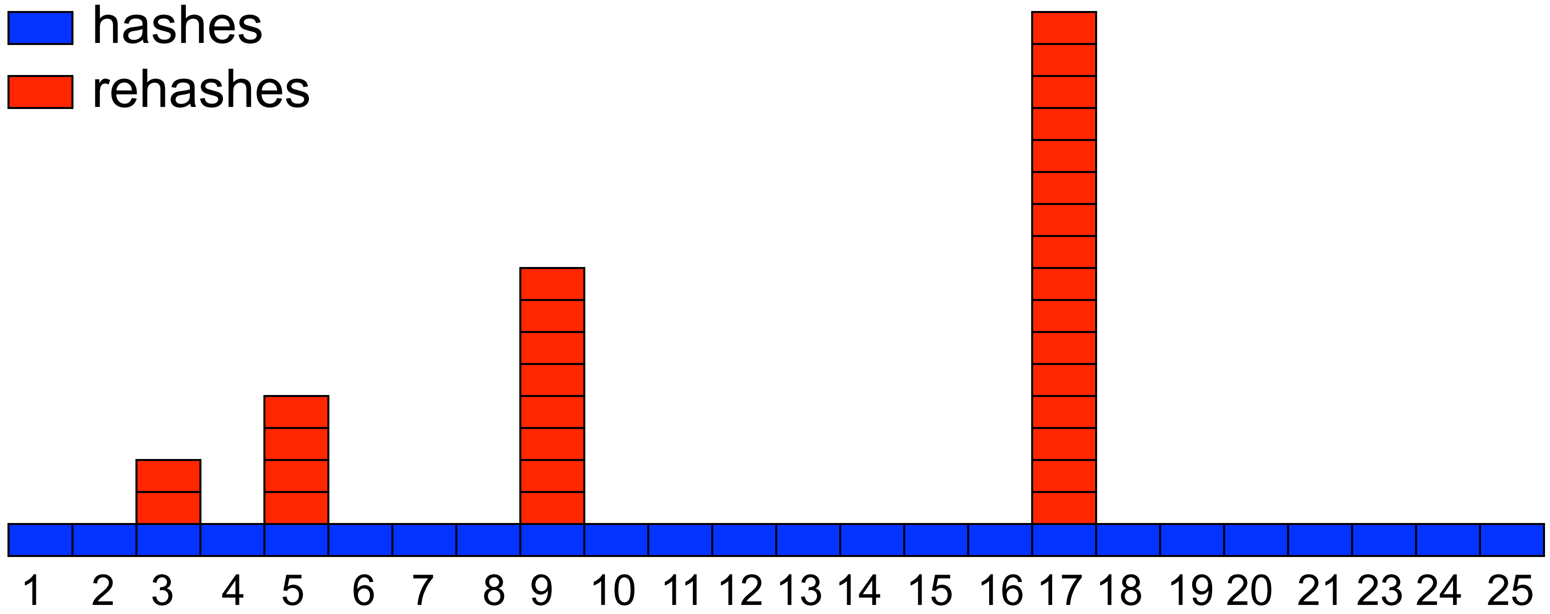
Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.



Amortized Analysis of Rehashing

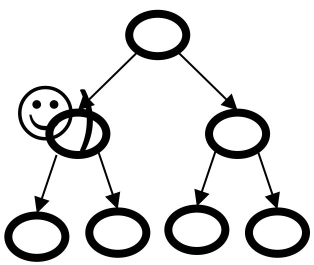
- Cost of inserting n keys is < 3n
- suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - n = 33, Total = 33 + 64 –2 = 95 < 99

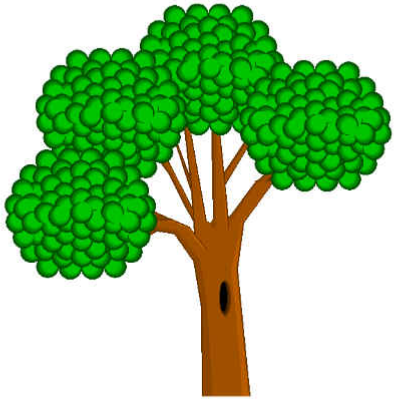
Terminology

We and the book use the terms
- "chaining" or "separate chaining"
- "open addressing"

Very confusingly,
- "open hashing" is a synonym for "chaining"
- "closed hashing" is a synonym for "open addressing"

(If it makes you feel any better,
most trees in CS grow upside-down ☺)





Equal objects must hash the same

- The Java library (and your project hash table) make a very important assumption that clients must satisfy…

  If `c.compare(a,b) == 0`, then we require

  `h.hash(a) == h.hash(b)`

- If you ever override equals
  - You need to override hashCode also in a consistent way
  - See CoreJava book, Chapter 5 for other "gotchas" with equals

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
  - But what is the cost of doing, e.g., findMin?
- Can use:
  - Separate chaining (easiest)
  - Open hashing (memory conservation, no linked list management)
  - Java uses separate chaining
- Rehashing has good amortized complexity.
- Also has a big data version to minimize disk accesses: extendible hashing. (See book.)