Hash Tables

Anna Baynes

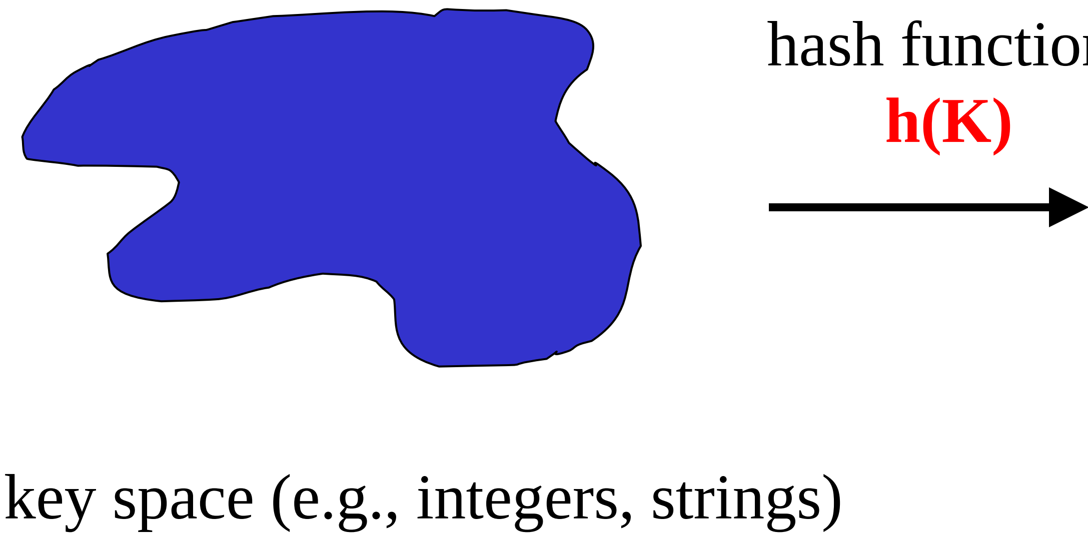
CSC 130

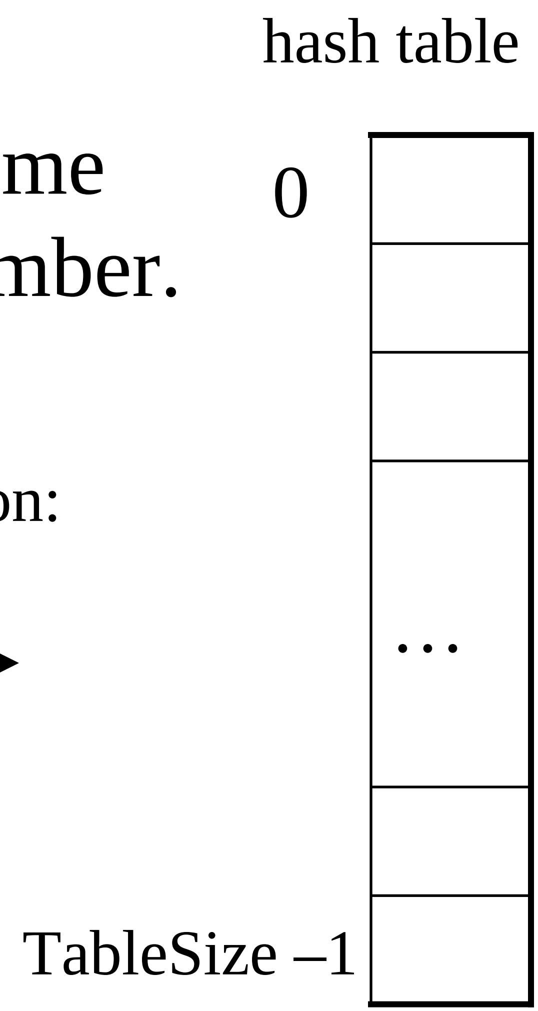
Dictionary Implementations So Far

	Unsorted linked list	Sorted Array	BST	AVL	Splay (amortized)
Insert					
Find					
Delete					

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:





Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Another Example

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	

Hash Functions

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Perfect Hash function:

Sample Hash Functions:

- key space = strings
- $s = s_0\ s_1\ s_2\ \ldots\ s_{k-1}$

1. $h(s) = s_0 \bmod TableSize$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i \right) \bmod TableSize$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \bmod TableSize$

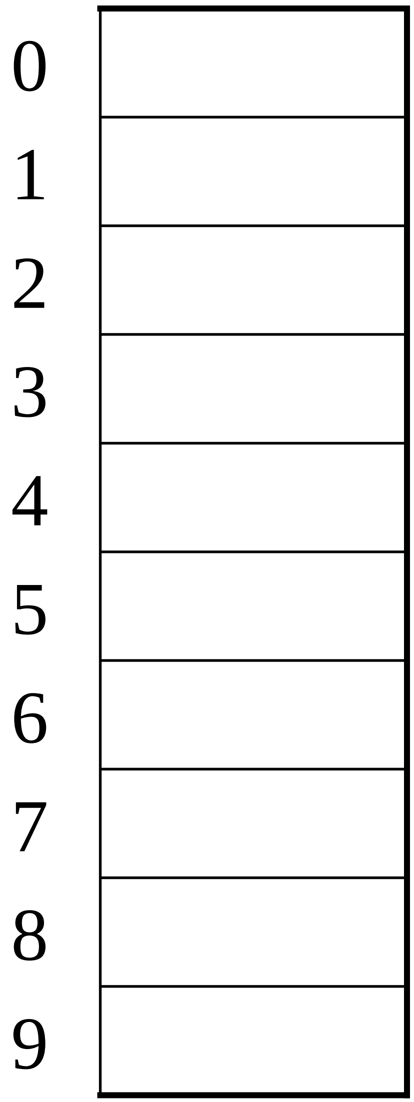
Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Separate Chaining



Insert:
10
22
107
12
42

- **<u>Separate chaining</u>:** All keys that map to the same hash value are kept in a list (or "bucket").

Analysis of find

- Defn: The load factor, λ, of a hash table is the ratio: $\frac{N}{M}$ $\leftarrow$ no. of elements $\leftarrow$ table size

For separate chaining, λ = average # of elements in a bucket

- Unsuccessful find:

- Successful find:

How big should the hash table be?

- For Separate Chaining:

tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10

    data hashes to 0, 3, <u>0</u>, 5, 1, <u>0</u>, <u>0</u>
  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, <u>9</u>, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

Open Addressing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Terminology Alert!

	"**Open** Hashing"	"Closed Hashing"
	equals	equals
Weiss	"Separate Chaining"	"**Open** Addressing"

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = $h(k) \bmod TableSize$

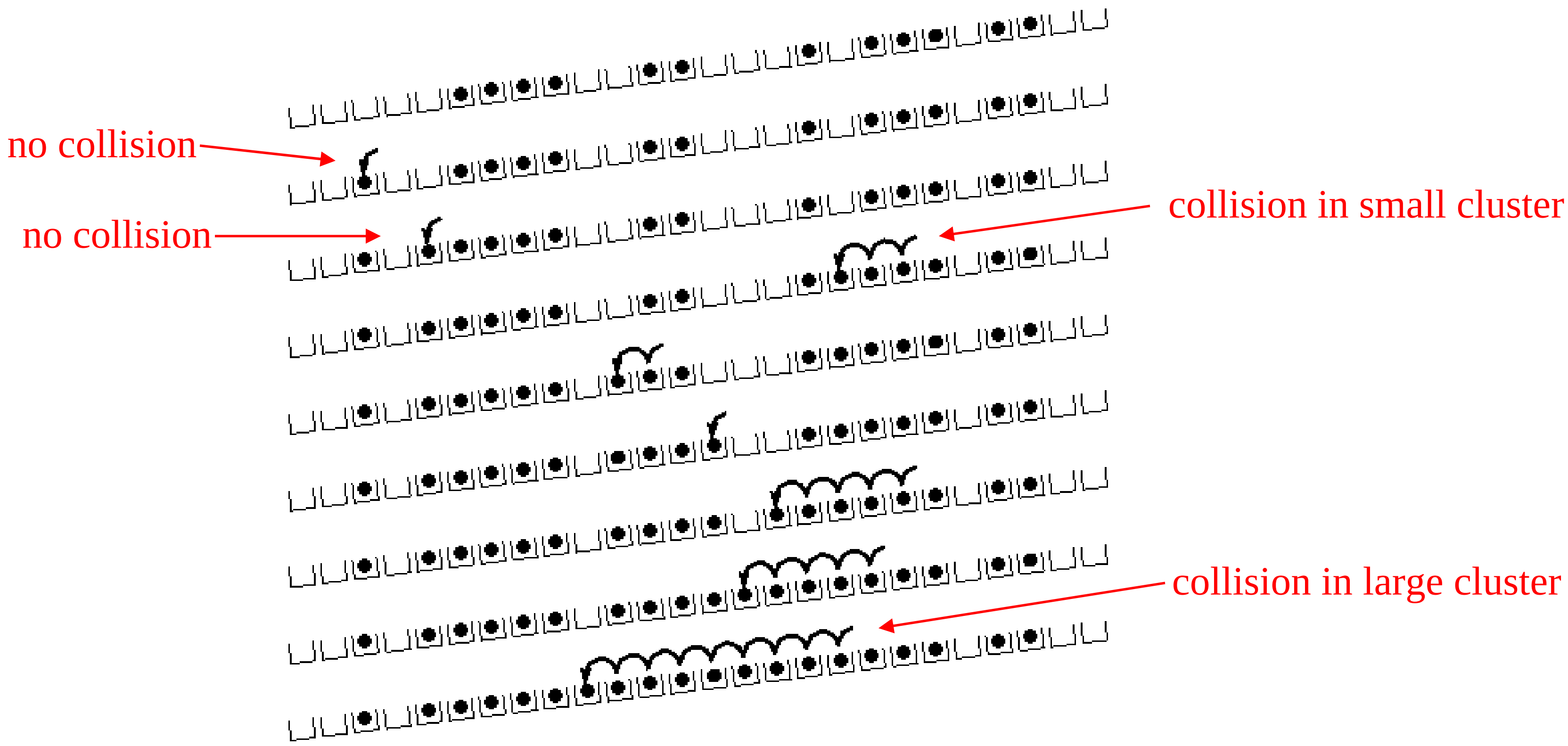
  1^{th} probe = $(h(k) + 1) \bmod TableSize$

  2^{th} probe = $(h(k) + 2) \bmod TableSize$

  . . .

  i^{th} probe = $(h(k) + i) \bmod TableSize$

Linear Probing – Clustering



[R. Sedgewick]

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - successful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$
  - unsuccessful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$
- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$

Quadratic Probing

Less likely to encounter Primary Clustering

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

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = (h(k) + i^2) mod TableSize

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

Quadratic Probing Example

insert(76) — $76\%7 = 6$

insert(40) — $40\%7 = 5$

insert(48) — $48\%7 = 6$

insert(5) — $5\%7 = 5$

insert(55) — $55\%7 = 6$

0	
1	
2	
3	
4	
5	
6	76

But… insert(47) — $47\%7 = 5$

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

- If size is prime and $\lambda < ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all `0` $\leq$ `i,j` $\leq$ `size/2` and `i` $\neq$ `j`

    `(h(x) + i²) mod size` $\neq$ `(h(x) + j²) mod size`
  - by contradiction: suppose that for some i $\neq$ j:

    `(h(x) + i²) mod size = (h(x) + j²) mod size`

    $\Rightarrow$ `i² mod size = j² mod size`

Quadratic Probing: Properties

- For *any* $\lambda < ½$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad
- But what about keys that hash to the same *spot*?
  - ***Secondary Clustering!***

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Resolving Collisions with Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

$M =$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Rehashing

Idea: 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?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically “significant” fields of objects

hashCode() and equals()

- To work right, particularly with collection classes like HashMap, hashCode() and equals() must obey this rule:

    if a.equals(b) then it must be true that

    a.hashCode() == b.hashCode()

  - Why?
- Reverse is not required

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.

Collision resolution

Collision:

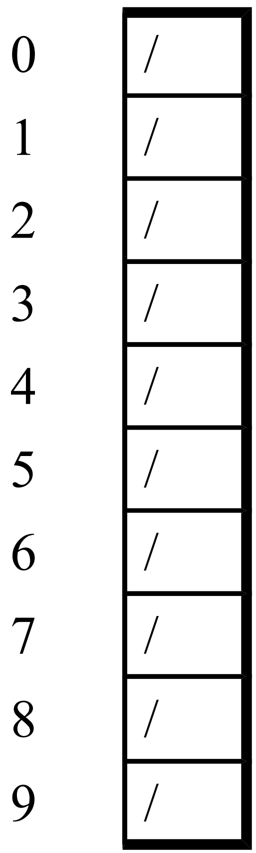
When two keys map to the same location in the hash table

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

So hash tables should support collision resolution

– Ideas?

Separate Chaining



Chaining:

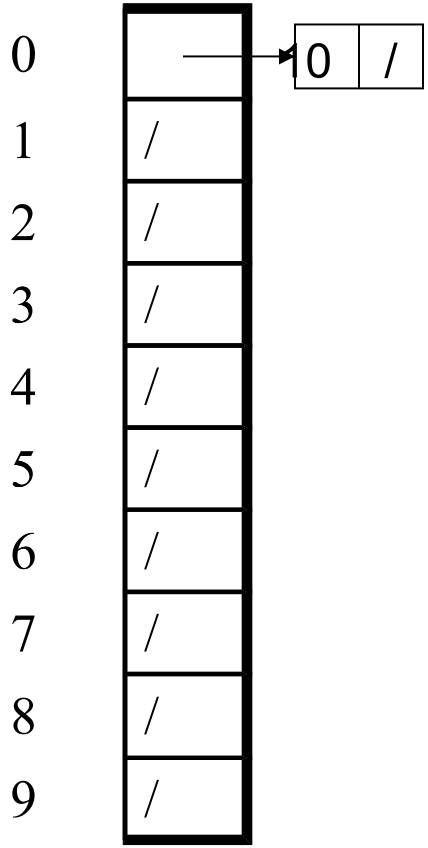
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

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

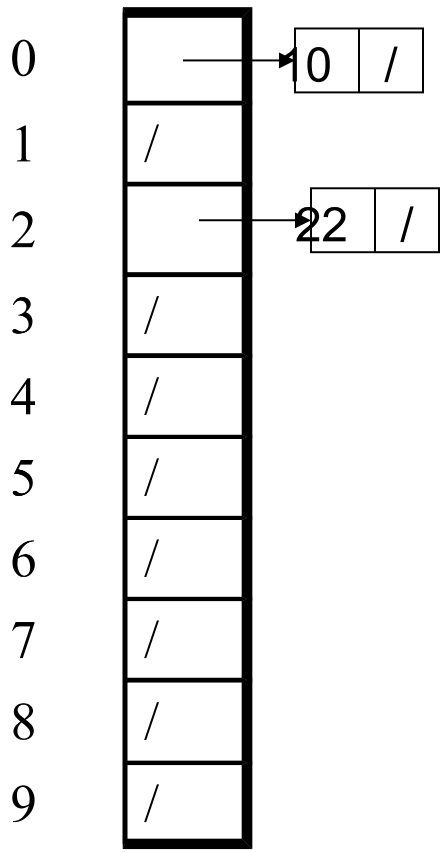
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

Separate Chaining



Chaining:

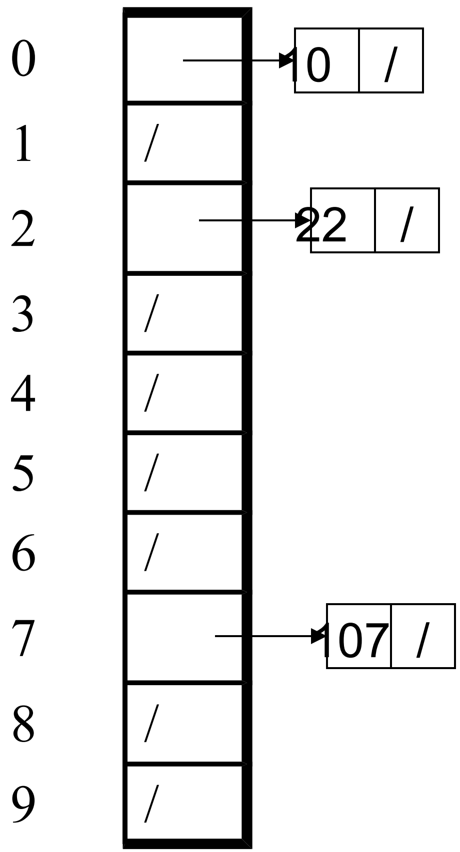
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

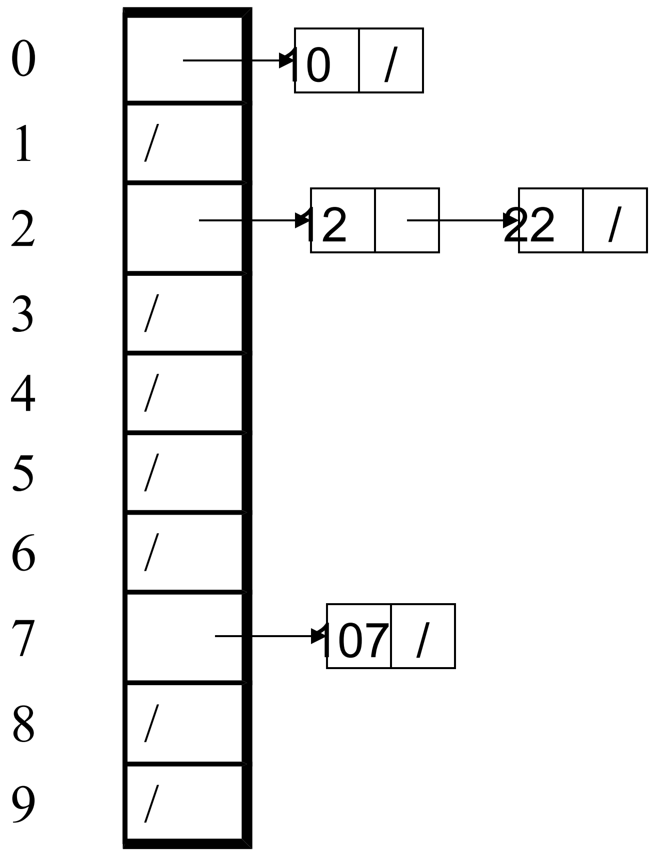
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

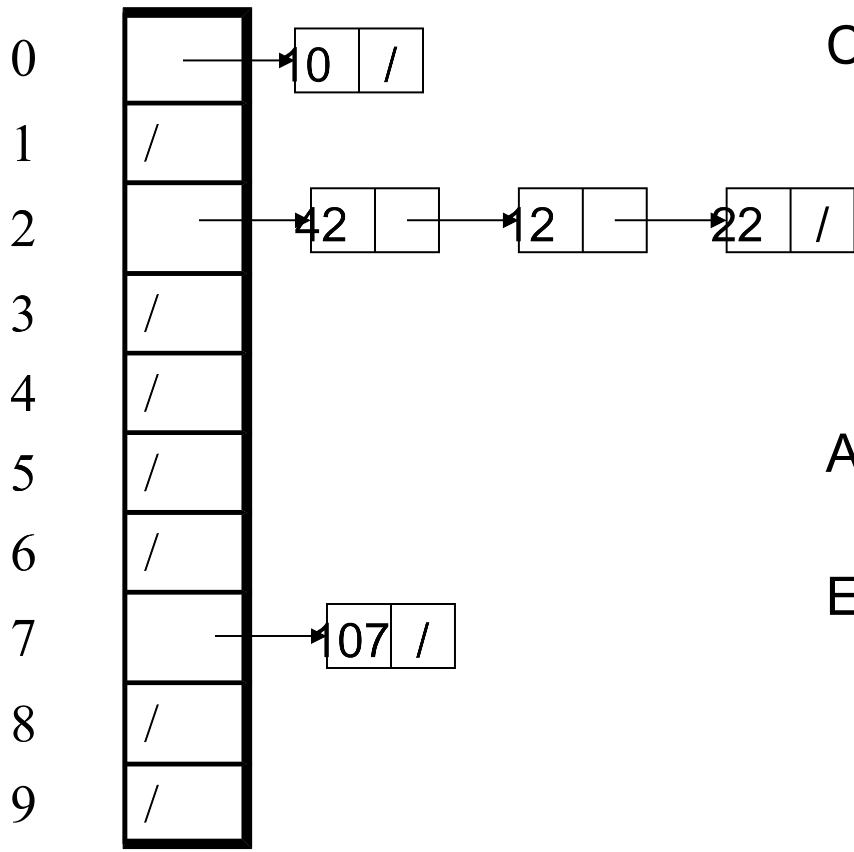
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

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:

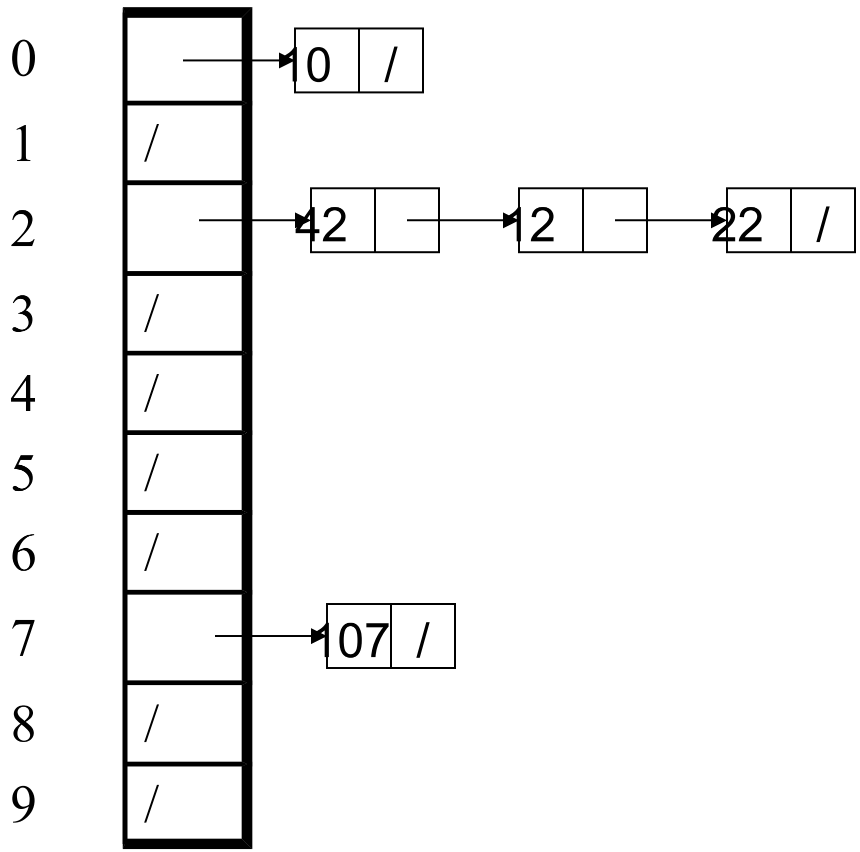
insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

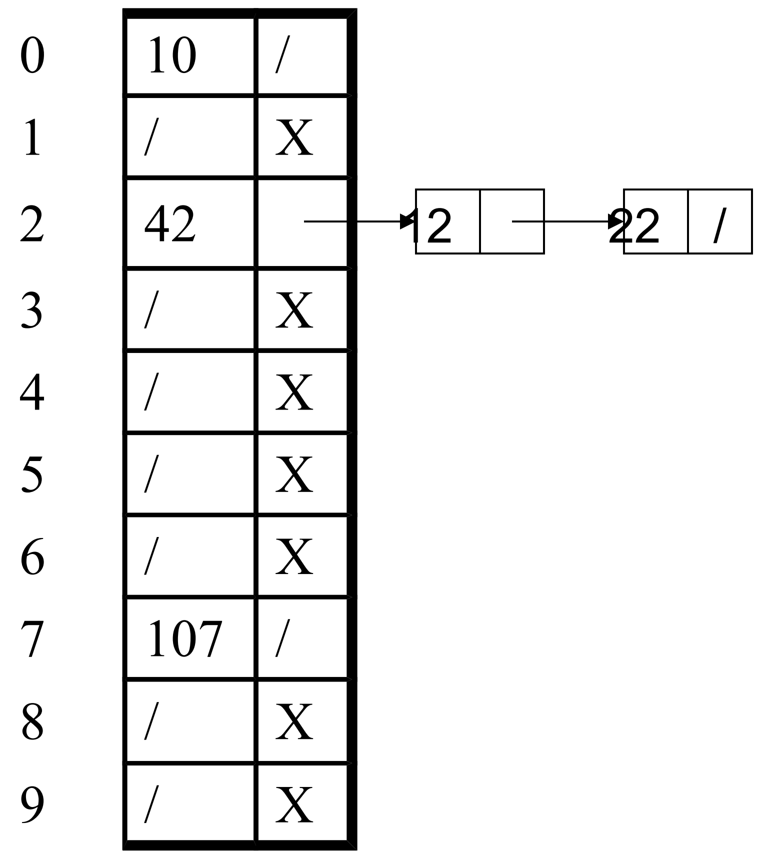
Thoughts on chaining

- Worst-case time for **`find`**?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)

- Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  - Linked list vs. array vs. chunked list (lists should be short!)
  -

Time vs. space (constant factors only here)





More Rigorous Chaining Analysis

Definition: The load factor, 2)21, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$ ${}^{2}_{4}$ number of elements

Under chaining, the average number of elements per bucket is

____

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

- Each unsuccessful `find` compares against ____ items
-

More rigorous chaining analysis

Definition: The load factor, 2)21, of a hash table is

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

$\frac{2}{4}$ number of elements

Under chaining, the average number of elements per bucket is 2)21

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

- Each unsuccessful `find` compares against 2)21 items
-

So we like to keep 2)21 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

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 2)21

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

Terminology

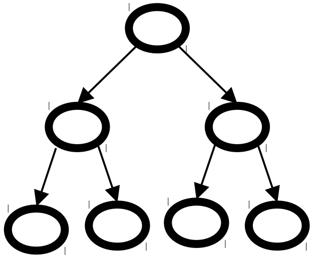
We and the book use the terms

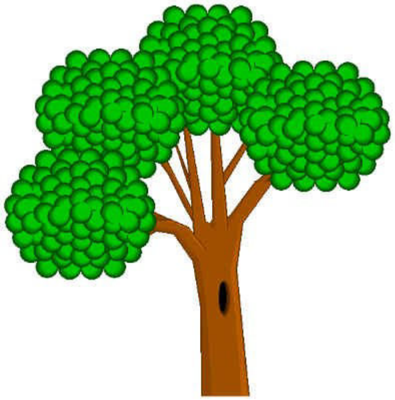
- "chaining" or "separate chaining"
- "open addressing"

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 □)





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`**?

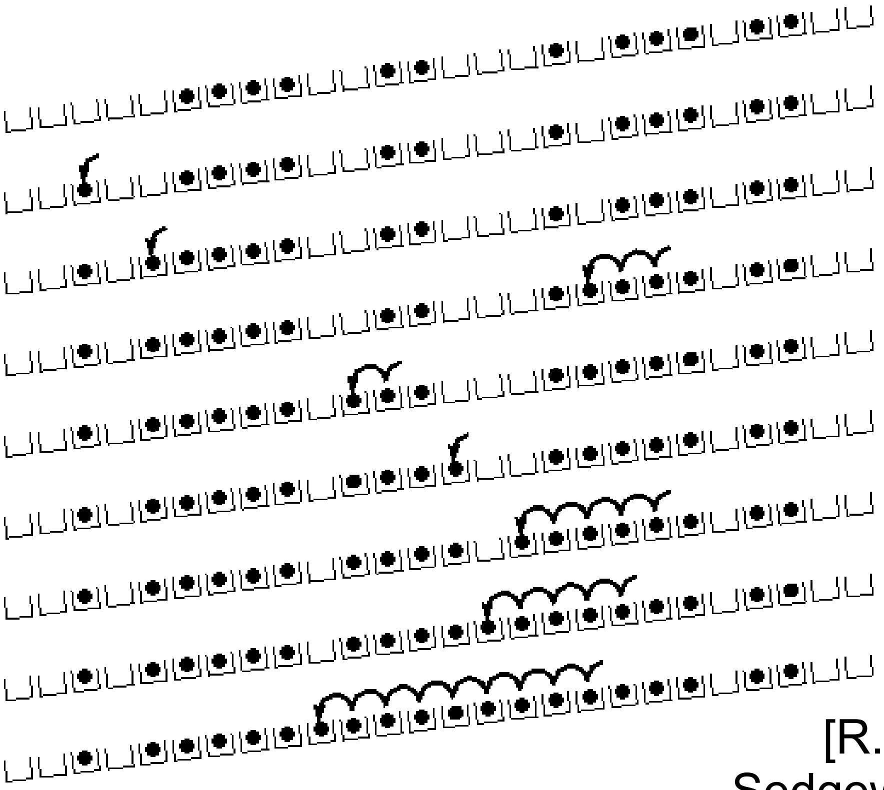
- ***Must*** use “lazy” deletion. Why?
  - Marker indicates “no data here, but don’t stop probing”
- Note: **`delete`** with chaining is plain-old list-remove

(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

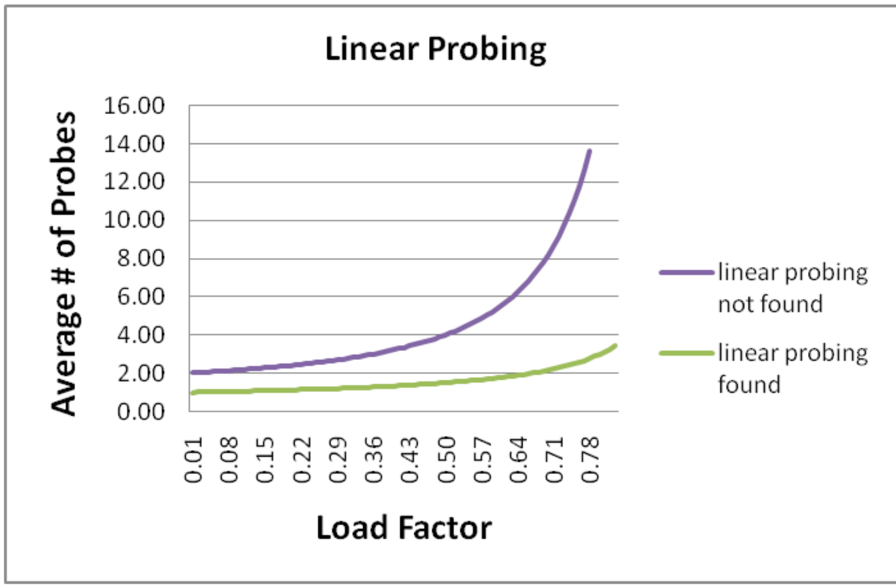
- Called primary clustering
- Saw this starting in our example

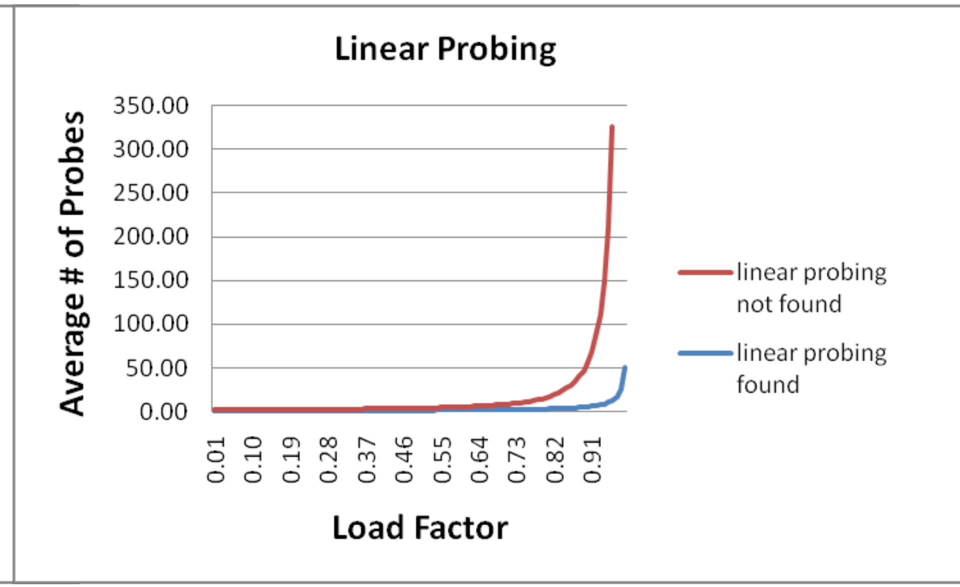


[R. Sedgewick]

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 2)21 and has no trouble with 2)21***>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

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

76 (76 % 7 = 6)

77 (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)
77 (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)
77 (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)
77 (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)
77 (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)
77 (40 % 7 = 5)
48 (48 % 7 = 6)
5 (5 % 7 = 5)
55 (55 % 7 = 6)
47 (47 % 7 = 5)

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 2)21 < ½, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
  - So: If you keep 2)21 < ½ and **`TableSize`** is *prime*, no need to detect cycles
  - Proof in textbook
    - Also, slightly less detailed
    - Key fact: For prime **`T`** and **`0 < i,j < T/2`** where **`i`** $\frac{4}{4}$ **`j`**,

$$(k + i^2) \% T \ \tfrac{4}{4}\ (k + j^2) \% T$$ (i.e., no index repeat)

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:

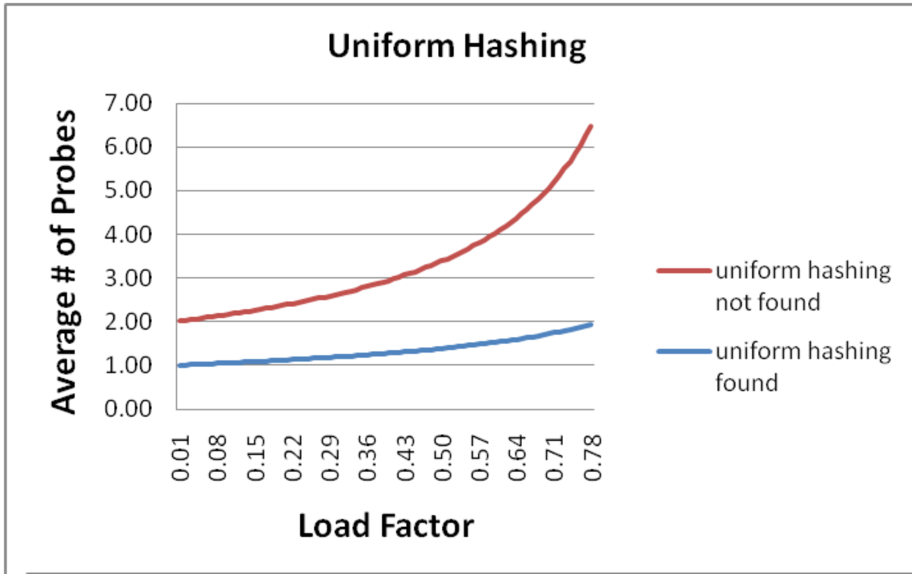
- 0^{th} probe: **`h(key) % TableSize`**
- 1^{st} probe: **`(h(key) + g(key)) % TableSize`**
- 2^{nd} probe: **`(h(key) + 2*g(key)) % TableSize`**
- 3^{rd} probe: **`(h(key) + 3*g(key)) % TableSize`**
- …
- i^{th} probe: **`(h(key) + i*g(key)) % TableSize`**

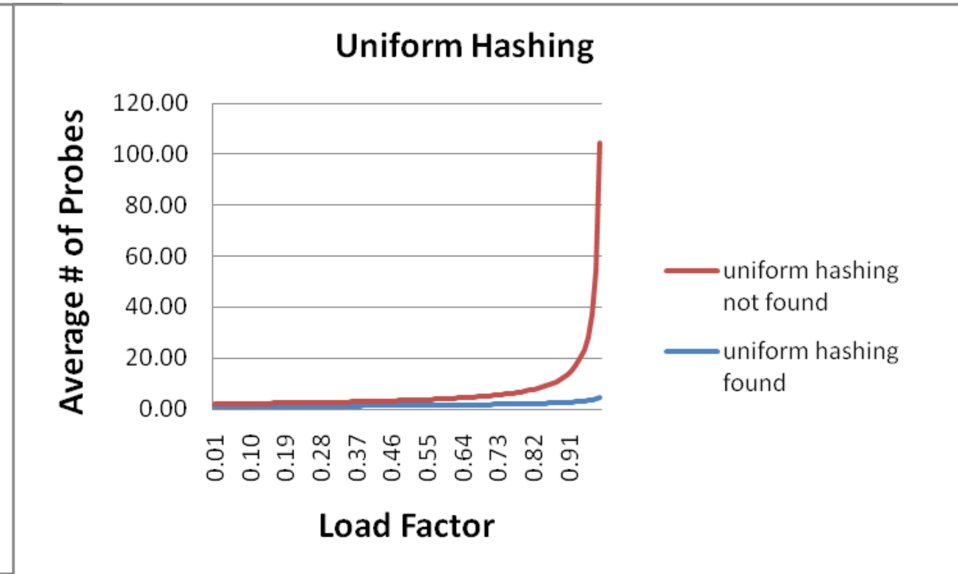
Detail: Make sure **`g(key)`** cannot be **`0`**

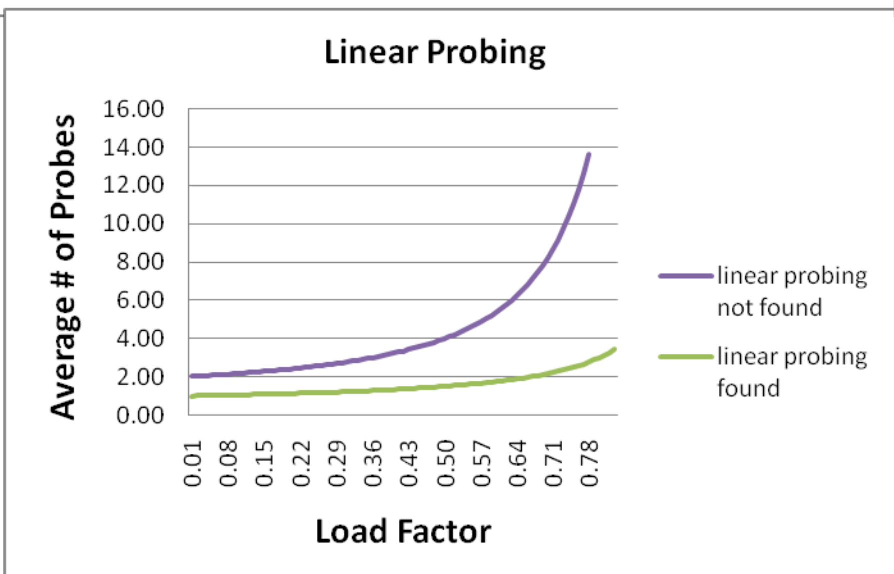
Double-hashing analysis

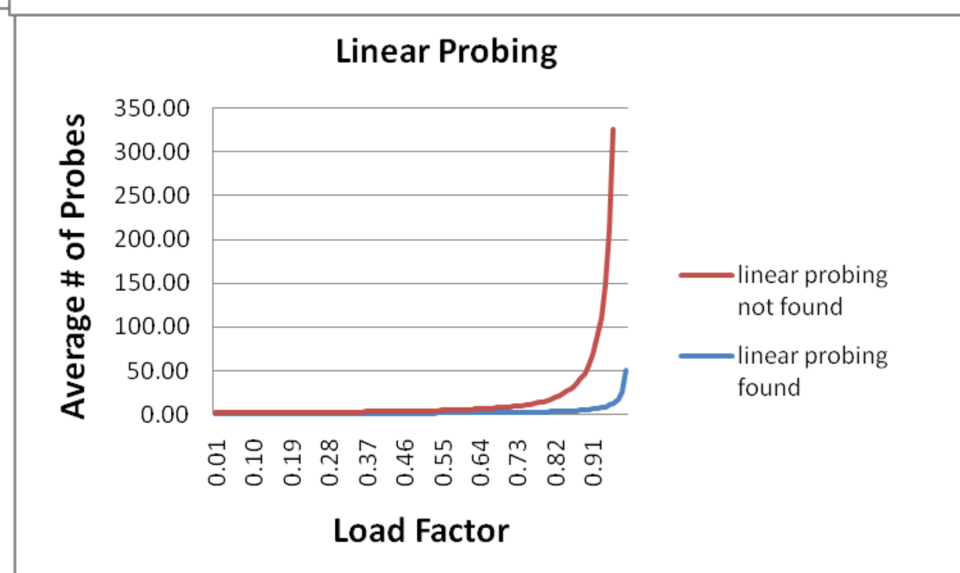
- Intuition: Because each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"
- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - It is known that 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

Charts


Uniform Hashing
Average # of Probes
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
0.01 0.08 0.15 0.22 0.29 0.36 0.43 0.50 0.57 0.64 0.71 0.78
Load Factor
uniform hashing not found
uniform hashing found


Uniform Hashing
Average # of Probes
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0.01 0.10 0.19 0.28 0.37 0.46 0.55 0.64 0.73 0.82 0.91
Load Factor
uniform hashing not found
uniform hashing found


Linear Probing
Average # of Probes
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0.01 0.08 0.15 0.22 0.29 0.36 0.43 0.50 0.57 0.64 0.71 0.78
Load Factor
linear probing not found
linear probing found


Linear Probing
Average # of Probes
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0.01 0.10 0.19 0.28 0.37 0.46 0.55 0.64 0.73 0.82 0.91
Load Factor
linear probing not found
linear probing found

Where are we?

- *Chaining* is easy
  - **`find`**, **`delete`** proportional to load factor on average
  - **`insert`** can be constant if just push on front of list

- *Open addressing* uses probing, has clustering issues as table fills
  - Why use it:
    - Less memory allocation?
    - Easier data representation?

- Now:
  - Growing the table when it gets too full (“rehashing”)
  - Relation between hashing/comparing and connection to Java

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

More on rehashing

- What if we copy all data to the same indices in the new table?
  - Will not work; we calculated the index based on **`TableSize`**

- Go through table, do standard insert for each into new table
  - Run-time?
  - *O*(n): Iterate through old table

- Resize is an *O*(*n*) operation, involving *n* calls to the hash function
  - Is there some way to avoid all those hash function calls?
  - Space/time tradeoff: Could store **`h(key)`** with each data item
  - Growing the table is still *O*(*n*); only helps by a constant factor

Equal Objects Must Hash the Same

- The Java library (and your project hash table) make a very important assumption that clients must satisfy…
- Object-oriented way of saying it:

  If **`a.equals(b)`**, then we must require **`a.hashCode()==b.hashCode()`**

- Function-object way of saying it:

  If **`c.compare(a,b) == 0`**, then we must require **`h.hash(a) == h.hash(b)`**

- Why is this essential?

Java bottom line

- Lots of Java libraries use hash tables, perhaps without your knowledge
- So: 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`**

By the way: comparison has rules too

We have not empahsized important “rules” about comparison for:

- All our dictionaries
- Sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all **`a`**, **`b`**, and **`c`**,

- If **`compare(a,b) < 0`**, then **`compare(b,a) > 0`**
- If **`compare(a,b) == 0`**, then **`compare(b,a) == 0`**
- If **`compare(a,b) < 0`** and **`compare(b,c) < 0`**,
  then **`compare(a,c) < 0`**

Final word on hashing

- The hash table is one of the most important data structures
  - Supports only **`find`**, **`insert`**, and **`delete`** efficiently

- Important to use a good hash function

- Important to keep hash table at a good size

- What we skipped: Perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing

- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums