CSE332: Data Abstractions

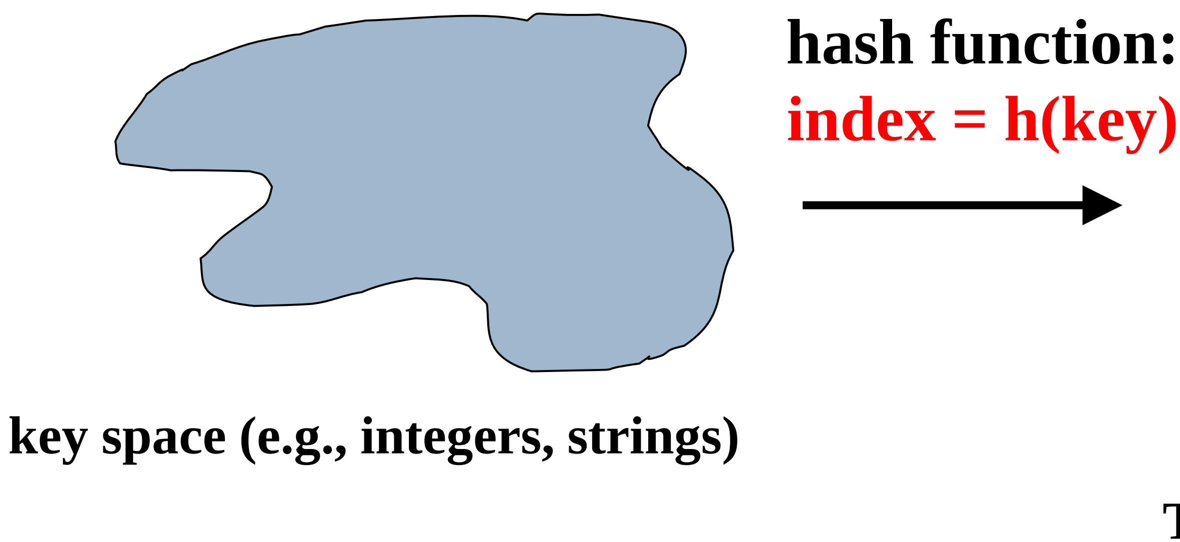
Lecture 11: Hash Tables

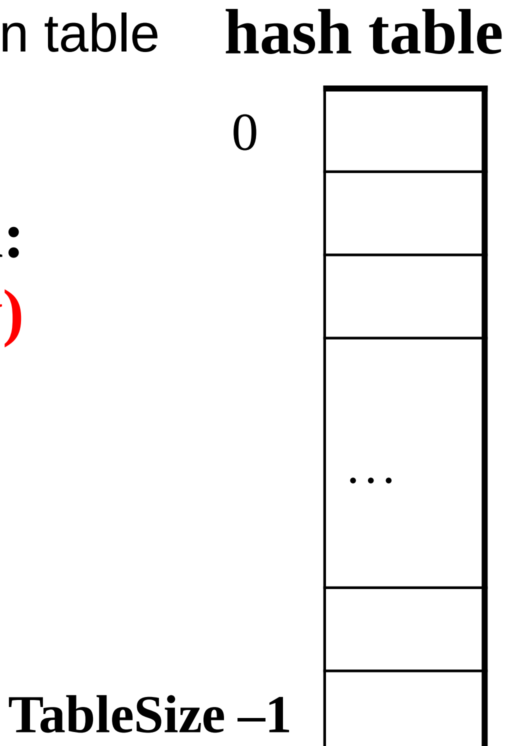
Tyler Robison

Summer 2010

Hash Table: Another dictionary

- Aim for constant-time (i.e., $O(1)$) **`find`**, **`insert`**, and **`delete`**
  - “On average” under some reasonable assumptions
- A hash table is an array of some fixed size
- Define a mapping from each key to a location in table
- Basic idea:





Hash tables

- There are m possible keys (m typically large, even infinite) but we expect our table to have only n items where n is much less than m (often written $n << m$)

Many dictionaries have this property

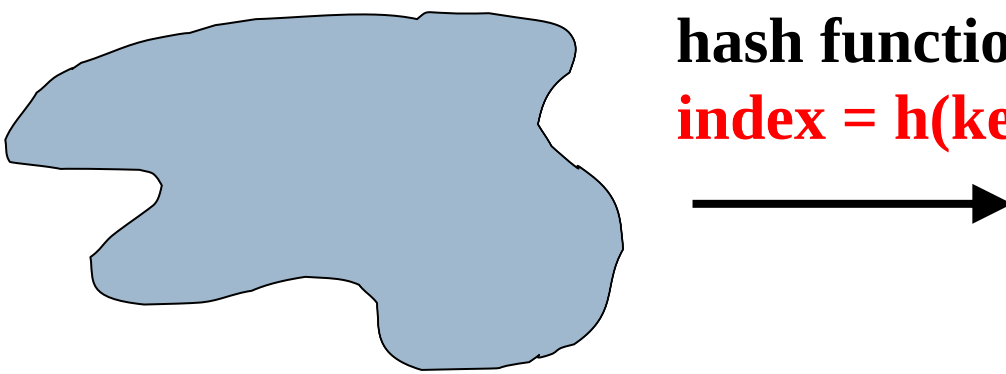
- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible student names vs. students enrolled
- AI: All possible chess-board configurations vs. those considered by the current player

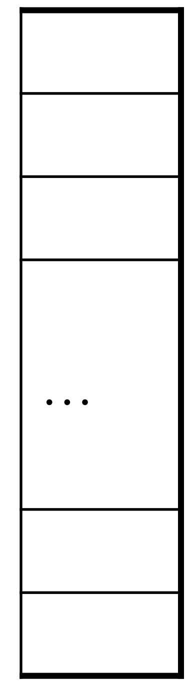
Hash functions

Hash function: Our key to index mapping

An ideal hash function:

- Is fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* a bit later

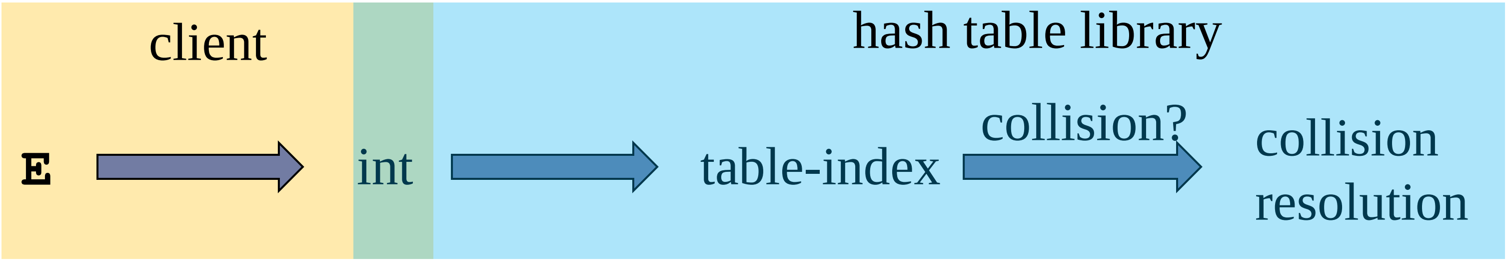




Who hashes what?

- Hash tables can be generic
  - To store elements of type **`E`**, we just need **`E`** to be:
    1. Comparable: order any two **`E`** (like with all dictionaries)
    2. Hashable: convert any **`E`** to an **`int`**

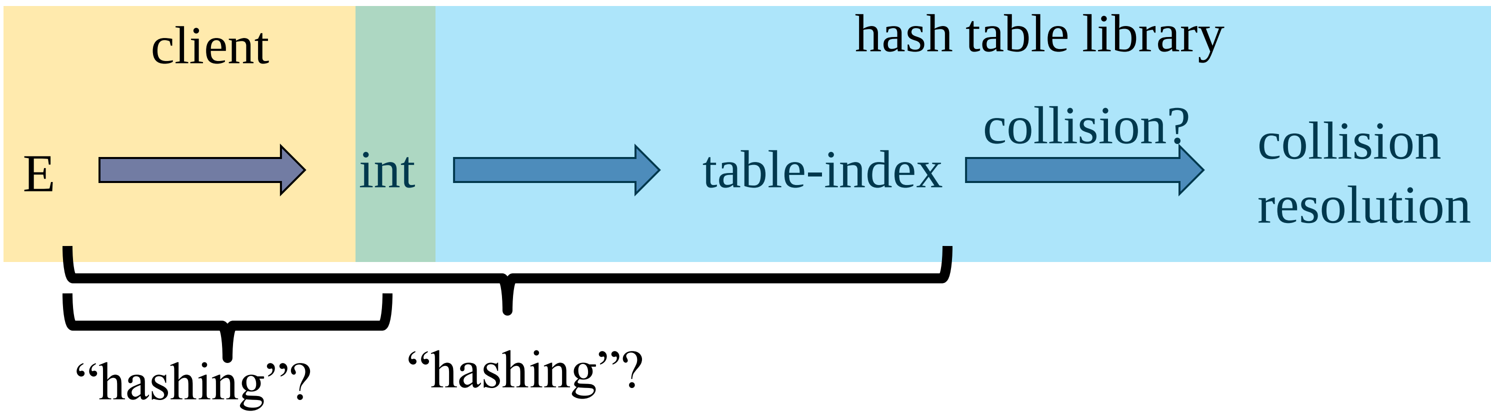
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- We will learn both roles, but most programmers “in the real world” spend more time on the client side, while still having an understanding of the library

More on roles

Some ambiguity in terminology on which parts are "hashing"



Two roles must both contribute to minimizing collisions

- Client should aim for different ints for expected items
  - Avoid "wasting" any part of **`E`** or the 32 bits of the **`int`**
- Library should aim for putting "similar" **`int`**s in different indices
  - conversion to index is almost always "mod table-size"
  - using prime numbers for table-size is common

What to hash?

In lecture we will consider the two most common things to hash: integers and strings

- If you have objects with several fields, it is usually best to have most of the "identifying fields" contribute to the hash to avoid collisions

- Example:
```
class Person {
   String first; String middle; String last;
   int age;
}
```

Hashing integers

- key space = integers
  - Useful for examples
- Simple hash function:

  `h(key) = key % TableSize`
  - Client: `f(x) = x`
  - Library `g(x) = x % TableSize`
  - Fairly fast and natural
- Example:
  - TableSize = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data "along for the ride")
  - What could go wrong?
    - Now insert 20….

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Collision-avoidance

- Collision: Two keys map to the same index
- With "**`x % TableSize`**" the number of collisions depends on
  - the ints inserted
  - **`TableSize`**
- Larger table-size tends to help, but not always
  - Example: Insert 12, 22, 32 with **`TableSize`** = 10 vs. **`TableSize`** = 6
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern, and "multiples of 61" are probably less likely than "multiples of 60"
  - Later we'll see that one collision-handling strategy does provably better with prime table size
  - Usually use something like 10 for examples though

0	12
1	
2	32
3	
4	22
5	

More arguments for a prime table size

If **TableSize** is 60 and…

- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table
- Lots of data items are 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

In general, if **x** and **y** are "co-prime" (means **gcd(x,y)==1**), then

(a * x) % y == (b * x) % y if and only if **a % y == b % y**

- So, given table size y and keys as multiples of x, we'll get a decent distribution if x & y are co-prime
- Good to have a **TableSize** that has not common factors with any "likely pattern" **x**

What if we don't have ints as keys?

- If keys aren't **`int`**s, the client must convert to an **`int`**
  - Trade-off: speed and distinct keys hashing to distinct **`int`**s
- Very important example: Strings
  - Key space K $= s_0 s_1 s_2 \ldots s_{m-1}$
    - Where s_i are chars: $s_i \in [0,51]$ or $s_i \in [0,255]$ or $s_i \in [0,2^{16}-1]$
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0$ % TableSize

   Anything w/ same first letter

2. $h(K) = \left(\sum_{i=0}^{m-1} s_i \right)$ % TableSize

   Any rearrangement of letters

3. $h(K) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right)$ % TableSize

   Hmm… not so clear

What causes collisions for each?

Java-esque String Hash

- Java characters in Unicode format; 2^{16} bits

$$h = s[0] * 31^{n-1} + s[1] * 31^{n-2} + \cdots + s[n-1]$$

- Can compute efficiently via a trick called Horner's Rule:
  - Idea: Avoid expensive computation of 31^k
  - Say n=4
  - h=((s[0]*31+s[1])*31+s[2])*31+s[3]

Specializing hash functions

How might you hash differently if all your strings were web addresses (URLs)?

Combining hash functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. When smashing two hashes into one hash, use bitwise-xor
   - Problem with Bitwise AND?
     - Produces too many 0 bits
   - Problem with Bitwise OR?
     - Produces too many 1 bits
3. Rely on expertise of others; consult books and other resources
4. If keys are known ahead of time, choose a *perfect hash*

Additional operations

- How would we do the following in a hashtable?
  - findMin()
  - findMax()
  - predecessor(key)
- Hashtables really not set up for these; need to search everything, O(n) time
- Could try a hack:
  - Separately store max & min values; update on insert & delete
  - What about ‘2^{nd} to max value’, predecessor, in-order traversal, etc; those are fast in an AVL tree

Hash Tables: A Different ADT?

- In terms of a Dictionary ADT for just **`insert`**, **`find`**, **`delete`**, hash tables and balanced trees are just different data structures
  - Hash tables $O(1)$ on average (*assuming* few collisions)
  - Balanced trees $O(\texttt{log}\ n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (collisions)
  - Yes, but **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`** go from $O(\texttt{log}\ n)$ to $O(n)$
    - Why your textbook considers this to be a different ADT
    - Not so important to argue over the definitions

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 we can resolve collisions in a couple of different ways:

- Separate chaining
- Open addressing

Separate Chaining

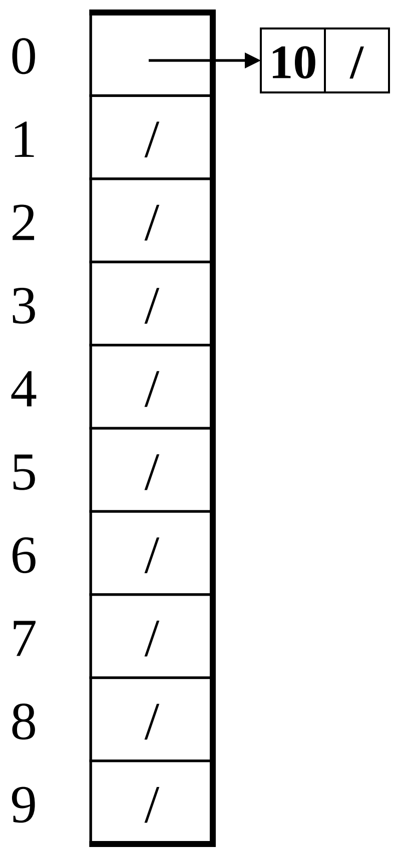
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

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

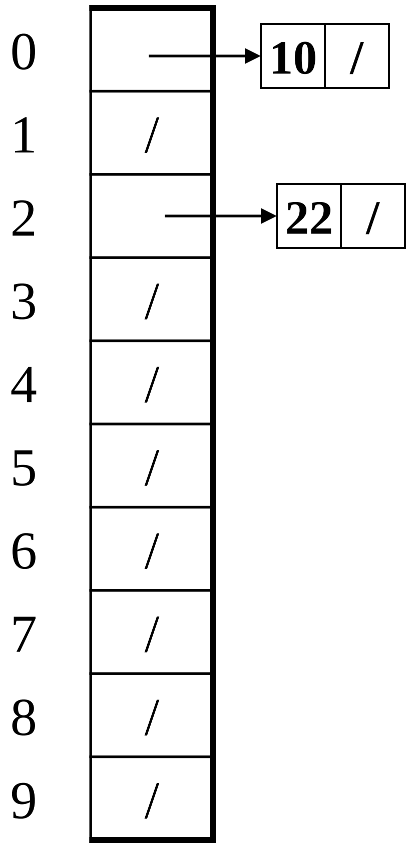


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

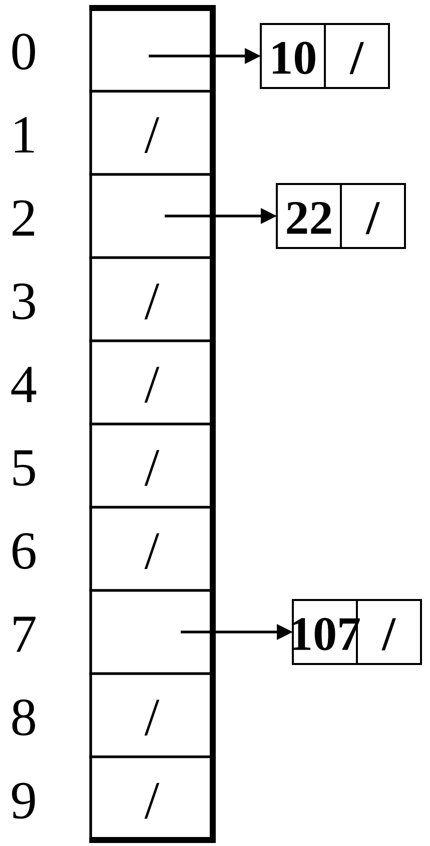


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

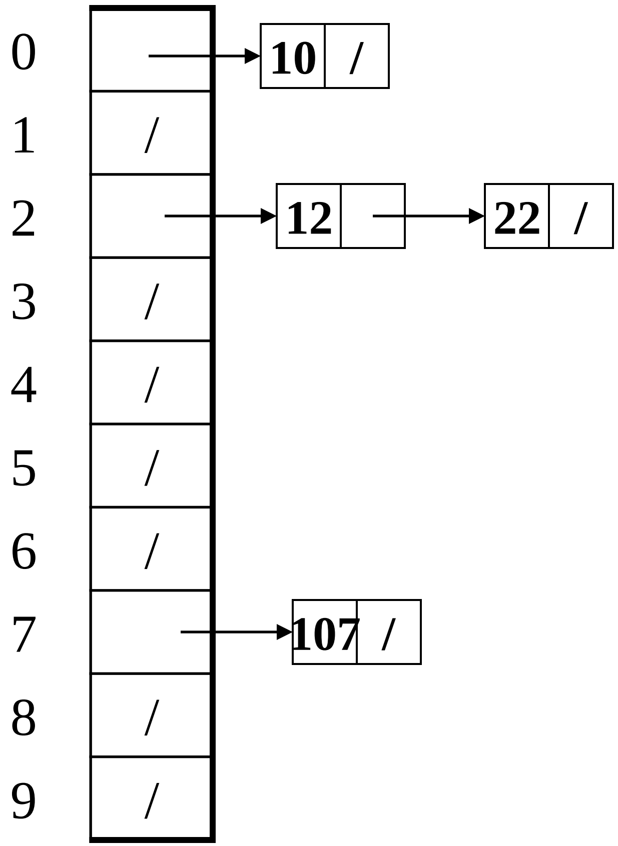


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

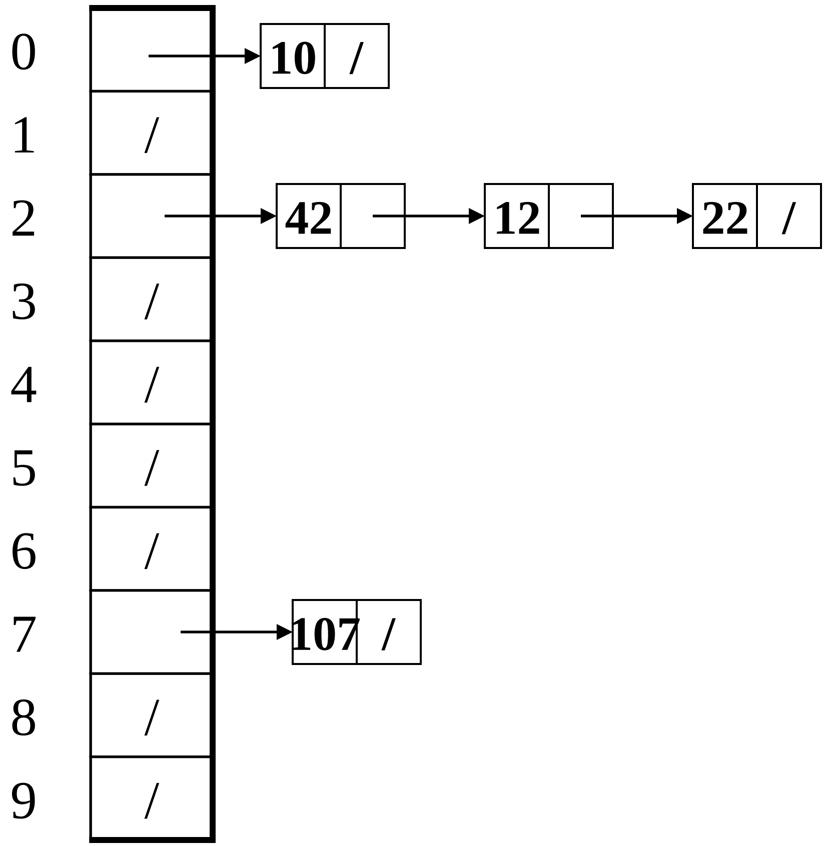


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

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

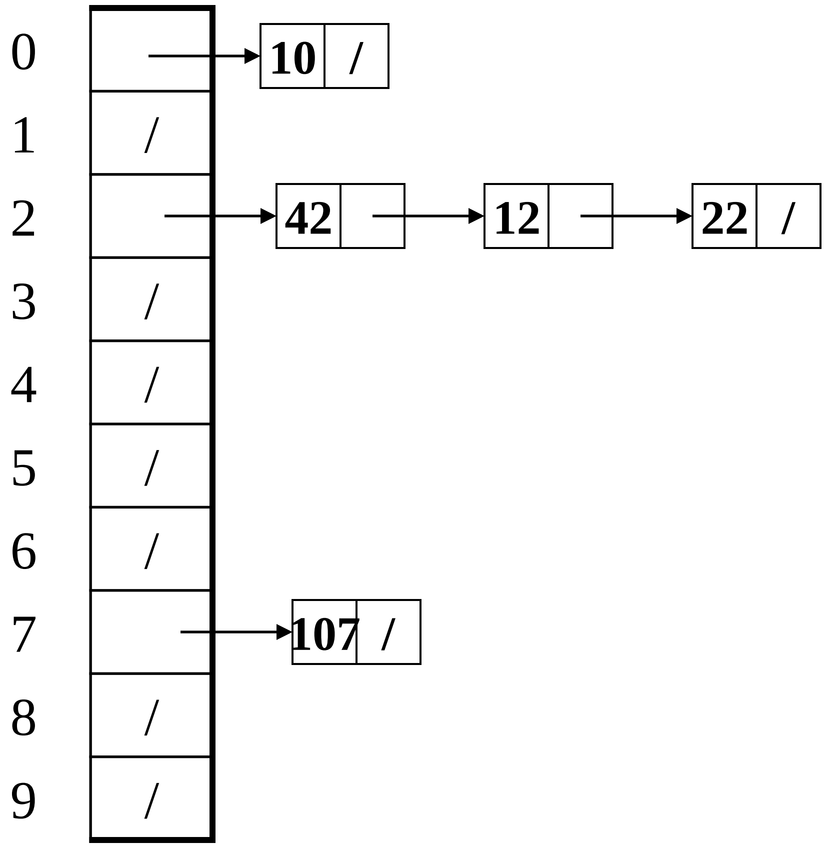
Why put them at the front?

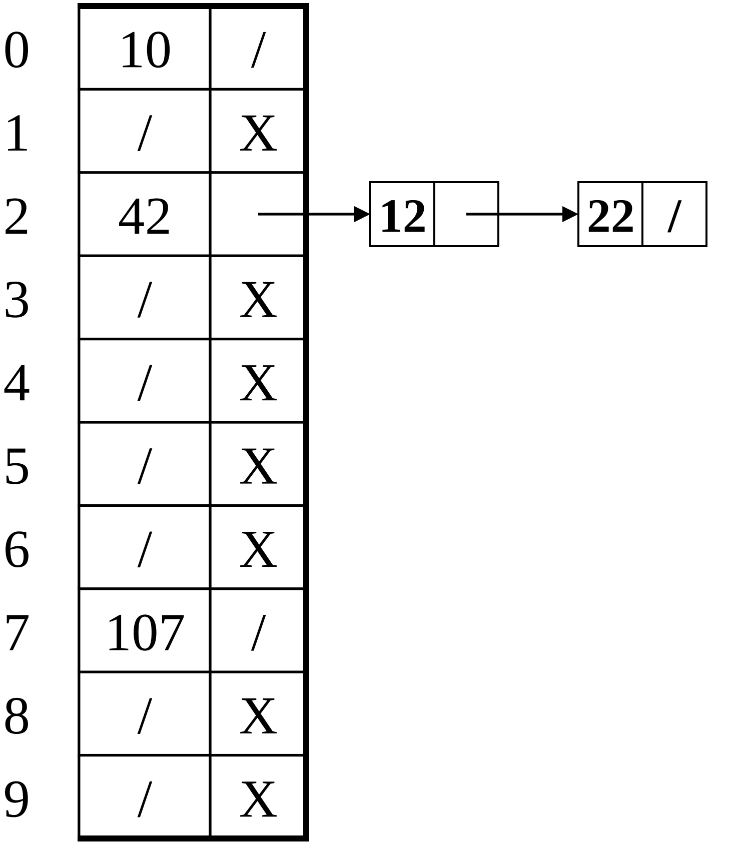
Handling duplicates?

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)
    - Keep # of items in each bucket small
    - Overhead of AVL tree, etc. not worth it for small n
- Beyond asymptotic complexity, some "data-structure engineering" may be warranted
  - Linked list vs. array or a hybrid of the two
  - Move-to-front (part of Project 2)
  - 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)





A more rigorous chaining analysis

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

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

N=number of elements

Under separate chaining, the average number of elements per bucket is…?

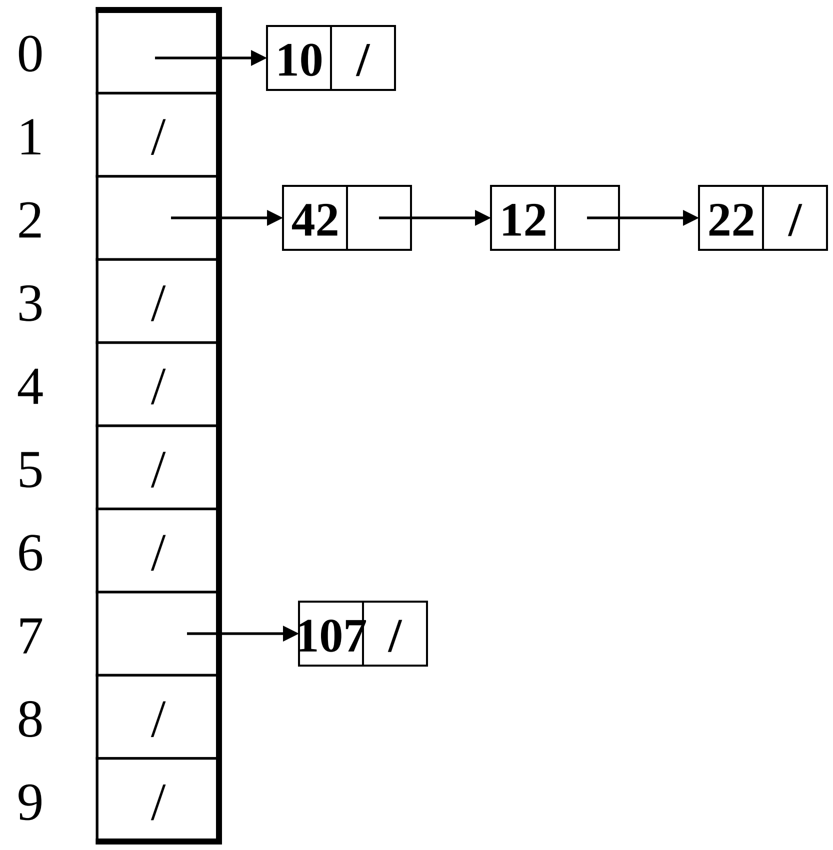
λ

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

- Each unsuccessful **`find`** compares against __λ__ items
- Each successful **`find`** compares against __$\lambda/2$__ items
- If λ is low, find & insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket
- Say, delete 12
- Similar run-time as insert



An Alternative to Separate Chaining: Open Addressing

- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If `h(key)` is already full,
  - Try `(h(key) + 1) % TableSize`
- **`That's full too?`**
  - Try `(h(key) + 2) % TableSize`
- **`How about`**
  - Try `(h(key) + 3) % TableSize`
- …
- Example: insert 38, 19, 8, 109, 10

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

An Alternative to Separate Chaining: Open Addressing

- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If **`h(key)`** is already full,
  - Try `(h(key) + 1) % TableSize`
- **`That's full too?`**
  - Try `(h(key) + 2) % TableSize`
- **`How about`**
  - Try `(h(key) + 3) % TableSize`
- …
- Example: insert 38, 19, 8, 109, 10

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

An Alternative to Separate Chaining: Open Addressing

- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If **`h(key)`** is already full,
  - Try **`(h(key) + 1) % TableSize`**
- **`That's full too?`**
  - Try **`(h(key) + 2) % TableSize`**
- **`How about`**
  - Try **`(h(key) + 3) % TableSize`**
- …
- Example: insert 38, 19, 8, 109, 10

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

An Alternative to Separate Chaining: Open Addressing

- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If **`h(key)`** is already full,
  - Try **`(h(key) + 1) % TableSize`**
- **`That's full too?`**
  - Try **`(h(key) + 2) % TableSize`**
- **`How about`**
  - Try **`(h(key) + 3) % TableSize`**
- …
- Example: insert 38, 19, 8, 109, 10

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

An Alternative to Separate Chaining: Open Addressing

- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If `h(key)` is already full,
  - Try `(h(key) + 1) % TableSize`
- **`That's full too?`**
  - Try `(h(key) + 2) % TableSize`
- **`How about`**
  - Try `(h(key) + 3) % TableSize`
- …
- Example: insert 38, 19, 8, 109, 10

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

Open addressing: Storing in the table

- This is *one example* of open addressing
  - More generally, we just need to describe where to check next when one attempt fails (cell already in use)
  - Each version of open addressing involves specifying a sequence of indices to try
- Trying the next spot is called probing
  - In the above example, our **`i`**th probe was **`(h(key) + i) % TableSize`**
    - To get the next index to try, we just added 1 (mod the Tablesize)
    - This is called linear probing
  - More generally we have some probe function **`f`** and use

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

    for the ith probe (start at i=0)
    - **`For linear probing, f(i)=i`**

More about Open Addressing

- Find works similarly:
  - Keep probing until we find it
  - Or, if we hit null, we know it's not in the table
- How does open addressing work with high load factor (***λ***)
  - Poorly
  - Too many probes means no more *O*(1)
  - So want larger tables
  - Find with *λ*=1?
- Deletion? How about we just remove it?
  - Take previous example, delete 38
  - Then do a find on 8
  - Hmm… this isn't going to work
  - Stick with lazy deletion

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

Terminology

We and the book use the terms

- “chaining” or “separate chaining”: Linked list in each bucket

    vs.

- “open addressing”: Store directly in table

Very confusingly,

- “open hashing” is a synonym for “chaining”

    vs.

- “closed hashing” is a synonym for “open addressing”

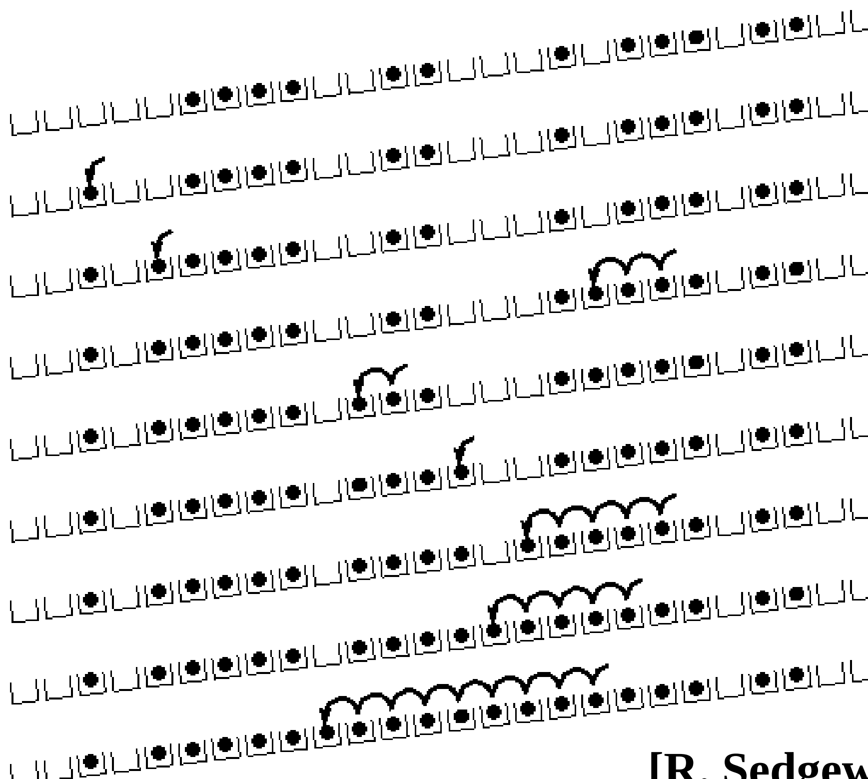
Primary Clustering

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

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

Saw this happening in earlier example

- Called primary clustering



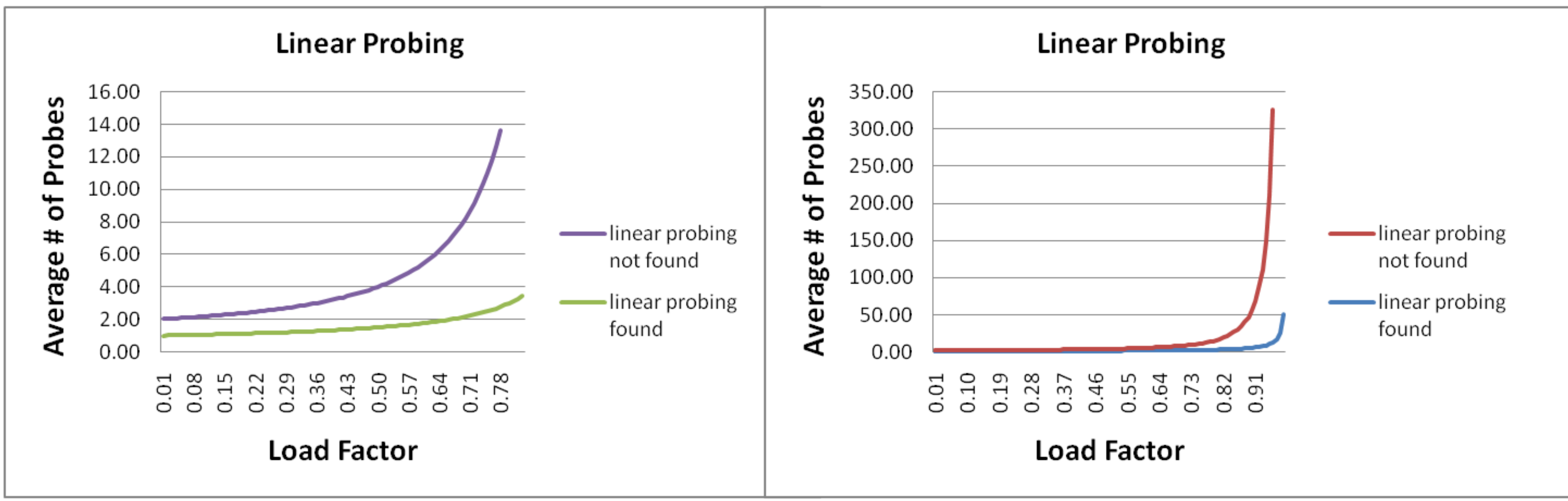
[R. Sedgewick]

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



Open Addressing: Quadratic probing

- We can avoid primary clustering by changing the probe function

- 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

How about 98?

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

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)**

Uh-oh: For all *n*, **`((n*n) +5) % 7 is 0, 2, 5, or 6`**

- 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

- For all *c* and *k*, `(n²+c) % k = ((n-k)²+c) % k`
- The bad news is: After **`TableSize`** quadratic probes, we will just cycle through the same indices
- The good news:
  - Assertion #1: If **T = TableSize** is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most **T/2** probes
  - Assertion #2: For prime **T** and $\mathbf{0 \leq i,j \leq T/2}$ where $\mathbf{i \neq j}$,

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

    That is, if T is prime, the first T/2 quadratic probes map to different locations
  - Assertion #3: Assertion #2 is the "key fact" for proving Assertion #1
- So: If you keep $\lambda < ½$, no need to detect cycles

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: quadratic nature quickly escapes the neighborhood

- But it's no help if keys initially hash to the same index
  - Called secondary clustering
  - Any 2 keys that hash to the same value will have the same series of moves after that

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

Open Addressing: Double hashing

- Idea:
  - Given two good hash functions h and g & 2 different keys k1 & k2, it is very unlikely that **`h(k1)==h(k2) & g(k1)==g(k2)`**
  - So make the probe function **`f(i) = i*g(key)`**
  - That is, check h(key), then h(key)+g(key), then h(key)+2*g(key), …
  - Even if h(key1)=h(key2), they'll most likely go different places for the next probe
- Probe sequence:
  - 0th probe: **`h(key) % TableSize`**
  - 1st probe: **`(h(key) + g(key)) % TableSize`**
  - 2nd probe: **`(h(key) + 2*g(key)) % TableSize`**
  - 3rd probe: **`(h(key) + 3*g(key)) % TableSize`**
  - …
  - ith probe: **`(h(key) + i*g(key)) % TableSize`**
- Detail: Make sure **`g(key)`** `isn't` **0**
  - **`Why?`**
  - **`Also, shouldn't be a multiple of TableSize`**

Double-hashing analysis

- Intuition: Since each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"
  - Say h(x)==h(y); it's unlikely that g(x)==g(y)

- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - No guarantee that i*g(key) will let us try all/most indices
  - It is known that this infinite loop, despite space available, cannot happen in at least one case:
    - **`h(key) = key % p`**
    - **`g(key) = q - (key % q)`**
    - **`2 < q < p`**
    - **`p`** and **`q`** are prime

Yet another reason to use a prime Tablesize

- So, for double hashing

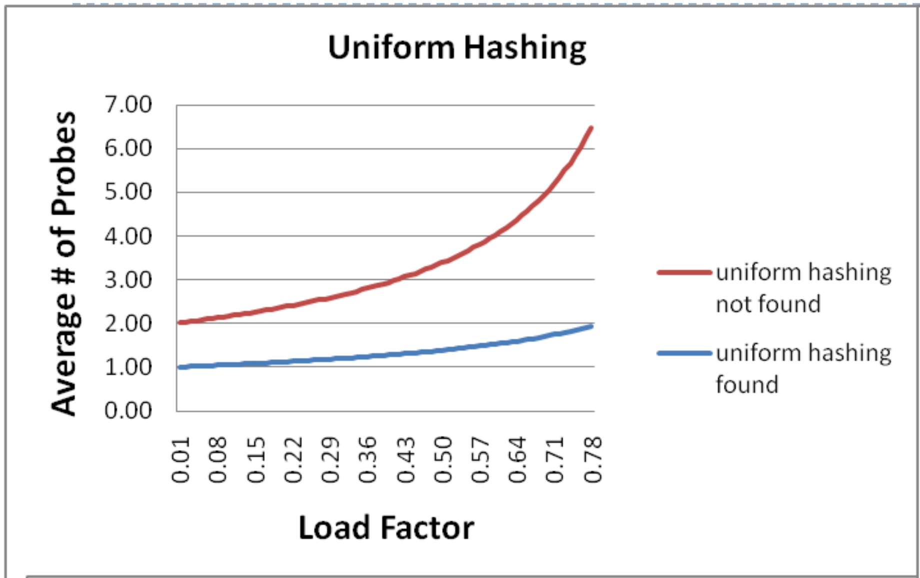
  i^{th} probe: `(h(key) + i*g(key))% TableSize`
- Say g(key) divides Tablesize
  - That is, there is some integer x such that x*g(key)=Tablesize
  - After x probes, we'll be back to trying the same indices as before
- Ex:
  - Tablesize=50
  - g(key)=25
  - Probing sequence:
    - h(key)
    - h(key)+25
    - h(key)+50=h(key)
    - h(key)+75=h(key)+25
- Only 1 & itself divide a prime

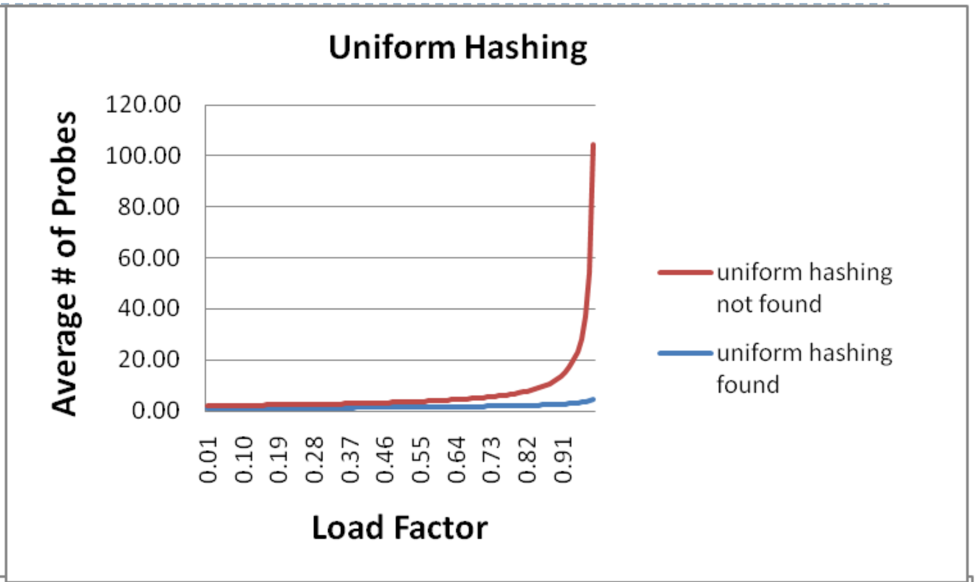
More double-hashing facts

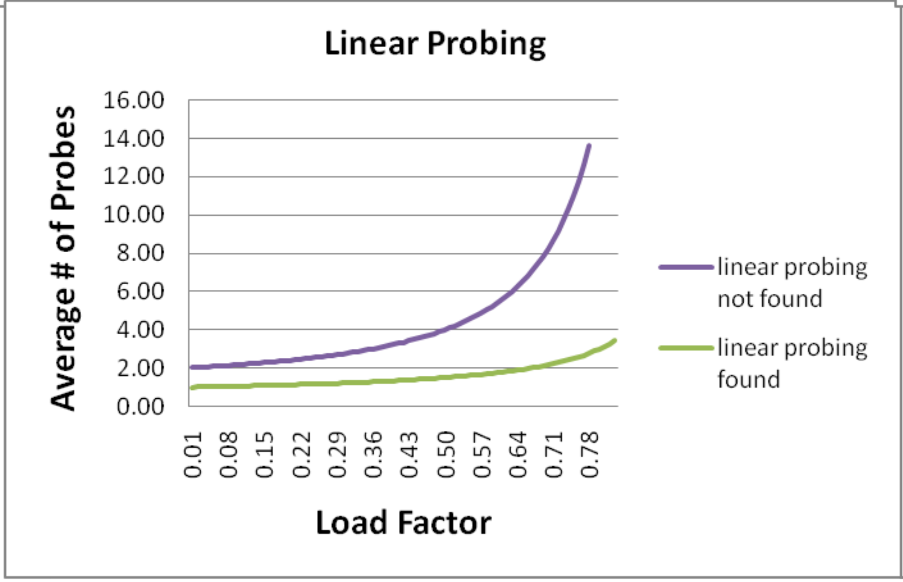
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`
- Non-trivial facts we won't prove:

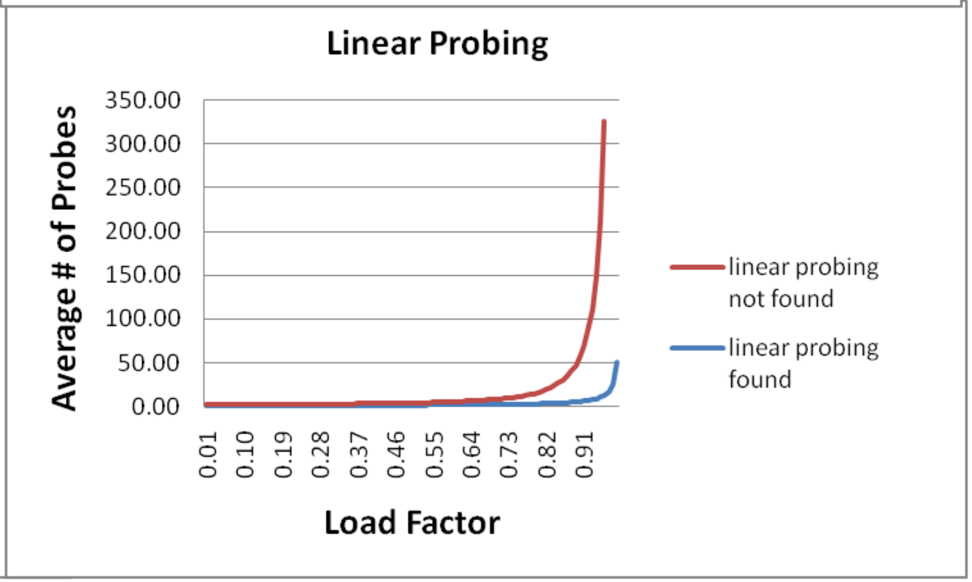
  Average # of probes given λ (in the limit as **`TableSize`** $\to\infty$)
  - Unsuccessful search (intuitive): $$\frac{1}{1-\lambda}$$
  - Successful search (less intuitive): $$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$
- Bottom line: unsuccessful bad (but not as bad as linear probing),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

Charts: Double hashing (w/ uniform hashing) vs. Linear probing









We've explored different methods of collision detection

- Chaining is easy
  - `find`, `delete` proportion to load factor on average; insert constant
- Open addressing uses probe functions, has clustering issues as table gets full
  - Why use it:
    - Less memory allocation
    - Some run-time overhead for allocating linked list (or whatever) nodes; open addressing could be faster
    - Arguably easier data representation
- Now:
  - Growing the table when it gets too full: Called 'rehashing'
  - Relation between hashing/comparing and connection to Java

Rehashing

- Like with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over
- 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…
    - 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
  - If you do need more primes, not too bad to calculate

More on rehashing

- What if we copy all data to the same indices in the new table?
  - Not going to work; calculated index based on TableSize – we may not be able to find it later
- Go through current table, do standard insert for each into new table; run-time?
  - O(n): Iterate through table

- But resize is an $O(n)$ operation, involving n calls to the hash function (1 for each insert in the new table)
  - Is there some way to avoid all those hash function calls again?
  - Space/time tradeoff: Could store **`h(key)`** with each data item, but since rehashing is rare, this is probably a poor use of space
    - And growing the table is still $O(n)$; only helps by a constant factor

Hashing and comparing

- For insert/find, as we go through the chain or keep probing, we have to *compare* each item we see to the key we're looking for
  - We need to have a comparator (or key's type needs to be comparable)
  - Don't actually need < & >; just =
- So a hash table needs a hash function and a comparator
  - In Project 2, you'll use two function objects
  - The Java standard library uses a more OO approach where each object has an **`equals`** method and a **`hashCode`** method:

```
class Object {
  boolean equals(Object o) {…}
  int hashCode() {…}
  …
}
```

Equal objects must hash the same

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

- OO 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)`

- What would happen if we didn't do this?

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

(Incorrect) Example

- Think about using a hash table holding points

```
class PolarPoint {
  double r = 0.0;
  double theta = 0.0;
  void addToAngle(double theta2) { theta+=theta2; }
  …
  boolean equals(Object otherObject) {
     if(this==otherObject) return true;
     if(otherObject==null) return false;
     if(getClass()!=other.getClass()) return false;
     PolarPoint other = (PolarPoint)otherObject;
     double angleDiff =
         (theta - other.theta) % (2*Math.PI);
     double rDiff = r - other.r;
     return Math.abs(angleDiff) < 0.0001
            && Math.abs(rDiff) < 0.0001;
  }
  // wrong: must override hashCode!
}
```

Aside: Comparable/Comparator have rules too

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`**

What would happen if compareTo() just randomly returned -1, 0 or 1?

Final word on hashing

- The hash table is one of the most important data structures
  - Supports only `find`, `insert`, and `delete` efficiently
  - FindMin, FindMax, predecessor, etc.: not so efficiently
  - Most likely data-structure to be asked about in interviews; many real-world applications

- Important to use a good hash function
  - Good distribution
  - Uses enough of key's values

- Important to keep hash table at a good size
  - Prime #
  - Preferable λ depends on type of table

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