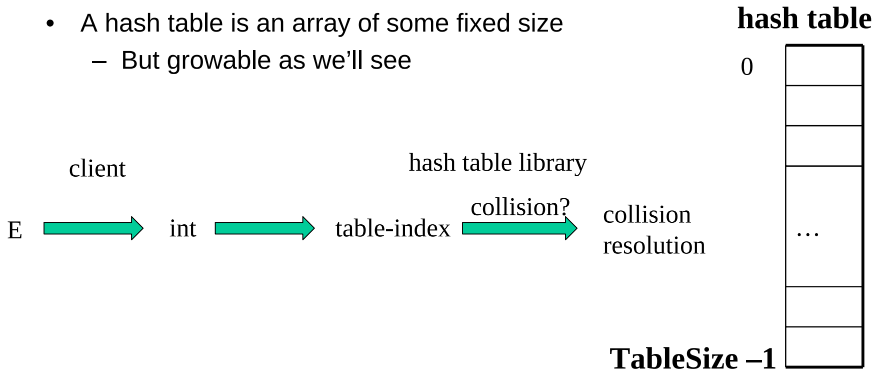
CSE332: Data Abstractions

Lecture 11: Hash Tables

Dan Grossman
Spring 2010

Hash Tables: Review

- 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
  - But growable as we'll see



hash table

0

client

hash table library

E → int → table-index $\xrightarrow{\text{collision?}}$ collision resolution

…

TableSize –1

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*(**`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*(**`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:

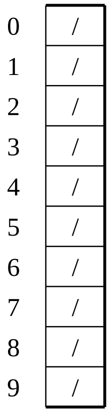
When two keys map to the same location in the hash table

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

So hash tables should support collision resolution

- Ideas?

Separate Chaining

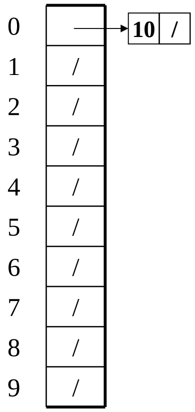


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

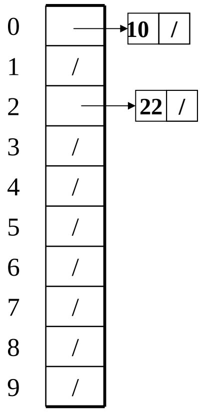


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

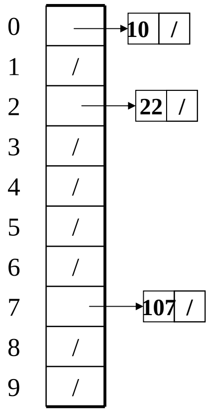


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

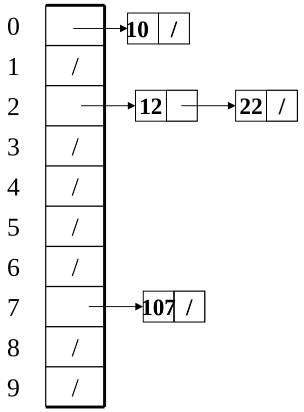


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

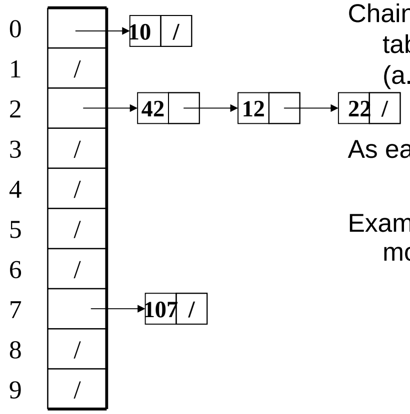


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

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”)

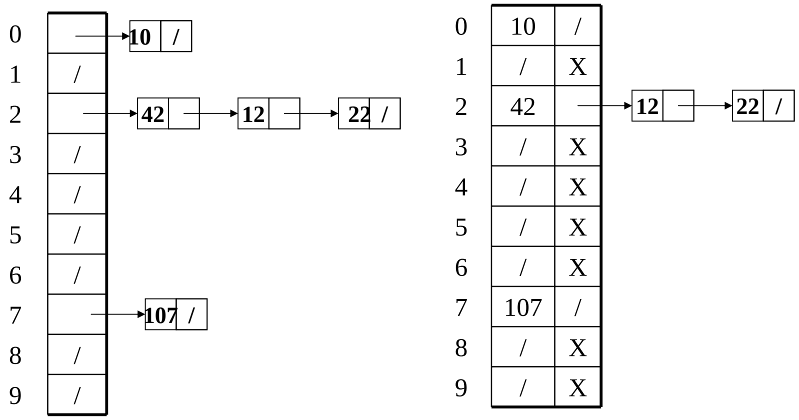
As easy as it sounds

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!)
  - Move-to-front (cf. Project 2)
  - Better idea: 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

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

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 ____ items
- Each successful **`find`** compares against _____ items

More rigorous chaining analysis

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

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

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
- Each successful **`find`** compares against $\lambda / 2$ items

---

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
- Our `i`th probe was `(h(key) + i) % TableSize`
  - This is called linear probing
- 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)

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

Okay, so **`insert`** finds an open table position using a probe function

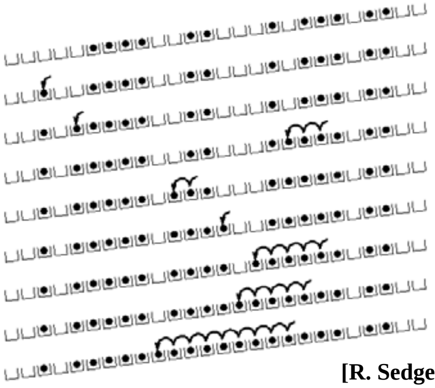
What about **`find`**?
- Must use same probe function to “retrace the trail” and find the data
- Unsuccessful search when reach empty position

What about **`delete`**?
- ***Must*** use “lazy” deletion. Why?
- But here just means “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 (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

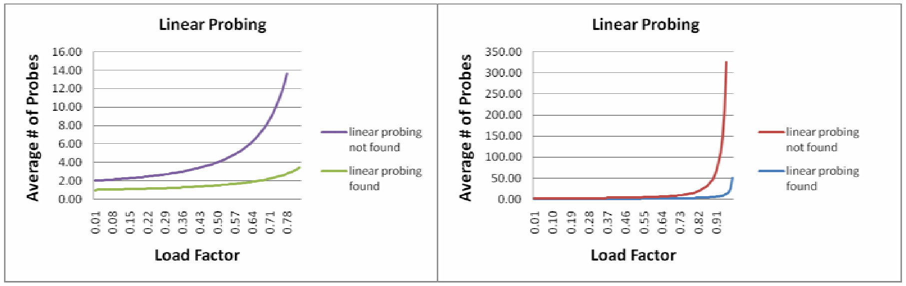
- Trivial fact: For any $\lambda < 1$, linear probing will find an empty slot
  - It is “safe” in this sense: no infinite loop unless table is full

- 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 (see chart)

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

- 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)
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`

- 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

- 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 $0 \leq i,j \leq T/2$ where $i \neq j$,
    $$(h(key) + i^2) \% T \neq (h(key) + j^2) \% T$$
  - 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: 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)` can’t be `0`

Double-hashing analysis

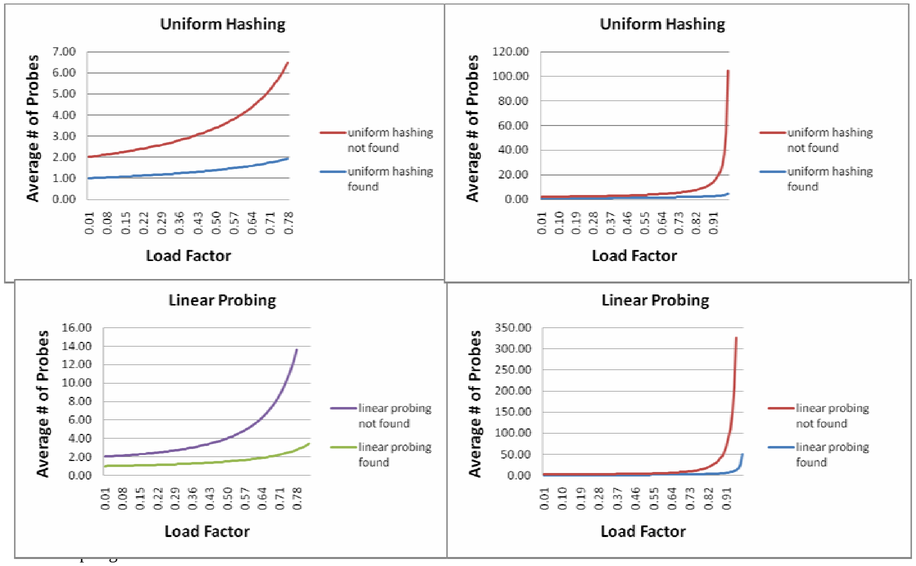
- Intuition: Since each probe is “jumping” by `g(key)` each time, we “leave the neighborhood” *and* “go different places from other initial collisions”
- But we could still have a problem like in quadratic probing where we are not “safe” (infinite loop despite room in table)
  - It is known that this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

Charts



Where are we?

- Chaining is easy
  - **`insert`**, **`find`**, **`delete`** proportion to load factor on average
- Open addressing uses probe functions, has clustering issues as table gets full
  - Why use it:
    - Less memory allocation?
    - Easier data representation?
- Now:
  - Growing the table when it gets too full
  - 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
- Especially 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

- We double the size (rather than “add 1000”) to get good amortized guarantees (still promising to prove that later ☺)
- But one resize is an $O(n)$ operation, involving n calls to the hash function (1 for each insert in the new table)
- 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 $O(n)$

Hashing and comparing

- Haven’t emphasized enough for a find or a delete of an item of type **`E`**, we *hash* **`E`**, but then as we go through the chain or keep probing, we have to *compare* each item we see to **`E`**.

- 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 i:

  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`**

Bad 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!
}
```

By the way: comparison has rules too

We didn't emphasize some important "rules" about comparison functions for:

- all our dictionaries
- sorting (next major topic)

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