CSE332: Data Abstractions

Lecture 9: Hashing

James Fogarty

Winter 2012
Administrative

• Midterm Review Poll

• Project 2a Due Wednesday

• Homework 4 Due Friday

• Feedback Plans
Homework 2, Problem 2

Need to percolate down

Also must percolate up
Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
  - If \( h(key) \) is already full,
    - try \( (h(key) + 1) \mod TableSize \). If full,
    - try \( (h(key) + 2) \mod TableSize \). If full,
    - try \( (h(key) + 3) \mod TableSize \). If full...
- Example: insert 38, 19, 8, 109, 10
Open Addressing: Linear Probing

• Why not use up the empty space in the table?

• Store directly in the array cell (no linked list)

• How to deal with collisions?

• If \( h(\text{key}) \) is already full,
  - try \( (h(\text{key}) + 1) \mod \text{TableSize} \). If full,
  - try \( (h(\text{key}) + 2) \mod \text{TableSize} \). If full,
  - try \( (h(\text{key}) + 3) \mod \text{TableSize} \). If full...

• Example: insert 38, 19, 8, 109, 10
Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
  - If \( h(\text{key}) \) is already full,
    - try \( (h(\text{key}) + 1) \mod \text{TableSize} \). If full,
    - try \( (h(\text{key}) + 2) \mod \text{TableSize} \). If full,
    - try \( (h(\text{key}) + 3) \mod \text{TableSize} \). If full...
- Example: insert 38, 19, 8, 109, 10
Open Addressing: Linear Probing

• Why not use up the empty space in the table?
• Store directly in the array cell (no linked list)
• How to deal with collisions?

• If $h(key)$ is already full,
  – try $(h(key) + 1) \mod \text{TableSize}$. If full,
  – try $(h(key) + 2) \mod \text{TableSize}$. If full,
  – try $(h(key) + 3) \mod \text{TableSize}$. If full…

• Example: insert 38, 19, 8, 109, 10
### Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?

- If \( h(key) \) is already full,
  - try \((h(key) + 1) \mod TableSize\). If full,
  - try \((h(key) + 2) \mod TableSize\). If full,
  - try \((h(key) + 3) \mod TableSize\). If full...

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>/</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
</tr>
<tr>
<td>4</td>
<td>/</td>
</tr>
<tr>
<td>5</td>
<td>/</td>
</tr>
<tr>
<td>6</td>
<td>/</td>
</tr>
<tr>
<td>7</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
</tr>
</tbody>
</table>
**Open Addressing: Linear Probing**

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
  - If \( h(key) \) is already full,
    - try \( (h(key) + 1) \mod TableSize \). If full,
    - try \( (h(key) + 2) \mod TableSize \). If full,
    - try \( (h(key) + 3) \mod TableSize \). If full...
- Example: insert 38, 19, 8, 109, 10
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**
  \[ h(key) + i \mod TableSize \]
- In general have some **probe function** \( f \) and use
  \[ h(key) + f(i) \mod TableSize \]

Open addressing does poorly with high load factor \( \lambda \)

- So we want larger tables
- Too many probes means we lose our \( O(1) \)
**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”

We also do trees 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”

```
| 10 | ✗ | / | 23 | / | / | 16 | ✗ | 26 |
```
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 probe 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 $\lambda$ (in the limit as $\text{TableSize} \to \infty$)
  – 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 (let’s look at a chart)
Analysis in Chart Form

- Linear-probing performance degrades rapidly as table gets full
  - Formula assumes “large table” but point remains

- Chaining performance was linear in $\lambda$ and has no trouble with $\lambda > 1$
Open Addressing: Quadratic Probing

- We can avoid primary clustering by changing the probe function

\[(h(key) + f(i)) \mod \text{TableSize}\]

- For quadratic probing:
  \[f(i) = i^2\]

- So probe sequence is:
  - 0\(^{th}\) probe: \(h(key) \mod \text{TableSize}\)
  - 1\(^{st}\) probe: \((h(key) + 1) \mod \text{TableSize}\)
  - 2\(^{nd}\) probe: \((h(key) + 4) \mod \text{TableSize}\)
  - 3\(^{rd}\) probe: \((h(key) + 9) \mod \text{TableSize}\)
  - ...
  - \(i^{th}\) probe: \((h(key) + i^2) \mod \text{TableSize}\)

- Intuition: Probes quickly “leave the neighborhood”
**Quadratic Probing Example**

TableSize=10

Insert:
- 89
- 18
- 49
- 58
- 79
Quadratic Probing Example

Table Size = 10

Insert:
89
18
49
58
79
### Quadratic Probing Example

Table Size = 10

Insert:
- 89
- 18
- 49
- 58
- 79
### Quadratic Probing Example

<table>
<thead>
<tr>
<th>TableSize=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert:</td>
</tr>
<tr>
<td>89</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td>79</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
</tbody>
</table>
**Quadratic Probing Example**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
</tr>
</tbody>
</table>

Table Size = 10

Insert:
89
18
49
58
79
Quadratic Probing Example

Table Size = 10

Insert:
89
18
49
58
79
Another Quadratic Probing Example

Table Size = 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

TableSize = 7

Insert:
76  \hspace{1cm} (76 \% 7 = 6)
40  \hspace{1cm} (40 \% 7 = 5)
48  \hspace{1cm} (48 \% 7 = 6)
5   \hspace{1cm} (  5 \% 7 = 5)
55  \hspace{1cm} (55 \% 7 = 6)
47  \hspace{1cm} (47 \% 7 = 5)
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

Table Size = 7

Insert:

<table>
<thead>
<tr>
<th>Insert</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>(76 % 7 = 6)</td>
</tr>
<tr>
<td>40</td>
<td>(40 % 7 = 5)</td>
</tr>
<tr>
<td>48</td>
<td>(48 % 7 = 6)</td>
</tr>
<tr>
<td>5</td>
<td>(5 % 7 = 5)</td>
</tr>
<tr>
<td>55</td>
<td>(55 % 7 = 6)</td>
</tr>
<tr>
<td>47</td>
<td>(47 % 7 = 5)</td>
</tr>
</tbody>
</table>
**Another Quadratic Probing Example**

Table Size = 7

<table>
<thead>
<tr>
<th>Insert</th>
<th>Index</th>
<th>Probe Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>55</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>47</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

(Insertion sequence and probe calculation: 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

Table Size = 7

Insert:
- 76 \hspace{1cm} (76 \mod 7 = 6)
- 40 \hspace{1cm} (40 \mod 7 = 5)
- 48 \hspace{1cm} (48 \mod 7 = 6)
- 5 \hspace{1cm} (5 \mod 7 = 5)
- 55 \hspace{1cm} (55 \mod 7 = 6)
- 47 \hspace{1cm} (47 \mod 7 = 5)
Another Quadratic Probing Example

Table Size $= 7$

Insert:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

Doh: For all $n$, $(5 + (n \times n)) \mod 7$ is 0, 2, 5, or 6

Proof uses induction and $(n^2 + 5) \mod 7 = ((n-7)^2 + 5) \mod 7$

In fact, for all $c$ and $k$, $(n^2 + c) \mod k = ((n-k)^2 + c) \mod k$
From Bad News to Good News

- After TableSize quadratic probes, we cycle through the same indices.

- The good news:
  - For prime $T$ and $0 \leq i, j \leq T/2$ where $i \neq j$,
    \[(h(key) + i^2) \mod T \neq (h(key) + j^2) \mod T\]
  - If $T = TableSize$ is prime and $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot in at most $T/2$ probes.
  - If you keep $\lambda < \frac{1}{2}$, 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*:
  - Any 2 keys that hash to the same value will have the same series of moves after that
  - Called *secondary clustering*.

- 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 $g$, it is very unlikely that for some key, $h(key) == g(key)$

$$(h(key) + f(i)) \mod \text{TableSize}$$

- For double hashing:
  $$f(i) = i \times g(key)$$

- So probe sequence is:
  - $0^{th}$ probe: $h(key) \mod \text{TableSize}$
  - $1^{st}$ probe: $(h(key) + g(key)) \mod \text{TableSize}$
  - $2^{nd}$ probe: $(h(key) + 2g(key)) \mod \text{TableSize}$
  - $3^{rd}$ probe: $(h(key) + 3g(key)) \mod \text{TableSize}$
  - ...
  - $i^{th}$ probe: $(h(key) + i \times g(key)) \mod \text{TableSize}$

- Detail: Must make sure that $g(key)$ cannot be 0
Double Hashing

T = 10 (TableSize)
Hash Functions:
\[ h(key) = key \mod T \]
\[ g(key) = 1 + ((key/T) \mod (T-1)) \]

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

13
28
33
147
43
**Double Hashing**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T = 10 (TableSize)

Hash Functions:
- \( h(key) = key \mod T \)
- \( g(key) = 1 + ((key/T) \mod (T-1)) \)

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

- 13
- 28
- 33
- 147
- 43
Double Hashing

T = 10 (TableSize)

Hash Functions:
\[ h(key) = key \mod T \]
\[ g(key) = 1 + ((key/T) \mod (T-1)) \]

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

13
28
33
147
43
**Double Hashing**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T = 10 (Table Size)

**Hash Functions:**

\[ h(key) = key \mod T \]

\[ g(key) = 1 + ((key/T) \mod (T-1)) \]

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

- 13
- 28
- 33
- 147
- 43
## Double Hashing

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T = 10 (TableSize)

### Hash Functions:
- $h(key) = key \mod T$
- $g(key) = 1 + ((key/T) \mod (T-1))$

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

- 13
- 28
- 33
- 147
- 43
Double Hashing

$$T = 10 \text{ (TableSize)}$$

Hash Functions:

$$h(key) = key \mod T$$

$$g(key) = 1 + ((key/T) \mod (T-1))$$

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

13
28
33
147
43

Doh:

$$3 + 0 = 3 \quad 3 + 15 = 18$$

$$3 + 5 = 8 \quad 3 + 20 = 23$$

$$3 + 10 = 13 \quad 3 + 25 = 28$$
Double Hashing Analysis

• Intuition:

  Because each probe is “jumping” by $g(key)$ each time, we should both “leave the neighborhood” and “go different places from the same initial collision”

• But, as in quadratic probing, we could still have a problem where we are not “safe” (infinite loop despite room in table)

• It is known that this cannot happen in at least one case:
  • $h(key) = key \mod p$
  • $g(key) = q - (key \mod q)$
  • $2 < q < p$
  • $p$ and $q$ are prime
Where are we?

- **Separate 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 it gets full
  - Why use it:
    - Less memory allocation?
    - Run-time overhead for list nodes; array could be faster?
    - Easier data representation?

- Now:
  - Growing the table when it gets too full (aka “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 that won’t be prime!
  – So go *about* twice-as-big
  – Can have a list of prime numbers in your code, since you probably will not grow more than 20-30 times, and can then calculate after that
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
Hashing and Comparing

• Our use of \texttt{int} key can lead to overlooking a critical detail
  – We initial \texttt{hash E},
  – While chaining or probing, we \texttt{compare} to \texttt{E}.
    • Just need equality testing (i.e., compare == 0)

• So a hash table needs a hash function and a comparator
  – In Project 2, you will use two function objects
  – The Java library uses a more object-oriented approach: each object has an \texttt{equals} method and a \texttt{hashCode} method:

```java
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

- Object-oriented way of saying it:
  
  ```java
  if a.equals(b), then we must require
  a.hashCode() == b.hashCode()
  ```

- Function object way of saying it:
  
  ```java
  if c.compare(a, b) == 0, then we must 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`
Comparable/Comparator Have Rules Too

We have not emphasized 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 \(\text{compare}(a, b) < 0,\) then \(\text{compare}(b, a) > 0\)
– If \(\text{compare}(a, b) == 0,\) then \(\text{compare}(b, a) == 0\)
– If \(\text{compare}(a, b) < 0\) and
  \(\text{compare}(b, c) < 0,\) then \(\text{compare}(a, c) < 0\)
A Generally Good `hashCode()`

- `int result = 17;`
- foreach field f
  - `int fieldHashCode =`
    - boolean: `(f ? 1: 0)`
    - byte, char, short, int: `(int) f`
    - long: `(int) (f ^ (f >>> 32))`
    - float: `Float.floatToIntBits(f)`
    - double: `Double.doubleToLongBits(f), then above`
    - Object: `object.hashCode()`
  - `result = 31 * result + fieldHashCode`
Final Word on Hashing

- The hash table is one of the most important data structures
  - Efficient find, insert, and delete
  - Operations based on sort order are not so efficient
    - e.g., FindMin, FindMax, predecessor

- Important to use a good hash function
  - Good distribution, uses enough of key’s meaningful values

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

- Popular topic for job interview questions
  - Also many real-world applications